A hierarchical layout for helping depicting software systems in a comprehensible visualization

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

Software Design

presented by

Paolo Calciati

under the supervision of

Prof. Michele Lanza

co-supervised by

Fernando Olivero

September 2011

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Paolo Calciati
Lugano, 15 September 2011

Abstract

Software systems grow in complexity as their size increases; extending or modifying a system which is not known has an high probability of introducing bugs or drifting away from its structure, hence the need for understanding the software system before modifying it. Reverse engineering provides techniques for helping developers dealing with the complexity of software systems, and one of these techniques is software visualization.

Software visualization deals with providing a visual representation of software systems. Laying out the pieces that compose a software system is not a simple task: software entities are intangible and there are many different relations connecting them that must be taken into account in order to display the composition and collaboration of the system.

A common approach in visualization techniques is to display only a single relation at a time; this results however in an incomplete picture of the system, requiring to create different views to achieve a complete understanding of it. Displaying multiple relations at once is very important to help the understanding of a system.

In this thesis we developed a layout algorithm that positions elements in such a way that reveals properties of the data being displayed. Our solution is based on a force-directed algorithm, which we adapted to place elements further or closer to each other according to the relations between them. The result is a layout that gives importance to the position in which the elements of the visualization are disposed, giving the chance to intuitively see related elements according to their proximity and/or clustering, resulting in a clear representation of the system.

Our approach is used in the implementation of *Hubble*, a tool to create the visualization of software systems providing users a high degree of customization upon it.

Acknowledgements

This thesis work is not only an artifact that I had to produce, but to me represents the five years that I invested during my graduation process. It is the end of an important period of my life, made of experiences, things that I learnt, good and bad moments, and the start of a new phase of my life. All of this has been possible because I had the opportunity of crossing my life with great people like:

Prof. *Michele Lanza*, who guided me not only in this thesis work, but also during the whole personal growth process I have been through in those years.

Fernando Olivero, who was my main support during my last steps, and has always been ready to help me.

Mele Trujillo, for your continuous support and for believing in me.

Armando Perico, the paths of our lives crossed and entwined, binding us together in the university study, as well as for the future. I have been really lucky to met you.

My parents *Alessandro Calciati* and *Anna Cabiati*, who gave me the starting point for achieving this and facing all the future challenges that will present to me.

All my beloved friends and family, who have always been there for me in the moments I needed.

Among all those people I could never forget to thank a special person who made this possible and has always been available for me: my uncle

Carlo Cabiati, THANK YOU

Contents

Figures

Chapter 1

Introduction

Throughout its lifecycle, a software system is subject to a continuous process of change. This process involves the system at various levels: its requirements, technologies, environment, architecture and design can all be subject to change.

Modifying a software system without having a sufficient understanding of it has a high probability of introducing defects or even disrupting the system's design [Rig11]. Understanding a software system means having enough knowledge of its parts, knowing what functionalities they implement, how they interact and what their dependancies are.

Understanding software systems is a very concrete problem of software engineering, since it sinks a significant part of the costs invested in software maintenance [Cor89], which is estimated to require sixty to ninety percent of the total software cost [Erl00, McK84, LS81, ZSG79].

A possible approach to contain those costs is software reengineering [CCI90], which aims at improving the design of parts of the system to make it more capable of embracing future changes [Bec00]. This does not however remove the problem of understanding a software system, which is a basic step towards modifying the software system, and, in spite of existing solutions, remains an open challenge [Wet10].

This problem is addressed by software reverse engineering [CCI90], which is defined as the process of analyzing a system to identify its components and their relations to create a representation of the system at another level of abstraction or in another form. A subdomain of software reverse engineering is software visualization.

1.1 Software Visualization

Software visualization has been defined by Price et al. as the use of the crafts of typography, graphic design, animations and cinematography with modern human-computer interaction technology to facilitate the human understanding and effective use of computer software [PSB92].

Software visualization is used by developers to assist the performing of analysis tasks [Hun96] and creating a mental model of the system. Moreover it is used to support the understanding of the system during its evolution: to reveal the design and the structure of the system (*e.g.,* the code with its execution, history, and uses) [Rei05], as well as analyzing it in order to discover its anomalies (*e.g.,* by showing coupling between classes). Software visualization is meant to support these tasks in an *effective* way [Hun96].

In his book on visual perception, Ware affirms that humans acquire more information through vision than through all the other senses combined [War04]. Consequently, visualizing software is an effective approach to support its understanding, which can be enhanced by studying how our brain processes visual input and applying this knowledge to the visual representation of the system which we are generating.

Gestalt principles describe how our perceptual system organizes disjoint elements into mental structures and groups [KN02]. The Gestalt principle we exploit the most in our approach are *proximity*, which states that close elements are perceived as grouped, and *similarity*, stating that similar elements are perceived as related; those two principles can be easily exploited when computing the layout of the elements of a system.

Wettel describes the intangibility of software, which has no physical shape or size [BE96] as one of the major challenges to program comprehension[Wet10]. Although this is true, the fact that software does not have an intrinsic shape can be seen as an advantage as well: it allows us to assign it a shape, and to choose this shape in such a way that it helps the comprehension of the software visualization, for example using it to suggest grouping of elements if they share the same shape.

1.2 Goals

Our goal is to develop a layout algorithm to visualize a software system: the algorithm must take into account multiple relations connecting the parts that compose the software system and present a representation of it showing the relations between its components.

We also listed some goals that our proposed solution has to achieve in order to allow developers to use it in an efficient way; these goals regard both implementation and visualization aspects.

- **Peripheral classes are disconnected**: the least connected classes must be placed in the outermost part of the visualization because they are not so relevant to the system.
- **Most connected classes are in the center**: the most connected classes should be the most important ones, hence should be placed in the center of the visualization.
- **Consistent layout (structural position has a semantic)**: users should be able to predict how nodes will end up being before running the algorithm. In addition to that applying the algorithm to a set of elements with same positions and edges must result in the same graph.
- **Speed**: the algorithm must complete in a reasonable amount of time.

- **Scalable**: the algorithm must be able to scale with an increased number of nodes.
- **User interaction**: we want users to be able to interact with the algorithm, generating the visualization according to their needs.
- **Display many relations**: we want to show different relations at once to directly present the whole structure of the system in a single visualization.
- **Flexible to focus**: users should also be able to focus on a specific part of the system in order to understand it better.
- **Uniform**: every level at which we apply the algorithm is treated as equal.

1.3 Approach

We decided to display three types of relations: package containment, classes' reference and inheritance relations. Package containment is useful to give an idea of the structure of the system, while the other two relations have a key importance in analyzing how the different elements are related to each other. In addition to those we show relations between packages, which are computed analyzing the relations between the classes of any two packages, and aggregating them to create the relations between the two packages.

The algorithm used to layout the elements of the software system is a customization of a force-directed layout algorithm, which has been proven surprisingly successful in producing good layouts of undirected graphs [GN98]. This class of algorithms is used on undirected graphs and lays out the nodes of the graph treating them as electrically charged particles repelling each other, and edges as springs connecting the nodes. The system obtained is then simulated as if it were a physical system and the positions of nodes when the algorithm stops, *i.e.,* when the sum of the forces applied to the system is zero, defines the position of the elements of the software system in its visualization.

One key aspect of force-based algorithms is that the final result depends on the initial positioning of the elements. Our intuition was to apply the algorithm multiple times, considering each time a different subset of elements and relations, providing a good starting position to all the elements before performing the final layout.

The approach we used is very similar to the one presented by Erni: *CodeMap* [Ern10], which we are going to analyze in the following chapter. He developed a cartography visualization of software based on vocabulary, and used Isomap [ZZ03] and Multidimensional Scaling [Lee01, HB95] to then place the elements in a meaningful way, with the goal of helping developers create a mental map of the system.

We focused on making the algorithm customizable in order to let users give different weights to different relations. After developing the algorithm we moved to the implementation of a visualization tool in order to implement our algorithm and see the results that could be obtained. What we ended up with is a hierarchical and customizable algorithm, capable of considering multiple types of relations to generate a complete layout of the system.

1.4 Contributions

The contributions of this work are:

1. The development of an algorithm for laying out the pieces composing a software system.

2. The implementation of *Hubble*, a tool which uses our layout algorithm for depicting a software system.

To create the visual representation of a software system, the first step we have to do is to extract from the system the information we want to visualize; those pieces of information are then aggregated to form a graph, which is the input to the layout algorithm which takes care of positioning elements in the visualization.

Our layout algorithm is based on a novel approach to the use force-directed algorithms on graphs, which considers the importance of initial positioning of the nodes of the graph with relation to their final positioning, and uses multiple runs of the algorithm – taking into account different elements in each of them – to achieve a better final result. Exploiting distance between two elements for suggesting their degree of coupling allows us to present more information without adding visual noise to the representation of the software system.

This approach is the core of *Hubble*, which adds to that a simple and clean interface that allows users to specify settings of the algorithm and to explore the visualization created in an easy and intuitive way.

1.5 Document's Structure

- In Chapter 2 we present some investigations that we have performed on some related works. Being software visualization a huge area of research, we arbitrarily chose three topics which have been part of our research and have been of inspiration to us while performing this work: Graph Libraries, CodeMap and UML Class Diagrams; we are going to introduce them and show what their advantages and drawbacks are.

- Chapter 3 explains in details our approach for laying out the elements that compose a software system. The algorithm is divided into three phases: in the first one the system is analyzed to extract relevant relations, which are used to build a graph representing the system in the second phase. Finally in the third phase the elements are laid out to produce the visualization of the system.

- Chapter 4 presents *Hubble*, a simple and intuitive tool we developed to implement our algorithm. We discuss its implementation, introducing how force-directed algorithms work, and then move on to the user interface. In the last Sections we present advantages and limitations of our tool.

- In Chapter 5 we discuss some visualizations created by *Hubble* upon different systems, pointing out what can be deduced from them and how those representation of the system can help users understanding it.

- Chapter 6 contains the conclusions of our work: after a short summary of what we have done, we re-state the contributions of our approach and present a list of possible future works that could be made to expand it.

Chapter 2

State of the Art

In this chapter we explore related works. Being the area of software visualization very wide, we arbitrarily present three different solutions: in Section 2.1 we present UML diagrams, which is for companies the standard language for creating models of software-oriented programs. Section 2.2 introduces graph libraries, and finally Section 2.3 we present *Codemap*, an Eclipse plug-in which uses software cartography to help developers understanding software, and whose approach is similar to the one we used.

2.1 UML Diagrams

The Unified Modeling Language is the de facto standard for the creation of models that represent object-oriented software. It provides several types of diagrams (13 in UML 2.0) that allow graphical representations of software systems.

A number of empirical studies have shown UML diagrams to be very useful during software maintenance [ABM+06, DT97]. They have in fact very spread use among developers, mainly thanks to the standardized way of representing everything, which can be immediately understood by everyone knowing it. The most used UML diagrams are *use case diagrams*, *sequence diagrams* and *class diagrams*.

Use case diagrams overview the usage requirements for a system, including the relationships of "actors" (human beings who will interact with the system) to essential processes, as well as the relationships among different use cases. UML Sequence diagrams are used to represent or model the flow for a specific use case or even just part of a specific use case, showing the calls between the different objects in their sequence. In this Section we are going to focus on class diagrams, since they are the ones which get closer to our approach when looking at what we want to represent. We are going to explain them in details in the following section.

2.1.1 UML Class Diagrams

UML Class diagrams are used to show how different entities of a system relate to each other. It model classes and their relationships, for example inheritance and composition, illustrating the static structure of a system [Sea05]. A class is depicted in the class diagram as a rectangle with three horizontal sections. The upper section shows the class's name; the middle section contains the attributes of the class; and the lower section contains its operations, which are the methods it implements. We can see this structure in Figure 2.1.

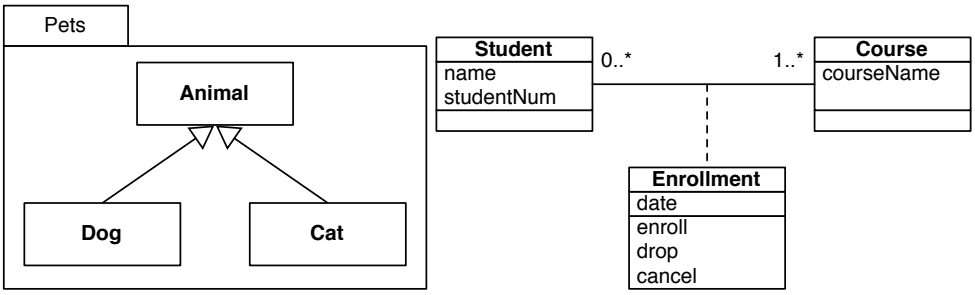

Figure 2.1. UML Class Diagram. On the left we see an example of inheritance. On the right we can instead see an association.

The most important relation for object-oriented languages is inheritance: it is displayed by an arrow with a closed arrowhead pointing to the superclass. Other relations between various objects are modeled with associations, showing the coupling between them. To show package containment classes belonging to a package are placed inside a big rectangle representing the package, while the package's name is shown in a small rectangle on the top left of the class.

2.1.2 UML in Software Visualization Tools

A lot of tools have been developed to automatically generate UML diagrams from source code, however very few of them are able to generate useful layouts that can make good use of the space used by the visualization[Eic02].

We can look at Figure 2.2, which shows an automatically generated UML diagram from Java source code using Rational Software Architect v7.0 [XW08], and at Figure 2.3, which has been generated by Rational Rose. We can notice from both the figures that the only thing that makes the placements of elements meaningful is aimed at not overlapping edges and keeping them as short as possible, which already tends to place related classes near to each other. We believe that this is however not sufficient.

From Wong and Sun's analysis on Rational Rose's automatic generation of UML Class Diagrams [WS06] it emerges that space's arrangement is quite good, however there are still some problems, for example *Thermometer* class should be closer to its superclass, while it is instead very close to *WmvcMenuItemCtl*, with whom it has no relations.

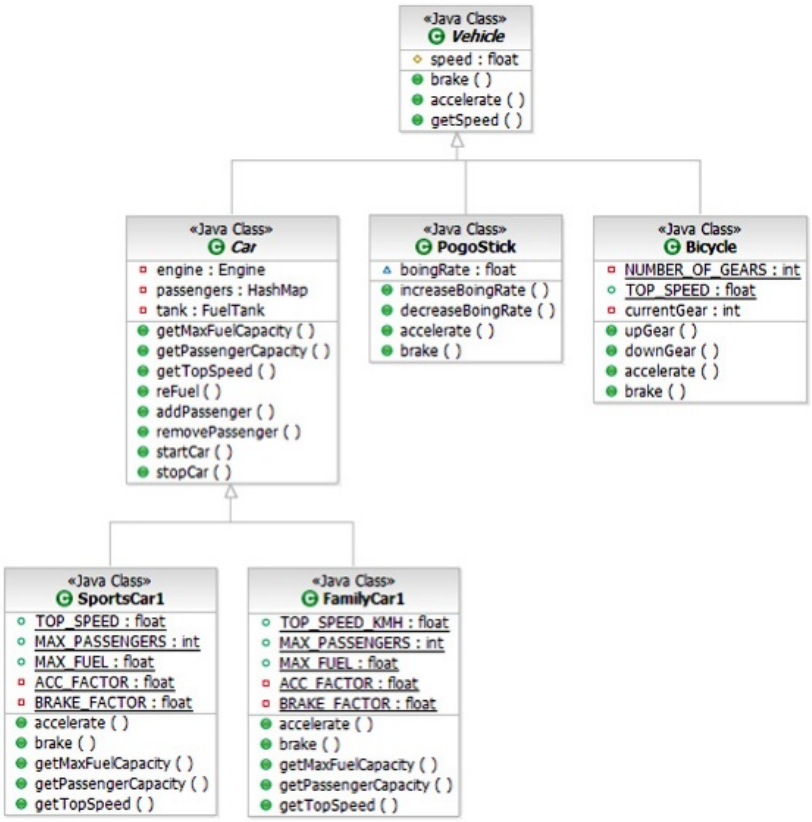

Figure 2.2. UML Class Diagram automatically generated with Rational Software Architect v7.0.

According to Tilley, UML diagrams, as a form of graphical documentation, can help software engineers to understand large-scale systems, but their efficacy depends on three main factors: syntax and semantics of UML, spatial layout of the diagrams, and domain knowledge [TS03]. And our focus of analysis goes in one of these three factors: the spatial layout.

2.1.3 Considerations

UML diagrams are a very powerful tools, widely used to communicate a system's structure and behavior thanks to its standardized language. It is however a very difficult tool to master, where still there is some confusion on the different visualization, as testimonies the very big amount of informations and articles upon it, which are sometimes also in conflict.

What emerged from our research is that, despite most of the tools used for software visualization make use of UML diagrams, the way of laying out the element is still too basic and has a large potential for being improved: layout is in fact not just about removing edge crossing and giving an aesthetically pleasing representation, but much more about how to make the process of comprehending the representation easier.

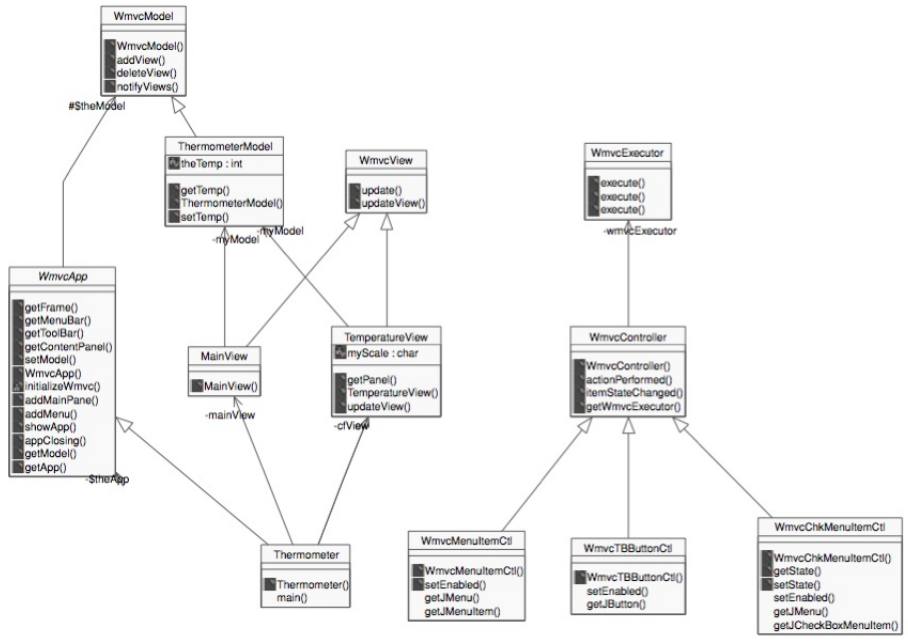

Figure 2.3. An UML Class Diagram automatically generated by Rational Rose.

In our proposed solution we tried to make a very good use of the space to solve as many visualization issues as we could, always keeping in mind to balance the amount of informations visualized with the complexity of reading the resulting visualization.

One big difference between our proposal and UML class diagrams is that UML tries to give more informations, already displaying some information relative to the class, for example its variables or methods, while our approach focus only on depicting the system without entering too much in detail. Every detail that is shown does in fact take space in the visualization, and we have to consider the fact that when we want to depict an entire system every little detail that we add gets multiplied by the number of elements on the visualization.

2.2 Graph Libraries

Graph libraries are program libraries designed to aid developers in generating visualizations and creating graphs. They are used to lay out graph, creating wide variety of images, which could also be used to depict software systems. Due to the very high number of graph libraries, in the following sections we are going to introduce a few, presenting different visualizations that can be achieved by using them.

2.2.1 D3

D3 is a JavaScript library for manipulating documents based on data. The graph realized with it and shown in Figure 2.4 is interactive: users can access it by opening a web browser and then pull nodes and see how the other ones will then start moving until the system will be back to equilibrium.

Figure 2.4. A force-directed graph created with *D3*.

2.2.2 Graphviz

Graphviz is a package of open source tool for drawing graphs specified in language scripts. It has multiple applications, and is even used in some well known software programs, for example *OmniGraffle 5*. It offers different layouts, which take descriptions of graph in DOT language scripts and generate diagrams from that. It also provides a discrete level of customization: colors, fonts, node layouts, line styles and hyperlinks can be specified by the user.

Figure 2.5 shows a software visualization representing dependencies between modification to a large program. The graph has been created with the goal of discovering which subsets of modifications could be tested separately or even removed by removing some key dependencies.

2.2.3 Considerations

These libraries offer a very solid base for developing software able to create visualization of different systems: having an algorithm for laying out graphs ready and tested allow developers to concentrate on the visualization and use it without having to deeply understand it. Implementing the layout algorithm – as we did in this work – does however offer a small advantage, even if it requires more

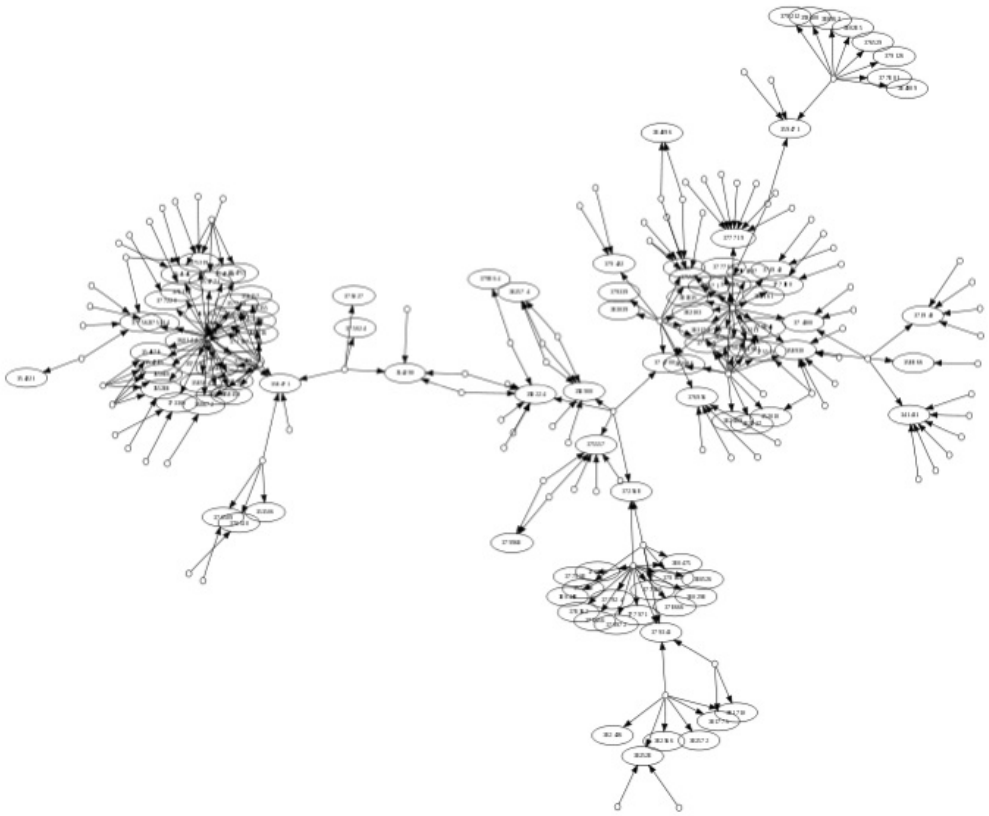

Figure 2.5. A software visualization obtained using *Graphviz* to represent dependencies between modifications to a large program.

time: using a library might not provide enough understanding of the algorithm, which would be obtained by working on it, and which would allow developers more ease of customizing it according to their needs.

The limitation of graph libraries is that they are too general for the problem we are addressing: additional code must be written in order to obtain the visualization, and this prevent them to be directly used for displaying systems. What developers need is a ready-to-use tool which they can immediately start working with and obtaining results.

2.3 Codemap

Codemap is an Eclipse plug-in presented by Erni in his master thesis work [Ern10], which makes use of software cartography in the context of an IDE to help developers establish a stable mental model of their software.

2.3.1 Algorithm Approach

The approach used by Codemap is to apply software cartography to obtain a visualization of the software that aims at helping the development team to establish a stable mental model of the project.

The first step is parsing the source files to create a term-frequency matrix according to their vocabulary. Then Latent Semantic Indexing [DDF+90] is used on the matrix to reduce its complexity. After that the algorithm proposed by Codemap makes use of both Isomap and Multidimensional Scaling to place all software elements in the visualization; this results in a two dimensional position on the visualization map for each software entity.

Isomap is an approach to reduce the dimensionality [JTL00]. It can find meaningful low-dimensional structures hidden in high-dimensional data: it represents as a spiral a two-dimensional plane that is embedded in a three dimensional space. When now measuring the distance in the two different dimensional planes it is possible to obtain two different distances for any two points, according to the way in which the distance is measured.

Multidimensional Scaling is an iterative approach to map high-dimensional data to a lower dimensionality space. It performs this task by preserving the relative distance between points in the best way possible. Multidimensional Scaling computes the dissimilarity values (a value indicating the difference between two elements, ranging from 0 if the elements are equal, up to infinity) between each pair of elements.

Multidimensional Scaling tries then to find a low-dimensional configuration where the proportions between the dissimilarities are as close as possible to the ones of elements in the higher-dimensional space. This is performed with an iterative approach, refining the solution at each step until a certain error threshold is reached.

Since Multidimensional Scaling's output depends on the initial placement of the elements, performing Isomap reduction before it and then running it on the output of Isomap grants a much better result in the end.

2.3.2 Codemap Tool

The result of performing the steps described in the above section is a two-dimensional position for each element we want to display. Starting from this cartographic visualization Codemap now computes the elevation of each pixel according to the size of the software element (computed based on the element's lines of code). The landscape is then rendered to achieve a pleasing view of the system, using colors and shading to help users in the analysis. A possible result is shown in figure 2.6.

Metrics and markers are rendered in additional layers on top of the landscape. This allows users to activate and deactivate each layer, customizing the map according to the visualization they want to achieve. These layers can be used to display a wide variety of things: a heat-map can be used to show code coverage and locations visited last; result of searches are displayed on the map by little pins; call hierarchies are displayed using an arrow-based overlay and also files being edited by other developers will appear on the map.

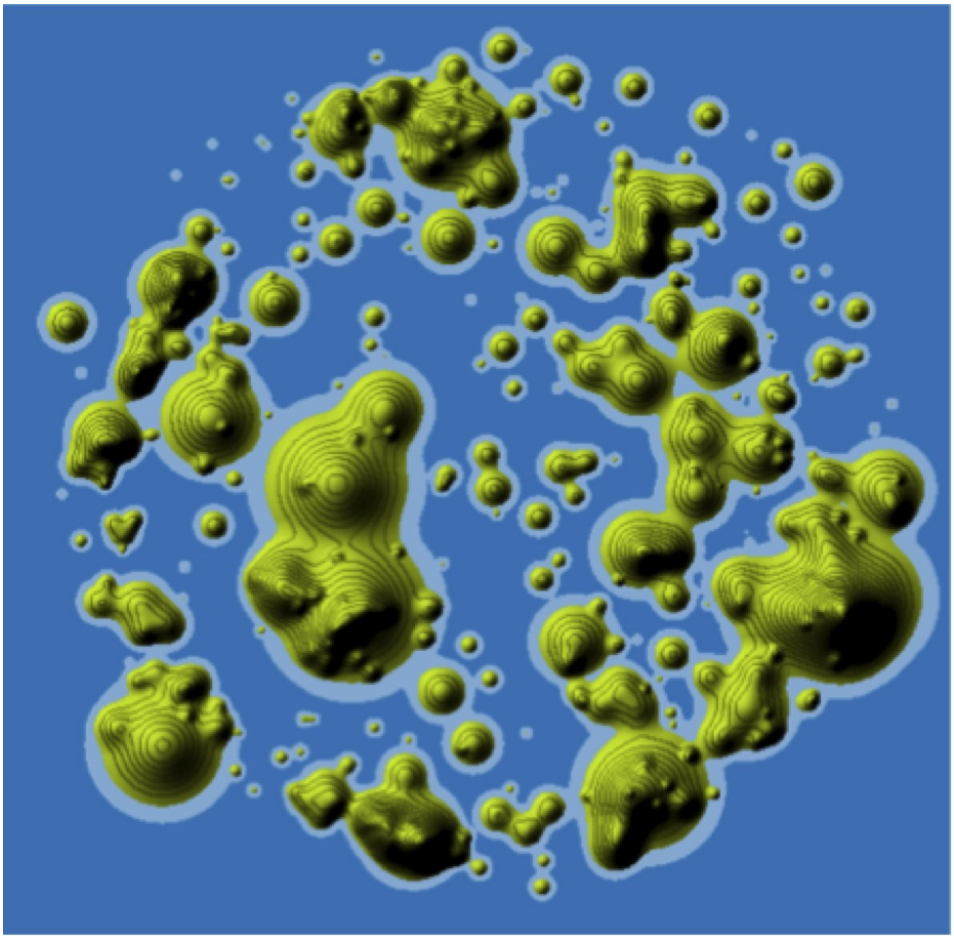

Figure 2.6. An example representation obtained using Codemap.

To achieve these results Codemap is completely integrated into the IDE (Eclipse), allowing to support the programmer in his tasks much better than a standalone analysis tool: this gives in fact the opportunity of performing some actions that would otherwise not be possible. Here we present Erni's list the programming task allowed by Codemap:

- Explore a system for reverse engineering purposes.
- Connect two Eclipse instances to show what other developers are currently working on.
- Compare different software metrics, for example code coverage vs. bug density.
- Navigate within a software system in a way which is completely integrated in the map: Codemap indicates open files on the map, and allows to open files from the map swell.

2.3.3 Considerations

Codemap's layout algorithm brings the innovation of making use of multiple steps for computing the position of the elements: having a good starting point is a great enhancement for multidimensional scaling to position them, and since this concept can also be applied to force-based algorithms, using a hierarchical algorithm is a very good starting point for creating a layout algorithm.

One important statement by Erni is not to forget for whom the tool is built: during the user study performed on Codemap it turned out that their decision to base the positioning of elements on their vocabulary was rather confusing for the participants. The lexical distance is in fact a good way to produce a visualization of the system, but this resulting visualization is not useful for developers software because it does not help too much them with the analysis and understanding of the software.

We decided to base our visualization on something which is useful to know and is always looked for when analyzing a system: knowing class coupling does in fact provide a good starting point for looking both where to extend software and where to look for bugs or anomalies.

A limitation of Codemap is that, despite considering the distance between classes (computed as lexical similitude) and translating it into a semantic positioning of the elements, it does not allow users to customize which relations to use and how much importance to give to each one, which limits the possibilities of analysis from the developers.

Chapter 3

Algorithm

Our goal is to support the understanding of a software system by creating a visualization of it in which the elements are laid out in a way that their positioning suggest which elements are related and which are instead disconnected.

In this chapter we explain in details our approach for laying out the parts that compose a software system. We present an overview of the three phases in which our algorithm is divided in Section 3.1, and then proceed analyzing them in the following sections: during the first phase, described in Section 3.2, we analyze the system to extract the elements we need for creating the visualization; with these data, in the second phase, we create a graph which we use to represent the structure of the system as explained in Section 3.3. In Section 3.4 we describe how we layout the elements that we now have, obtaining the visualization of the system. Finally in Section 3.5 we discuss the results obtained with our approach. During the analysis of the three phases, our description will be supported by graphs showing the intermediate results we reach.

While the first two phases of our algorithm are more related to the implementation, and will be discussed more exhaustively in the next Section, the third one is more conceptual and will be analyzed in detail in this section.

3.1 Overview

The layout of the part composing the software system is performed using a force-directed layout, a class of algorithms used to layout graphs by treating nodes as electrically charged particles repulsing each other, edges as springs connecting the nodes, and then simulating the graph as if it was a physical system. The nodes tend repulse each, while edges tend to pull closer the two nodes they connect. Force-directed algorithm are presented in details in Section 4.1, however a quick introduction was needed to understand how our approach work.

The input that we have is a list of the packages that we want to visualize. Starting from these packages we need to collect all the elements composing the software system that we want to take into account and then lay them out to create a representation of the system. To reach our goal and create a comprehensible visualization our approach is divided into the following three phases:

1. Analyze the system to collect elements we want to visualize and the relations according to which we will position them. To get them we have to start from the package level and go down until method level; we then go up again to class level while gathering information at each level.

2. Obtain the information still missing, such as relations between packages, which are computed from the relation between classes extracted in the first phase; we then build a graph containing all the information gathered, which we use to model the software system.

3. Lay out the nodes of the graph using the force-directed algorithm. We perform the layout in three different steps, starting from considering each package individually and laying out its classes, proceeding then to layout the packages according to relations between them, and ending up in the last step considering the whole informations gathered and positioning all classes.

The structure of our approach is presented in pseudocode in Algorithm 3.1. In each of the Sections in which we describe a phase of the algorithm, we show the corresponding part of the pseudocode.

Algorithm 3.1 High level description of our approach.

```
INPUT: packages
classes ← getClasses(packages)
classesRelations ← getRelations(classes)
classesRelations ← classesRelations ∪ getPackageContainment(classes, packages)
packagesRelations ← getRelations(classes, classesRelations)
graph ← createGraph(packages, classes, classRelations, packagesRelations)
for all pkg in packages do
  layoutPackageNodes(graph, pkg, classes, classesRelations)
end for
layoutPackageNodes(graph, packages, packagesRelations)
layoutClasses(graph, classes, classRelations)
return graph
```

3.2 Phase I: System Analysis

```
INPUT: packages
classes ← getClasses(packages)
classesRelations ← getRelations(classes)
classesRelations ← classesRelations ∪ getPackageContainment(classes, packages)
```

In this phase we analyze the system in order to extract the different elements that we use to create our visualization. In order to do that we start from the package level and go down until the method level, extracting all the elements that we want to position to visualize our system: classes and packages. We then begin from the method level and go up again until the package level, extracting

the relations which we use to define the final positioning of classes and packages.

We explored different ways for extracting relationships of the software system being analyzed; our first choice was Moose, a platform for software and data analysis [NDG05]. We switched later to using Smalltalk's reflective property to programmatically browse classes and directly extract relations from them.

The input to our algorithm is the list of packages that the user wants to display. The first step we perform is to collect all of the classes contained in those packages, obtaining all of the elements that we want to lay out. At this point we start working to get the three types of relations that we want to display: inheritance, reference and package containment relations, which will define the final positioning of classes and packages.

The first step is obtaining all reference relations going down to the method level, browsing all the methods of each class and creating a relation to every class which is referenced. We then step a level up and collect inheritance relations by asking each class for its superclass, and adding a relation from the class to its superclass, and finally cycle through all the packages and add a package relations for each package to all the classes contained in it.

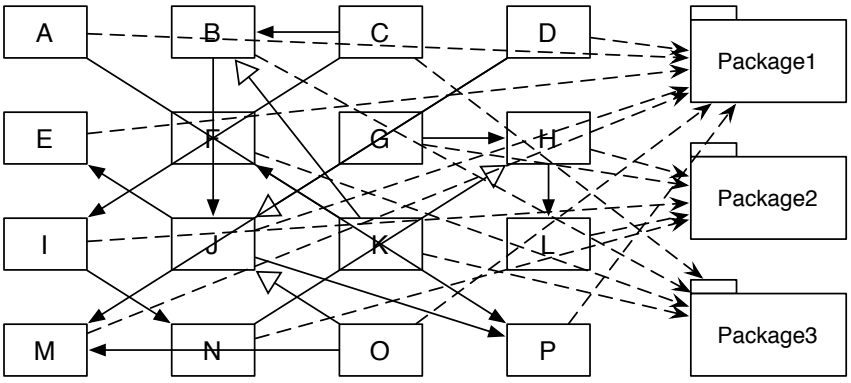

Figure 3.1. Elements we extracted analyzing the system during phase I.

Figure 3.1 shows in a visual way the pieces of information we have now: we managed to obtain the following elements:

- classes, shown as rectangles labeled with alphabet letters.
- packages, represented like packages in UML diagrams.
- class reference relation, represented as filled arrows with black arrowhead.
- inheritance relations, shown as filled arrows with white arrowhead connecting two classes and pointing to the superclass.
- package containment relations, represented as dotted arrows starting from classes and pointing the package containing it. In the following Figures this relations will be however represented by placing the class inside the package.

We display all those elements in a grid, showing all relations connecting them: with this layout it is almost impossible to understand the structure of the system, and our goal for the remaining part of the algorithm is to display those elements in the way which helps the comprehension of the software system.

3.3 Phase II: Graph Creation

$packagesRelations \leftarrow getRelations(classes, classesRelations)$
$graph \leftarrow createGraph(packages, classes, classRelations, packagesRelations)$

We compute the remaining relations and then build the graph representing the system, in order to build the graph representing the structure of the system. We then use this graph as input of the layout algorithm in the following phase, and in the end of the three phases the graph will contain the final position of all of the elements that we want to visualize to represent the system.

To collect the two missing types of relations we need to go up one level, and end up where we started from: at the package level. Package containment relations are computed by cycling all packages, and for each package creating a relation from it to each of its contained classes.

At this point we already have all relations of classes, and can compute the relations between packages. To do it we cycle all packages and, for each package P, we parse all classes it contains, analyzing relations between classes computed in the previous phase; for each class of another package related to the classes of P we create a relation from package P to that package. Additional relation found between two packages will increase the weight of the relation connecting them.

The two relations that we are still missing are package containment relations, and the relationships between packages. To compute the relations between packages we cycle them all, and for each package P, we parse all his classes and create a relation from P to each package which contains a class referenced by a class inside package P.

At this point we build the graph, transforming all packages, classes and relations, in the corresponding nodes and edges of the graph, as explained in Section 3.3.1. All the positions of nodes are generated randomly, and will be updated in the next phase, where the graph we obtained is used as data model.

3.3.1 Data Model

We decided to model the data we are working with as a graph: a collection of nodes and edges; we use nodes to represent the elements of the software system that we have to position, which are classes and packages, while edges represent the different types of relations connecting those elements. Figure 3.2 illustrates an example graph which encodes a system, which we are briefly analyzing to better present what is stored inside it.

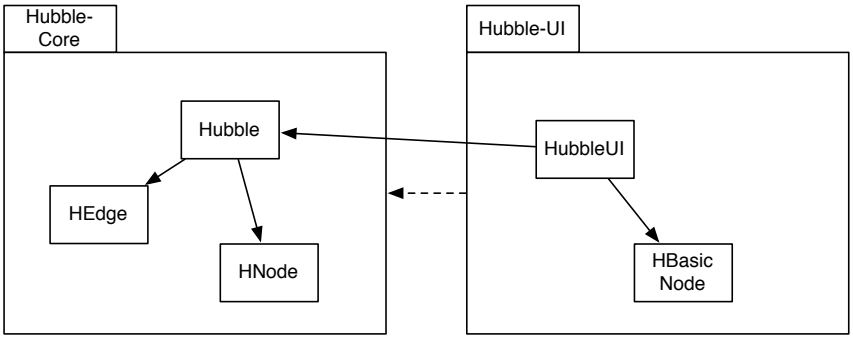

Figure 3.2. Example structure of a graph which encodes a system.

In the example graph we can count seven nodes – two packages, *Hubble-Core* and *Hubble-UI*, containing three and two classes, respectively – and some edges connecting them. We can notice that since there is an inter-package reference from *HubbleUI* class to *Hubble* class, there is also a package reference from *Hubble-UI* package to *Hubble-Core* package.

This data model allows us to have everything stored as a collection of nodes and edges, simplifying the complexity deriving from the different elements of the system we are exploring, and allowing us to only have to handle two different objects. The final position of each nodes after the algorithm completes defines the position that element will have in the visualization of the system. In the following subsections we are going to analyze more in details the elements that compose our graph.

Nodes

In our graph we dispose of class nodes and package nodes, which are used to show relations at different levels in our algorithm. The former are used to represent classes, showing the relations between them in the visualization of the system according to their spatial distance: the closer two class nodes are, the more the two corresponding classes are related to each other.
Package nodes work basically in the same way but at a different level: they show the relations between packages according to their positioning, and moreover they condition the positioning of the classes in the final step of the layout phase: each class node in fact has a relation connecting it to its containing package, whose goal is to prevent the class node from drifting too much away from the package node during the layout steps.

Edges

In our model we use four different types of edges: reference, inheritance, package containment and package relation. All of them have a weight attribute which can be specified by the users to enable them to customize the visualization by giving more importance to a specific relation. Edges are grouped in two different categories, relation edges and binding edges, according to how they influence the positioning of classes during the layout phase of the algorithm:

- **Relation edges**. This class of edges includes reference, inheritance and package relation edges, which represent the relations between two classes or packages. They are used to specify which classes should be drawn near each other in the visualization of the system.
- **Binding edges** are used to represent the belonging of a class to its containing package. Their use in the algorithm is to prevent classes from being positioned too far from their package.

3.4 Phase III: Layout

for all pkg in $packages$ **do**
  $layoutPackageNodes(graph, pkg, classes, classesRelations)$
end for
$layoutPackageNodes(graph, packages, packagesRelations)$
$layoutClasses(graph, classes, classRelations)$
return $graph$

The last phase of our algorithm deals with taking the graph created in phase II and laying out its nodes according to its edges. This is done in three steps: in each of them we focus on different types of relations to layout parts of the graph, similarly to the hierarchical approach of phase I to extract information from the system being analyzed. During the layout phase we however start from laying out each individual package, and then start aggregating elements until we lay out all the elements of the graph.

The following subsections present the three steps, whose starting point in this phase is basically what we shown in Figure 3.1, to which we have to add the references between packages that were computed in phase II. The explanation of each of the steps of our algorithm is accompanied with a Figure presenting the results of applying that steps; each image has the relations which are not used blurred out, making the relations which were considered to position the nodes stand out.

Layout Class Nodes of Each Package Individually

In the first step we want to position all classes inside their containing packages, and lay them out considering relations internal to the package: we consider each package individually, and for each of them we perform the following tasks:

1. We lay out all of the classes contained in the package according to the intra-package relations between them *i.e.,* leaving out all the relations that relate classes belonging to two different packages.
2. The package node is translated in such a way that its center ends up coinciding with the center of the classes we just laid out.
3. The radius of the package node is set to have all the classes of the package positioned inside its range.

The result of this step is shown in Figure 3.3: it already looks much more understandable than what we had before performing the step, with classes already positioned inside their enclosing packages.

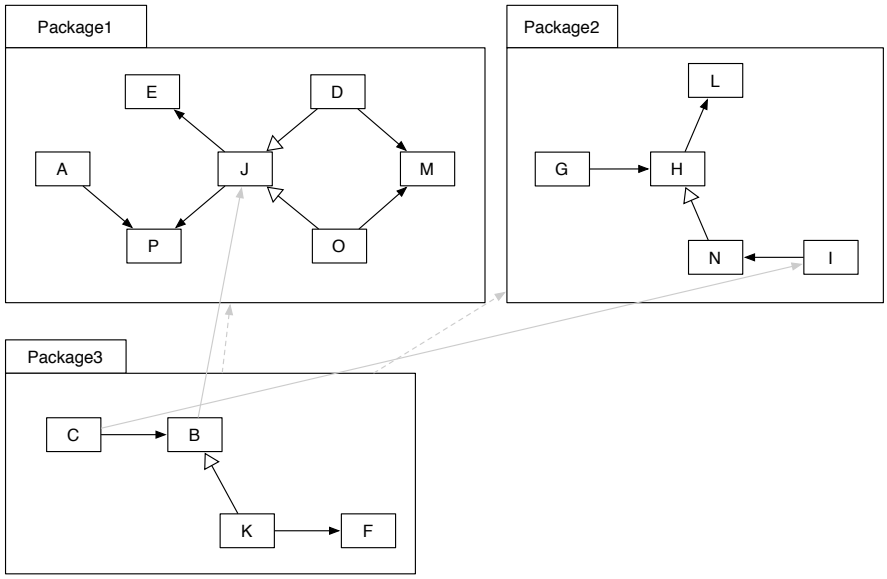

Figure 3.3. Representation of the graph representing the system after the first layout step.

Layout Package Nodes

The goal of the second phase is to lay out packages according to packages relations; this process is very similar to the one applied for laying out the classes of each package in the first step. To achieve this result we run the layout algorithm considering just package nodes and package relations edges.

The result is shown in Figure 3.4. Just looking at packages we can get a general idea of the high-level structure of the system. We can in fact see that *Package3* is positioned in between the other two, showing that it is connected to them both, while the other two do not have reference to each other; since package nodes' position already suggest the relations between them, they are not going to be moved further.

Layout Class Nodes

In the last step we want to lay out class nodes to show relations between them according to their distance in the visualization. We run the algorithm keeping in consideration class nodes and all edges computed in phase II; package nodes are not used because we don't want them to be moved anymore, however package containment edges are used to prevent classes from moving away from their containing packages.

The result obtained after this step is shown in Figure 3.5. We can see from it that classes' positioning now resembles the disposition of packages: since classes contained in *Package3* reference both classes of *Package1* and classes *Package2*, the classes of those two packages got positioned closer to *package3*.

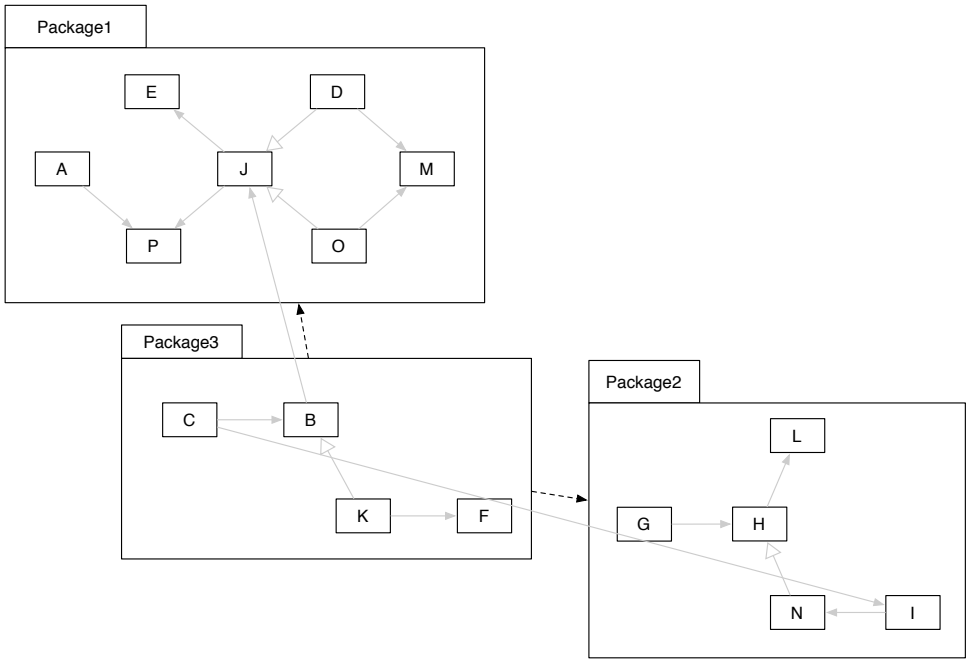

Figure 3.4. Representation of the graph after laying out package nodes.

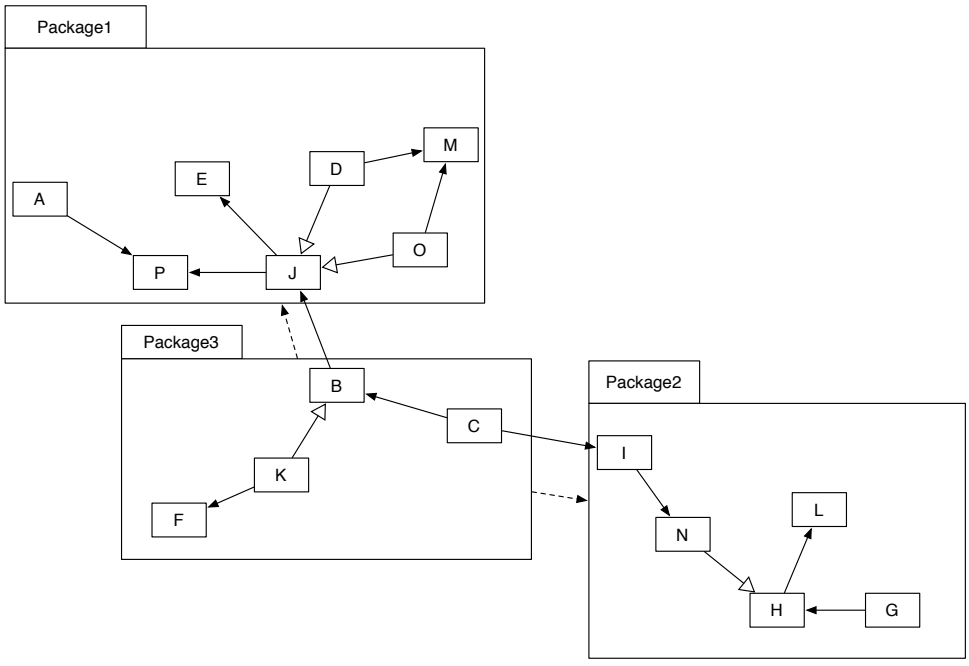

Figure 3.5. Representation of the graph after laying out class nodes. This coincides with the visualization of the system.

3.5 Achievements

Visualizing the software system using our approach allows the achieving of some of the goals presented in Section 1.2, as well as obtaining other positive results which were not part of our initial set of goals.

Class Positioning

Among the set of goals proposed inside the Introduction, two of them were related to the positioning of classes:

- Most connected classes must be positioned in the center of the visualization.
- Least connected classes must be laid out in the outer side.

In this Section we are going to explain how these two goals are achieved. Even if the goals were regarding classes, everything presented in the following paragraphs is valid for packages as well.

Thanks to the use of force-directed algorithm to lay out classes, our algorithm managed to achieve both goals; in order to understand how this is accomplished we must first explain what happens during the layout phase. That phase is divided into three steps, and since the final positioning of classes happens in the third one, our analysis is only considering it.

Due to the nature of the layout algorithm, the more edges a class has, the more classes will be placed near it. In the case a group of related class nodes has a high number of edges those classes will end up forming an agglomerate of classes *i.e.,* a group of classes linked, directly or indirectly, to each other. In the case this amount of nodes and edges is elevated, there would be no way to layout all the elements respecting the distances each pair of classes connected by an edge normally has, causing the agglomerate to look like a group of classes disorderly positioned.

Since our force-directed algorithm keeps in account weight of nodes when positioning them, an agglomerate of nodes will have a an higher overall weight than a single node, or a group of few of them. This would result in having classes related with the agglomerate – classes which have a relation with one class belonging to the agglomerate – being attracted by it and behaving like satellites, ending up near it in the visualization of the system. This places the agglomerate in the center of the visualization with relation to the nodes connected with it, directly or indirectly; in case there were disjoint subsets of the systems, each of them would end up displayed around its biggest agglomerate.

in addition to having the most connected classes in the center of the visualization, our algorithm also places a node with many connections in the middle of all the nodes connected to it, assuming that there are no other forces in play. Assuming node A is connected with other five nodes, those would be placed all at the same distance from node A, and due to the repulsion force each of them would be pushed away from each other. As a result of this node A would end up being in the center, with the other five nodes radially disposed around it.

What has been said in this Section proves our first goal of positioning most connected classes in the center; the achieving of the second goal is much easier to explain: unconnected classes due to

the repulsion force of nodes, will be pushed away from all other classes, ending up being placed in the outermost part of the visualization.

Multiple Relations Shown

Another goal we imposed ourselves was to show different types of relations. This goal was fully achieved: we can in fact display different types of relations:

- Package containment
- Inheritance
- Class reference

In addition to those we also display the degree of coupling between packages, which is obtained by analyzing the relations between the classes contained in pairs of packages. This achievement is very important when visualizing a system because it allows developers to better understand its structure, facilitating them to analyze and inspect it, and enabling them to make mental connections that would be otherwise impossible to make.

Multilevel Analysis

In our final visualization we manage to lay out in a single image both nodes and packages, positioned according to the multiple relations connecting them. This allows us to analyze the system at two different levels: at package level and at class level.

The former options provides a high-level overview of the software system, which shows how the different packages interact between each other. With this information a developer has already some understanding of the structure of the system, and this knowledge will also support him when performing a lower level analysis going down to the class level.

At class level we can perform a more in-depth analysis of the system, which allows understanding the degree of coupling between classes, exploring more in details the connection between different parts of the system, which could be already recognized with the high-level analysis.

The advantage of having the possibility to perform a multilevel analysis satisfies the goal of creating a visualization flexible to focus: developers can get an overview of the system, and then explore the parts of it they are most interested in.

Chapter 4

Tool

In this chapter we analyze and discuss Hubble, the tool we developed to implement our algorithm.

In Section 4.1 we discuss in details the force-directed algorithm, which is the base of our layout process. Section 4.2 introduces enhancements made to the force-directed algorithm. In Section 4.3 we discuss the user interface, presenting the control panel and the display screen. Finally in Section 4.4 and Section 4.5 we discuss the result of our implementation, presenting a achievements and limitations of it.

4.1 Force-directed Algorithm

Force-based or force-directed algorithms are a class of algorithms for displaying graphs in an aesthetically pleasing way: this is achieved by assigning force among nodes. The most common to do this is to assign forces as if edges were springs connecting the nodes (Hooke's law), and nodes were electrically charged particles (Coulomb's law). The entire graph is then simulated as if it were a physical system: the forces iteratively are applied to the nodes pushing them apart or pulling them close together until the system reaches an equilibrium state. The physical interpretation of this equilibrium state is that all the forces are in mechanical equilibrium.

In the case of spring-and-charged-particle graphs, the edges tend to have uniform length (because of the spring forces), and nodes that are not connected by an edge tend to be drawn further apart (because of the electrical repulsion), which are goals that we imposed ourselves at the beginning of this work.

An important advantage of force-based algorithms is *flexibility*: force-directed algorithms can in fact be easily adapted and extended to fulfill additional aesthetic criteria: this makes them the most versatile class of graph drawing algorithm, and also is the reason we have chosen this algorithm to work on our graph. We took in fact advantage of flexibility of this class of algorithms to customize our implementation, allowing users a high degree of freedom in the algorithm.

There are also some limitations of force-based algorithms: high running time and poor local minima. Typical force-based algorithms are generally considered to have running time equivalent to

$O(v^3)$, where v is the number of nodes. This is because the number of iterations is estimated to be $O(v)$, and for each iterations all pairs of nodes need to be visited and their mutual force computed, which has a time estimation of $O(v^2)$.

Force-directed algorithms produce graphs with minimal energy, in particular one whose total energy is a local minimum. This local minimum found can be in many cases considerably worse than the global minimum, and this would result in a low-quality graph. This problem is partially solved by our hierarchical approach: since the result of force-directed algorithms depends on the initial position of the nodes involved in the algorithm, performing additional runs prior to laying out classes results in a much better result than just running the algorithm with random starting positions.

4.2 Implementation

In this section we discuss a few important implementation-related decisions, which have been important during the development of our tool.

We explored different ways of extracting the relationships within the program to layout. Our first implementation of the algorithm was using Moose, a platform for software and data analysis.we switched at a later stage to directly extracting relations inside Smalltalk, which allows us to programmatically browse classes and directly ask objects for references and inheritance. This way we do not need to use external tools, having a way to access data which is directly embedded into the programming language, which is independent from third party's software, faster and simpler.

In our implementation of the force-directed layout algorithm we make use of binding nodes, a concept we introduced with our implementation: a binding node is a node which does not necessarily represent an element of the graph and its only purpose is to be used as a basis to apply forces to other nodes in order to obtain the desired positions on the nodes.
Bindings nodes are used to show package containment: we make each binding node attract all of the nodes belonging to a package to it. In order not to impair the positioning of the other elements a binding node has an attraction threshold under which its force is not applied. This way elements are free to arrange themselves according to other relations shown, however nodes which are too far will begin getting attracted in order to contain the package in a reduced space.

4.3 User Interface

We kept our user interface very simple in order to be intuitive and provide a quick way of performing the different actions possible on the program. It consists of two parts: the control panel and the display screen. They are shown together in Figure 4.1.

4.3.1 Display Screen

A screenshot of the display screen is shown in Figure 4.2. Packages are shown as colored bubbles, with their size depending on the number of classes they contain. Packages are placed according to

their relations, similarly to what happens to classes, so looking at packages can give us a very high overview of the system.

Classes are represented as rectangles, colored with the same color of their package. Class nodes have an algorithm which adapts the name of the class to the space available in the label, only displaying the part of the name which fits it, and displaying a bigger part of it as the user zooms in and the label's size increases.

All of the elements displayed in the screen can be dragged and re-positioned to give the possibility of modifying the automatically generated view in order to better analyze the system.

Left-clicking a class opens a menu where users can choose to inspect it, or to open a browser on that class, providing developers a way to explore and modify the code without having to switch to another program. This greatly enhances its usability, providing developers a quick access to source code, and allowing them to use Hubble's visualization as starting point for opening classes and modifying them.

Another menu can be shown left-clicking an empty area, this allows users to toggle edge displaying, return to the starting zoom level, which allows to see the entire area, and close the display screen. The two menus are the only way users have to interact with the visualization from the display screen. All other tasks can be done from the control panel.

4.3.2 Control Panel

The control panel is used to control the visualization. We are now going to present what it allows to do with the help of Figure 4.3. It is divided into four different sections:

The top part allows to choose which package to visualize, and presents a list of the packages currently selected; upon pressing the "Choose packages" button, a new window pops up, and users can perform a filtering on packages by their name, and add or remove them from the current selection. After deciding what they want to display, pressing the "OK" button will start the computations, and once they will finish the Display Screen will appear, presenting users the visualization.

The central parts of the Control Panel contains some sliders which are used to adjust the weight modifier for the different types of edges that connect classes. The higher a modifier is set, the more two nodes connected by that type of edge will be near in the visualization. This possibility to specify the weights of the various relations was inspired by the work of Noack and Lewerentz [NL05].

After that we can find a list of options users can set in order to specify settings for the algorithm; the setting users can define are:

- Repulsion force. Specifies the modifier of the repulsion force.
- Ideal edge length. Expresses the ideal length of edges, which is the distance nodes connected by an edge will try to be placed at.
- Repulsion radius. Specifies how far from each other nodes will be affected by the force of repulsion.

- Max iterations. Expresses the maximum number of iterations that the algorithm will run.
- Desired iterations. Expresses the number of desired iterations that algorithm will try to run, this
- Attraction distance. Specifies how far the binding attraction of package nodes will begin to take effect.

Finally the lower part is dedicated to the controls of the camera, that allows users to easily and intuitively explore the display screen. There are four buttons, each of them for moving the camera along one direction.

4.4 Achievements

In this section we present the main results we reached with our implementation; first of all we concentrate on the results related with the list of goals we discussed in Section 1.2.

The display screen offers a limited range of options for navigating the visualization, hence is very intuitive and simple to use and still provides enough functionalities to exhaustively explore the system. Users can also customize all the parameters of the algorithm, both regarding the specifications of the layout algorithm *e.g.,* number of iterations or repulsion force, as well as giving different weights to different relations between classes. This high freedom of customization helps reaching a very important goal, which is the one of building a tool flexible to focus *i.e.,* which allows developers to concentrate the analysis on a specific part of a software system in order to explore it in details.

Flexibility does however not only lie in the customization of the parameters: our tool allows in fact to get a general representation of the system, which presents its structure, and then gives developers the possibility to zoom in and explore it more in details, concentrating on the relations between single elements.

In addition to those results, which were proposed as goal, we also managed to provide a tool which makes use of the capabilities of our brain to show additional information through the use of coloring and proximity. Coloring is used to represent package containment: classes contained in a package will in fact be represented with the same color of their enclosing package. This enables users to immediately recognize to which package a class belongs to without having to draw any edge or connection, which would have added visual noise to the visualization. Proximity is used to show if two elements are related according to how distant they are, even if the edges are not being displayed.

4.5 Limitations

Our tool presents some limitations, which we are going to list according to their importance:

1. **Class Nodes Label**. *Hubble* uses an algorithm for adapting the label of classes to the space available for displaying it, cutting it and adding trailing dots if it does not fit in the class node, to prevent it from exiting from the classes. However when the system is zoomed out

the classes are so small that their names are reduced to only a couple of letters, making it impossible to distinguish them. When zooming in the problem persists and, since classes usually have long names, even with the maximum zoom in allowed it is often impossible to distinguish classes or to find a particular one.

2. **Clustering**. Since the goal of our system is to suggest relations between elements based on their proximity, developers must be able to clearly see the distance between two nodes to understand their degree of coupling. This is however true only if there is not a cluster of nodes, . We will present an example use case in which we occur into this problem in Section 5.4.

3. **Scaling**. Force-directed algorithms work best with a medium-sized graph *i.e.,* containing around fifty nodes. When this number grows the algorithm begins to get slow. This problem is due to the nature of the algorithm, and even if we managed to speed it up with a large amount if nodes or relations it will take much time to complete.

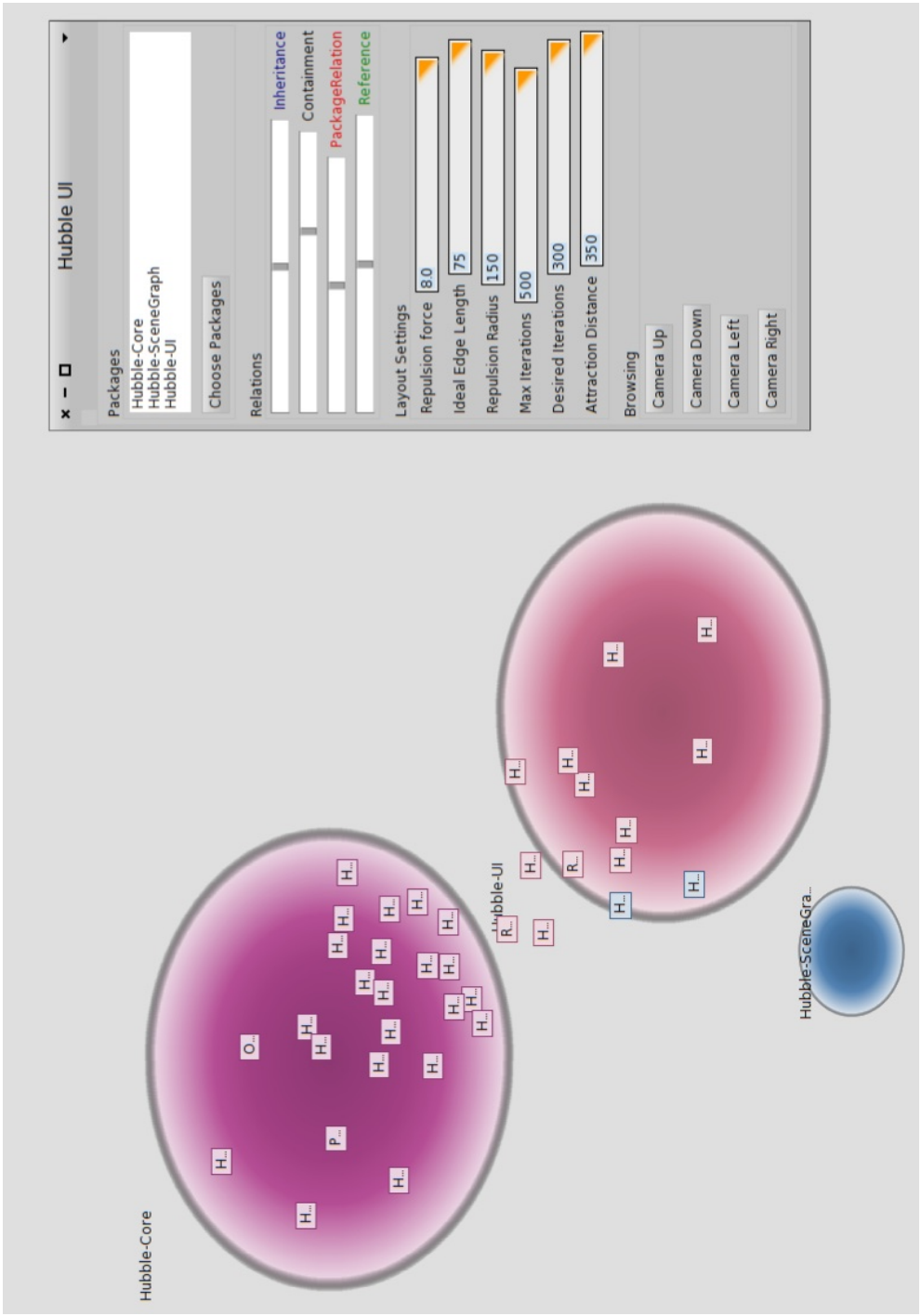

Figure 4.1. A view upon *Hubble*'s user interface.

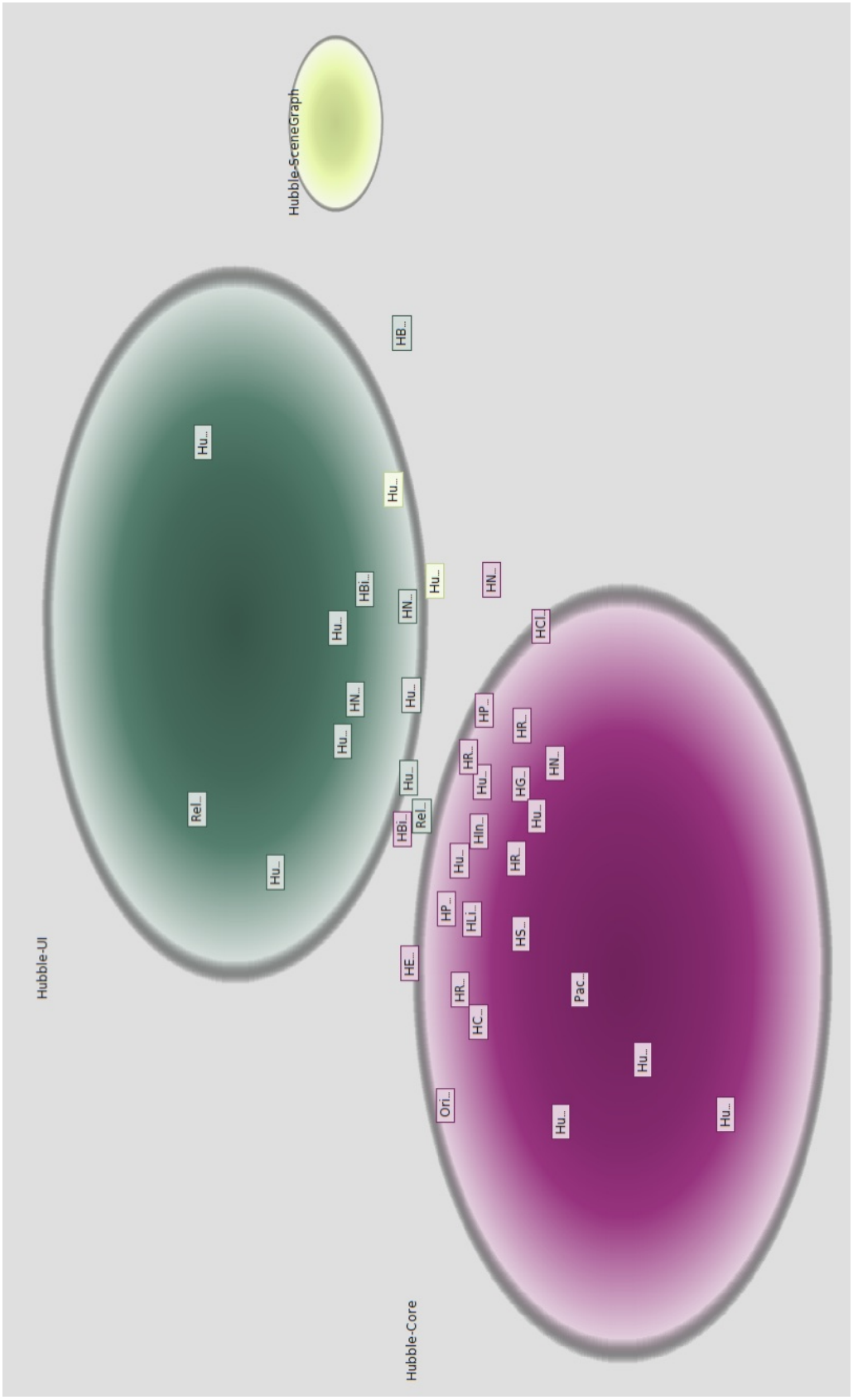

Figure 4.2. An example view of Hubble.

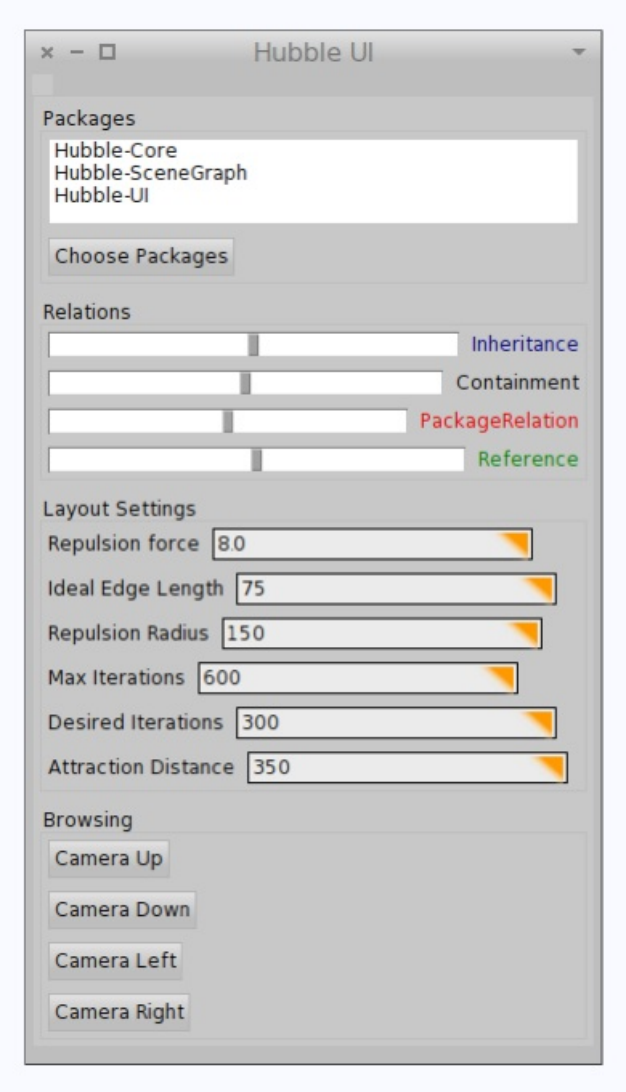

Figure 4.3. Hubble Control Panel

Chapter 5

Evaluation

In this chapter we evaluate our tool by visualizing different systems, and then analyzing the obtained representations.

The first system we analyze, in Section 5.1, is *Hubble*, to provide an example use and also show how its structure looks like. After that we visualize *Gaucho* in Section 5.2, which is being used by *Hubble*, and then in Section 5.3 we show the two systems together to analyze their interactions. Finally in Section 5.4 we try to visualize the *Collections* subsystem of Smalltalk.

5.1 Hubble Analysis

To show what the product we have developed is capable of doing, we are now going to use it to analyze itself, taking advantage of this demonstration to also show and present in a few more details details what we have done.

5.1.1 General Analysis

We open the program and immediately press the "Choose packages" button. We specify "Hubble" in the search box, add the three Hubble packages to the list of selected packages, and press the OK button to start computing the visualization.
After waiting a very few second while classes' references are cached and the algorithm is run, the visualization appears; we adjust the camera a bit to have it in the center of the scene and we obtain what is shown in Figure 5.1.

The first thing that stands out when we first see the image is that there are three different packages, visualized as colored bubbles, and that almost all of the classes have been placed in the middle of the image.

We can distinguish at a first glance the three packages, visualized as colored bubbles, and recognize which classes belong to which package: the two packages "Hubble-UI" and "Hubble-Core" are bigger than "Hubble-SceneGraph", meaning that they contain much more classes: "Hubble-SceneGraph" package has in fact only two classes in it.

Another thing we can understand from the image is that the *Hubble-UI* package has relations with both the other two packages, while *Hubble-Core* and *Hubble-SceneGraph* have no relations between themselves and are in fact placed far from each other.

In contrast to the central cluster of classes, we can also find some classes which are placed on the outside area: we suppose those classes don't have connections with other classes of the system. Since we are displaying the whole Hubble and these classes are not referenced, the most probable option is that those are classes whose code has been moved elsewhere, and that those are now disjoint from the system.

After a little bit of investigation in the code we see that in fact these are classes are not used anywhere, and that their functionality has been moved in other places, so they could be removed without any harm.

5.1.2 In-depth Analysis

The visualization of the global system allowed us to get the information presented above, however to have a better understanding of the system we need to explore it more in details: we zoomed in the central part, where the two big packages get near each other, to try to understand something more from the many classes in that area, as shown in Figure 5.2.

We can clearly see from the image that there are only a few classes, namely *HubbleUI*, *Hubble-LayoutSpec* and *RelationsMapper*, from the *Hubble-UI* package that are close to the *Hubble-Core* package, so we suppose that these are the classes that bridge the two packages.

HubbleUI class is also in the middle of many other classes, and thus we suppose that all of them are related with it, according to the considerations we already presented in Section 3.5, where we discussed the tendency of elements connected with many other to have them radially disposed around it.

Because of the high density of classes in the area, it could happen that some classes gets displayed near each other even if they don't have any relation, but just due to the fact that they are in connection with some other classes that are instead related. In order to prove if our observations were right we toggled edges displaying on while still focusing the same area of the visualization, as shown in Figure 5.3. The first thing we checked is that *HubbleUI* class is indeed the one which effectively connects the *Hubble-Core* package with the *Hubble-UI* package: we can in fact count eleven relations with neighboring classes.

We can see that also the other two classes *RelationsMapper* and *HubbleLayoutSpec* have connection with the *Hubble-Core* package, despite being in the *Hubble-UI* package. We decided then to quickly browse the two classes and we found out that they are used to store the settings of the algorithm, which are decided by the user in the control panel and must then be used in the *Hubble-Core* package before the algorithm is run.

5.2 Gaucho Subsystem

The first thing we want to analyze is the subpart of *Gaucho* that we use in our system: we did not need to import the whole *Gaucho*, so we would like to explore how what we needed to connect to *Hubble* works.

Figure 5.4 shows how this subpart of *Gaucho* looks like, however being it a larger system than *Hubble*, there are more packages and connections; as a result all of the classes of the various packages got attracted to the center of the visualization. Since we would like to analyze it at a more general level we decided to increase the weight of package containment relations to see classes tidily closer to their packages. The result of this new visualization can be seen in Figure 5.5.

If we wanted to keep all the classes inside their respective packages on the visualization we could increase even more the weight on package containment relations, however the result already obtained is good enough to learn something from the visualization.

We can immediately notice that two packages, namely *GauchoMorphic-SystemSnapshots* and *GauchoMorphic-Icons*, are not connected to the rest of the system: their classes are in fact still contained in the package, meaning they have no relations with other classes. We can also notice that in package *GauchoMorphic-Icons* classes are not even connected between themselves.

In contrast with the two packages just described, *GauchoMorphic-Core* is clearly the central package of this subsystem: is easy to see that it is connected to almost all of the other ones, and has in fact been placed in the center of the visualization.

Being all connected classes attracted in the center, the classes which stayed on the outer packages are classes which are not connected with the rest of the system. After this analysis of the *Gaucho* subsystem, we can now visualize it together with *Hubble*, focusing on their interaction.

5.3 Interaction between Gaucho and Hubble

After selecting the packages we want to display and waiting more or less a minute, the visualization of the two systems appears as shown in Figure 5.6.

With classes of the two systems already cached, *Hubble* took a bit less than one minute to visualize them, and this is an already good result in terms of scaling since we are working with slightly more than a hundred elements if we combine the two systems.

At a first glance we immediately recognize that the visualization is very similar to the one of *Gaucho* presented in the above section in Figure 5.5: it has in fact the same structure. The *GauchoMorphic-Core* is again in the center of the image, with other packages surrounding it; we added in fact just three packages, which cannot modify too much how the *Gaucho* system is represented.

This time however also the *Hubble-Core* package is in the center, overlapping by a large amount

of space with *GauchoMorphic-Core*. This is because they are both related with the packages surrounding them, and the algorithm we developed tends to place such classes in the middle or on the sides of the elements they are connected with, as we previously explained in Section 3.5.

The other two packages of the *Hubble* system took place in the circle around the cores of the two systems. They have structurally the same disposition thay had when we explored the system in Section 4, where *Hubble-UI* connected to both of the other two, which are instead not neighboring each other.

With these two element we can now state that our algorithm always processes the data and returns visualizations that are very similar to each other when run on the same system multiple times. This is clearly valid for the general structure of the system, as the positioning of single classes depends on the random starting position of elements on the first run, and cannot therefore be always identical.

An alternative view of the two system is displayed in Figure 5.7 to show how the random starting position can influence the final positioning and that in case this happens the structure of the system is however still preserved. All of the statements that we have done based on Figure 5.6 are in fact still valid.

The only structural difference between the two visualizations is the positioning of *Hubble-Core*, which is not anymore in the center of the packages. This happens because it was not connected to all of the packages that were surrounding it in Figure 5.6, but only to some of them, and therefore it was just an aleatory event that he did get positioned in the center. *Hubble-Core* is however in the central part of the visualization also in Figure 5.7, meaning that it is one of the most connected packages, as opposed for example to *Hubble-SceneGraph* package, which is only connected to *Hubble-UI* package and therefore placed on the outer part.

One interesting peculiarity of the visualization of Figure 5.7 is the interaction between *GauchoMorph* class and *GauchoMorphic-Events-Conditions* package. What happens here is that *GauchoMorph*, which is related with a lot of classes, is also linked to *GMCondition*, which has connection with many of the classes of the package. The two classes would tend to be pulled close one another by the algorithm, however they are also being pulled away from each other by their other relations. As a result we can see that *GMCondition*, which is the only class in the package that has relations outside it, got pulled out of its package, and all of the classes it is related to have been pulled to the edge of *GauchoMorphic-Events-Conditions* package.

This is possible because *GauchoMorph* and *GMCondition* are quite far from each other, and so the edge connecting them is strong enough not only to pull the two classes close, but also to cause a movement of many other classes in the *GauchoMorphic-Events-Conditions* package. The edge was however not strong enough to pull other classes out of that package, as each class who would have been gone out of it would have triggered the binding force that would have pulled it back to the package. We can see this in more details in Figure 5.8, where *GauchoMorph* class is identified by a red oval, while *GMCondition* is identified by a black one.

5.4 Collections

With this example we want to show some of the limitations of our tool, but emphasizing the fact that they are limitations relative to our implementation, and re-arranging the algorithm would enable us to solve them.

Figure 5.9 shows the visualization of the *Collections* packages included in the last release of *Pharo*, which is the framework in which *Hubble* was developed. What we wanted to show with this example is the interaction between *Hubble* and the *Collections* package.

Already when selecting the classes we noticed a high number of packages, we could in fact count fifteen of them, already leaving out all of the tests, since including them would have more than doubled that amount. At a first gaze at the visualization we can immediately notice from the high overlapping of packages that there are a lot of relations between them.

This is confirmed by the fact that almost all of the classes are concentrated in the middle area, but this is a different situation with respect to the analysis of Figure 5.6. The previous analysis was also showing all classes positioned in the center, but in that case the connections were still visible, while now classes are so clustered that is impossible to see what is connected with what.

Figure 5.10 shows a detailed view of *Collections* packages: we zoomed in the cluster of classes and turned edge showing on to emphasize the extremely high number of relationships taking place. Our system cannot show in a good way such a high dense graph, and this is one of the limitations of our approach, which is much better at working with a smaller number of relations.

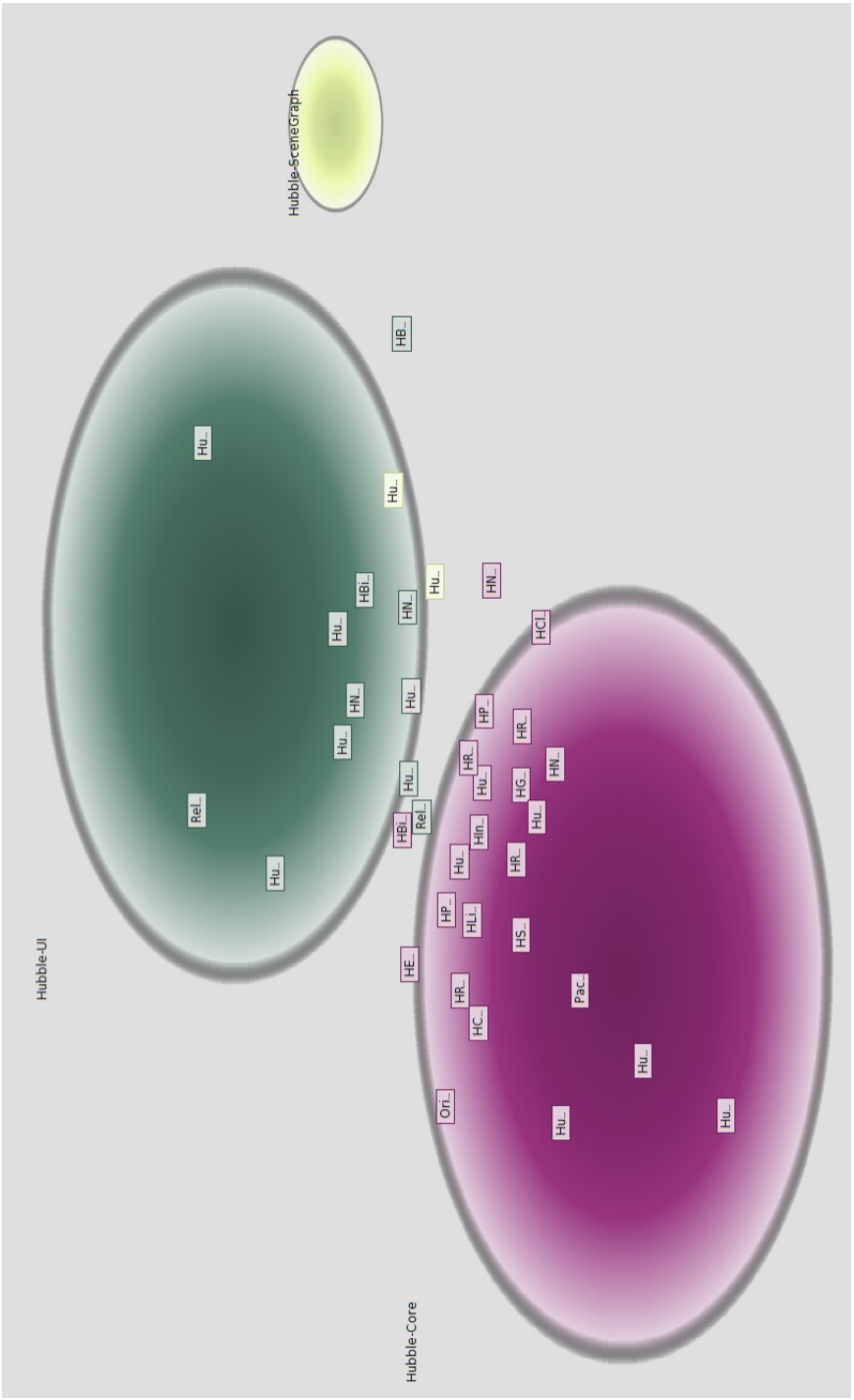

Figure 5.1. Hubble system visualization

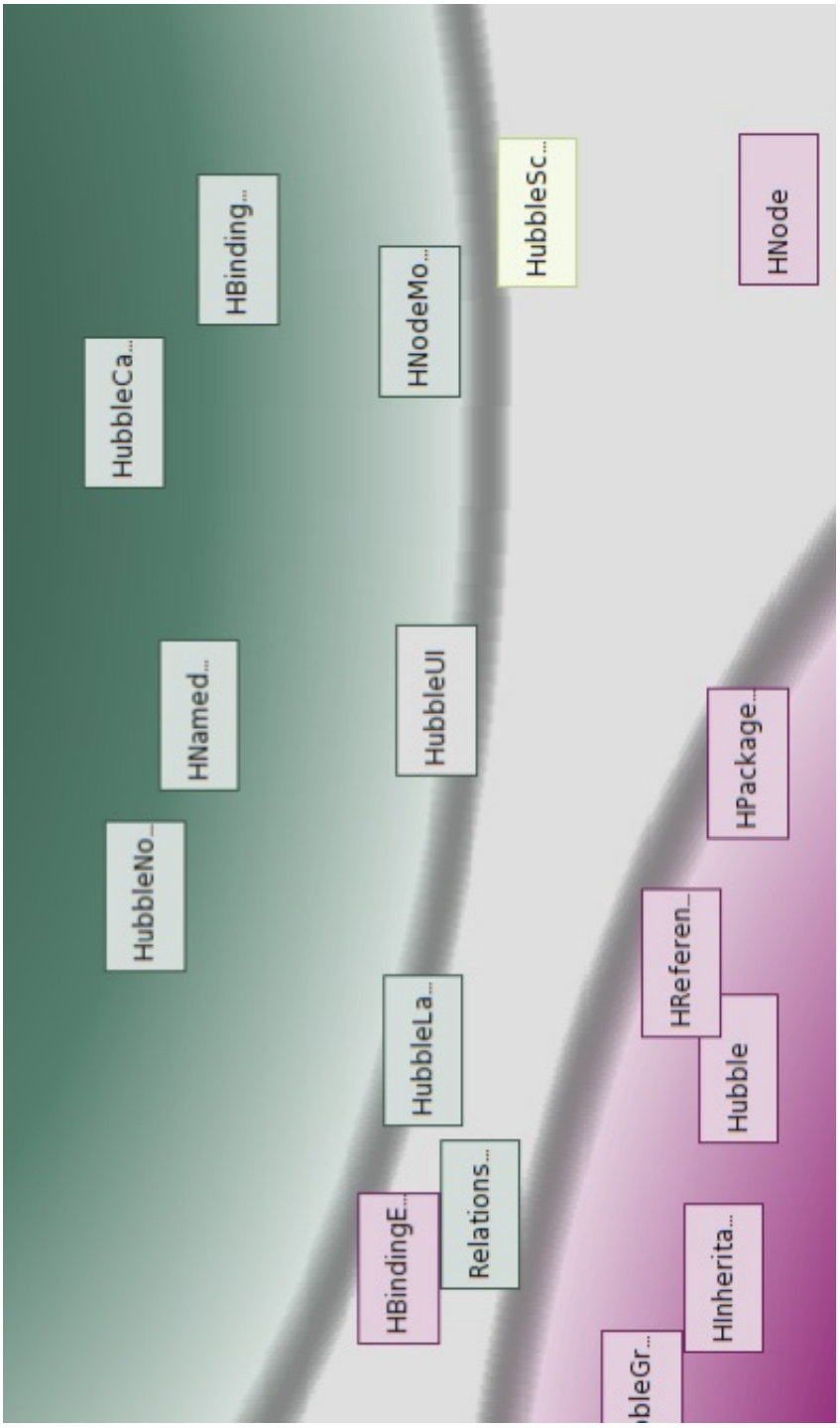

Figure 5.2. A zoomed view of the system's visualization.

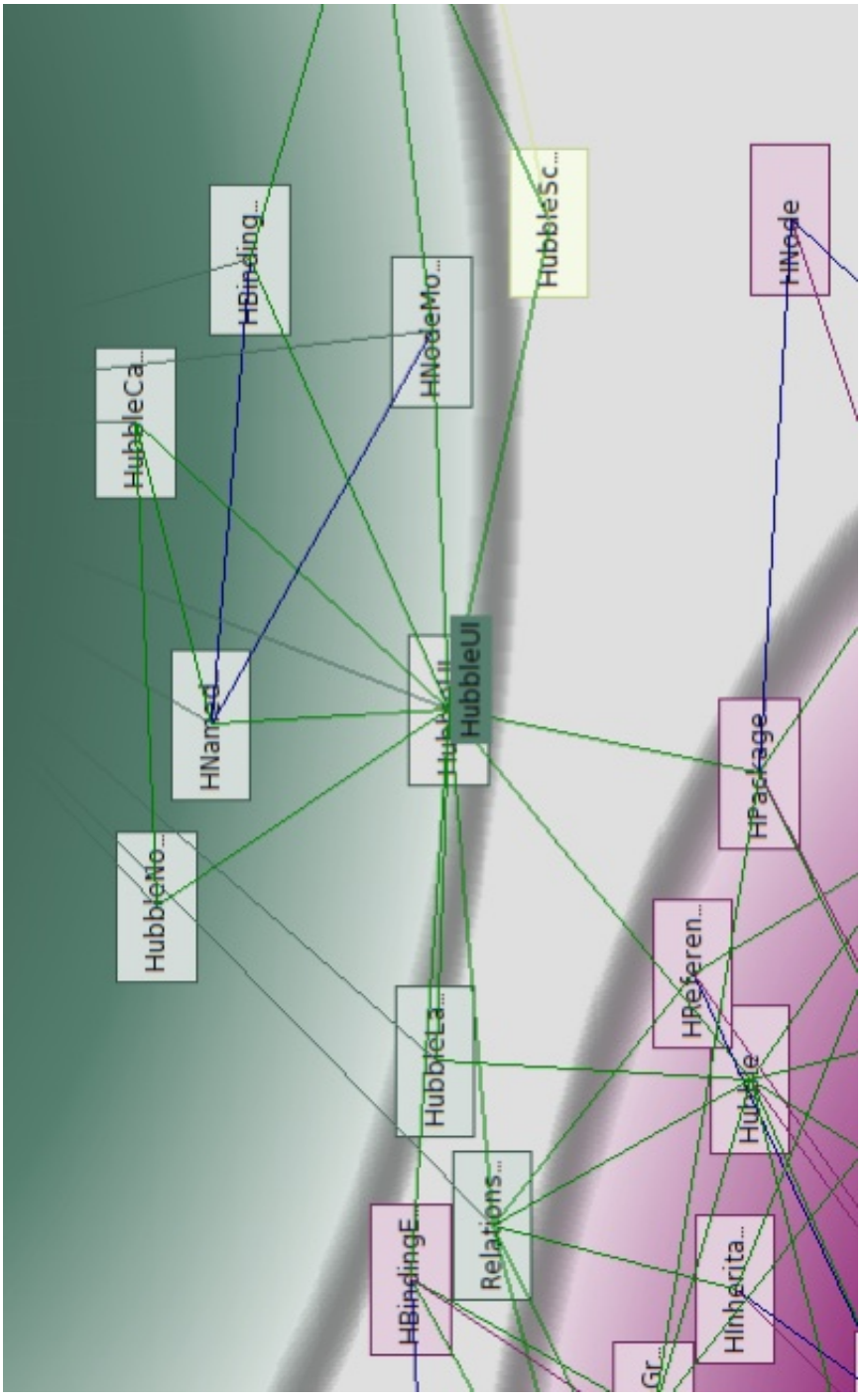

Figure 5.3. Hubble system visualization zoom, with edges displaying toggled on.

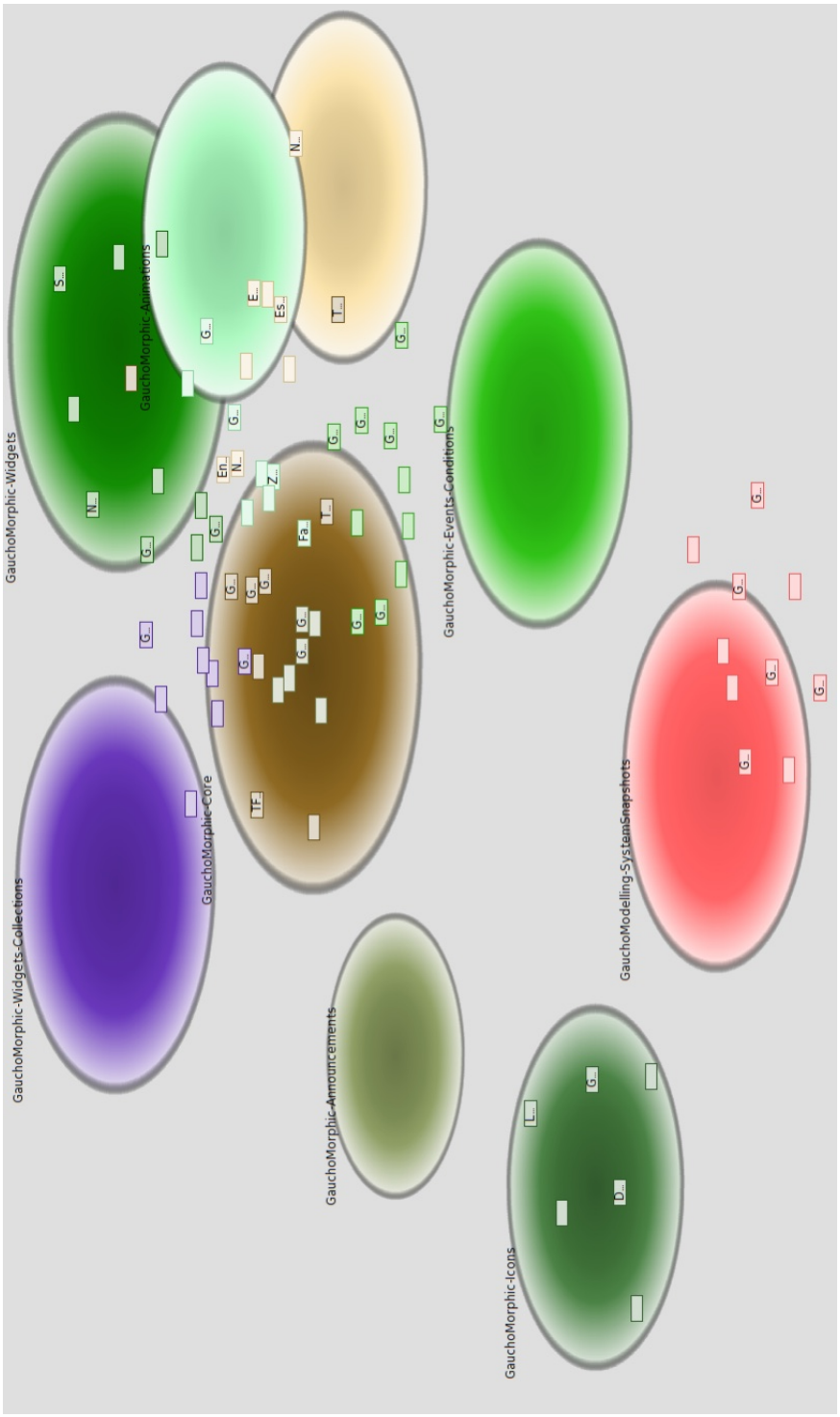

Figure 5.4. A general view of *Gaucho*.

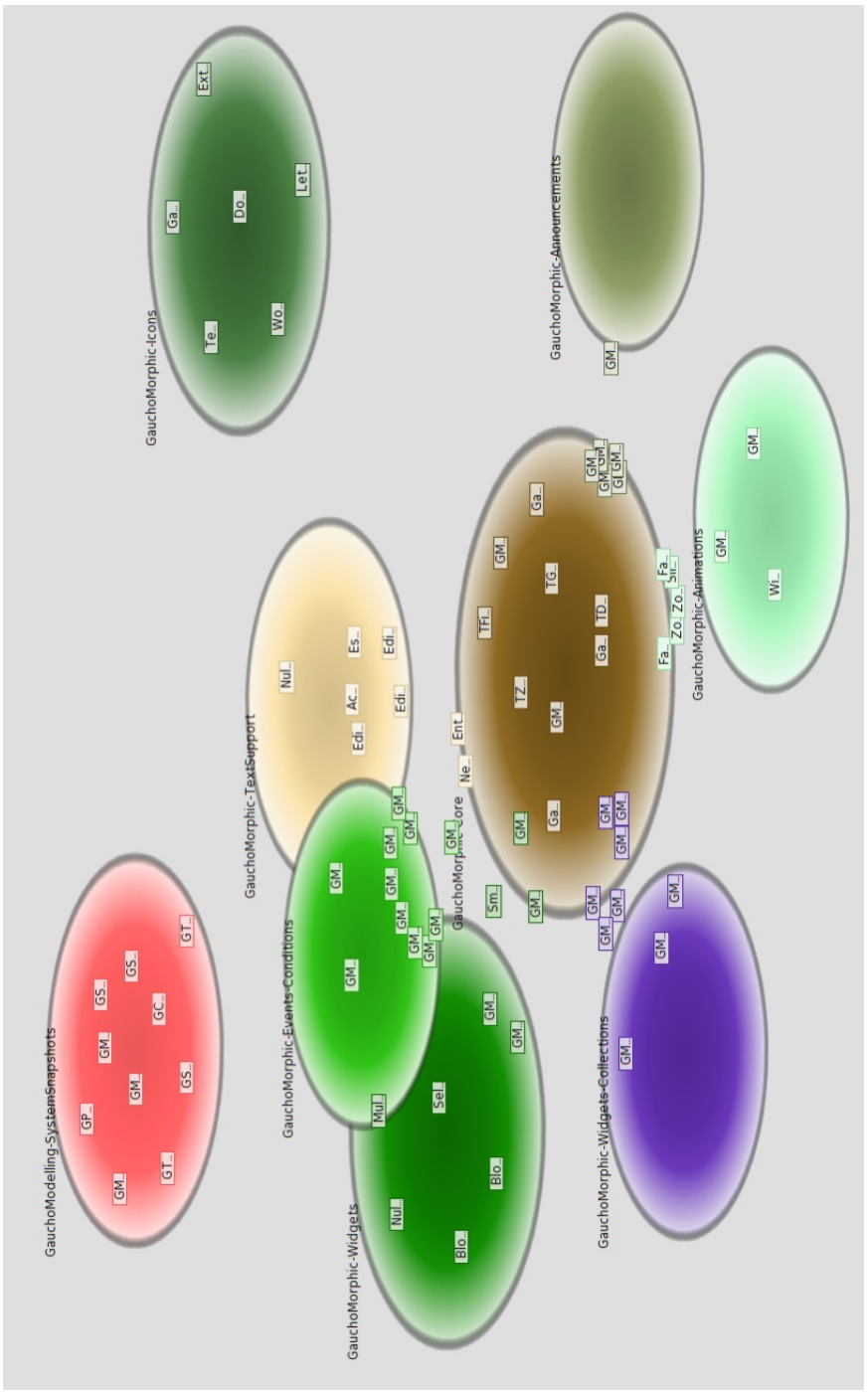

Figure 5.5. A general view of *Gaucho* with increased weight package containment relations.

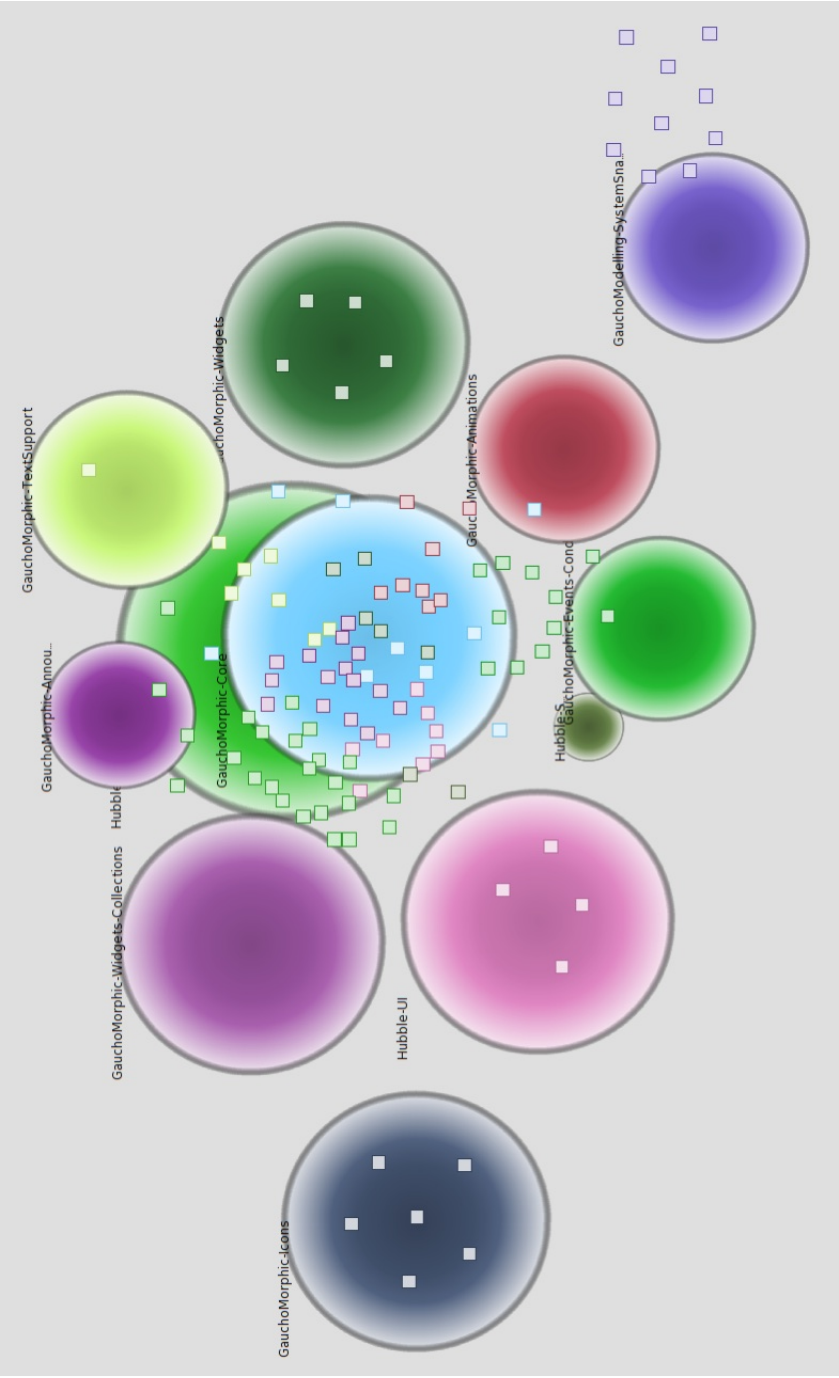

Figure 5.6. A general view of *Gaucho* and *Hubble*.

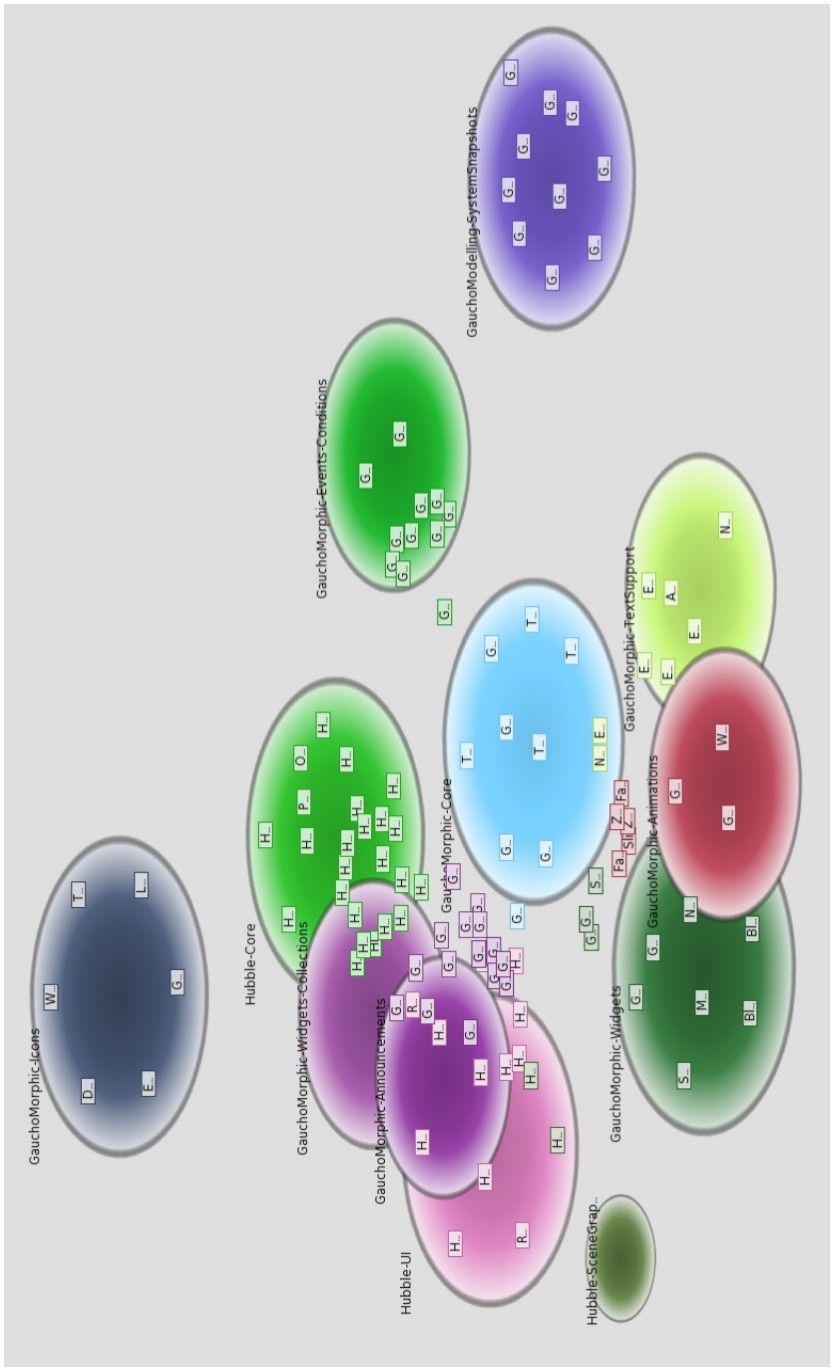

Figure 5.7. An alternative general view of *Gaucho* and *Hubble*.

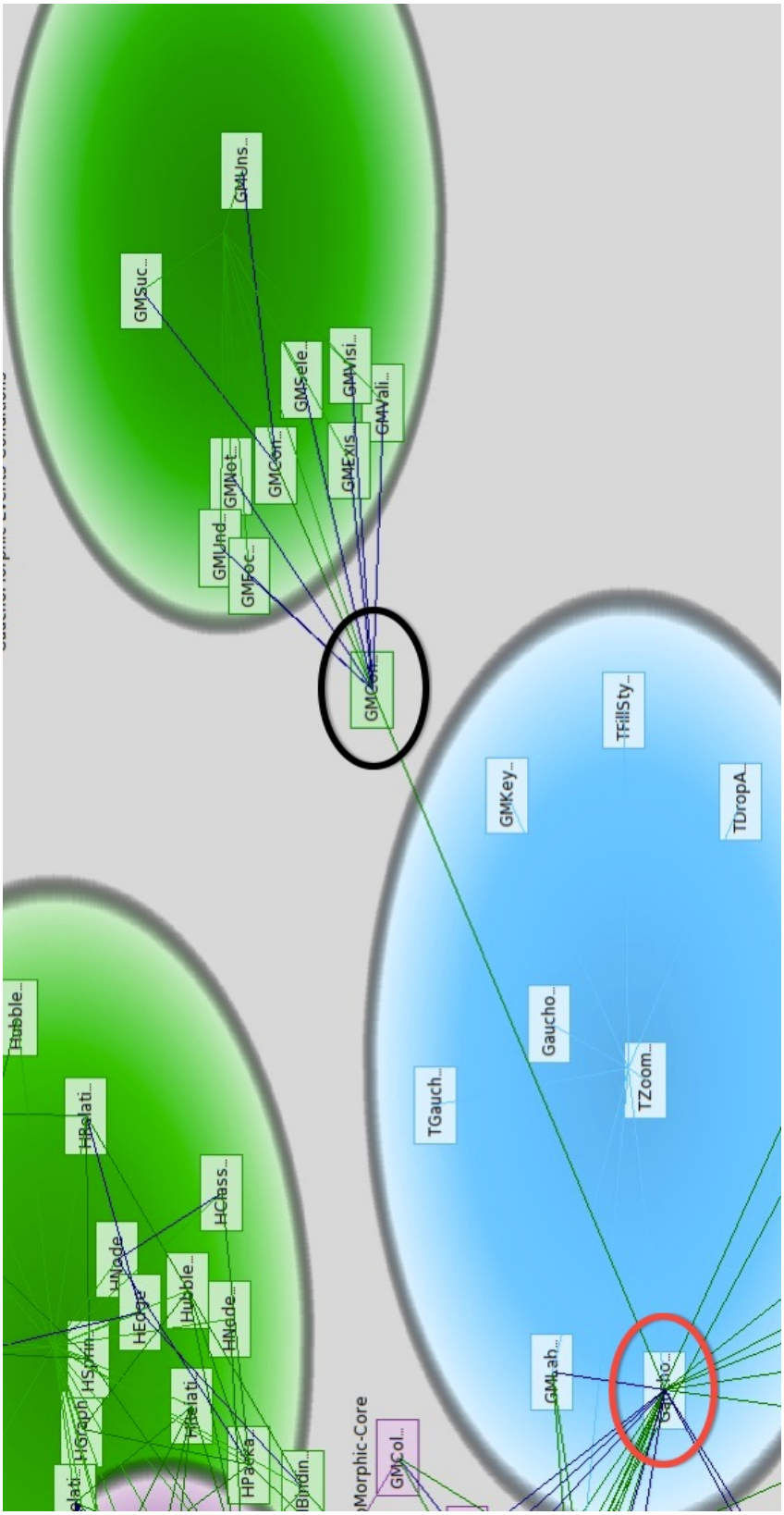

Figure 5.8. Analysis of relations between classes.

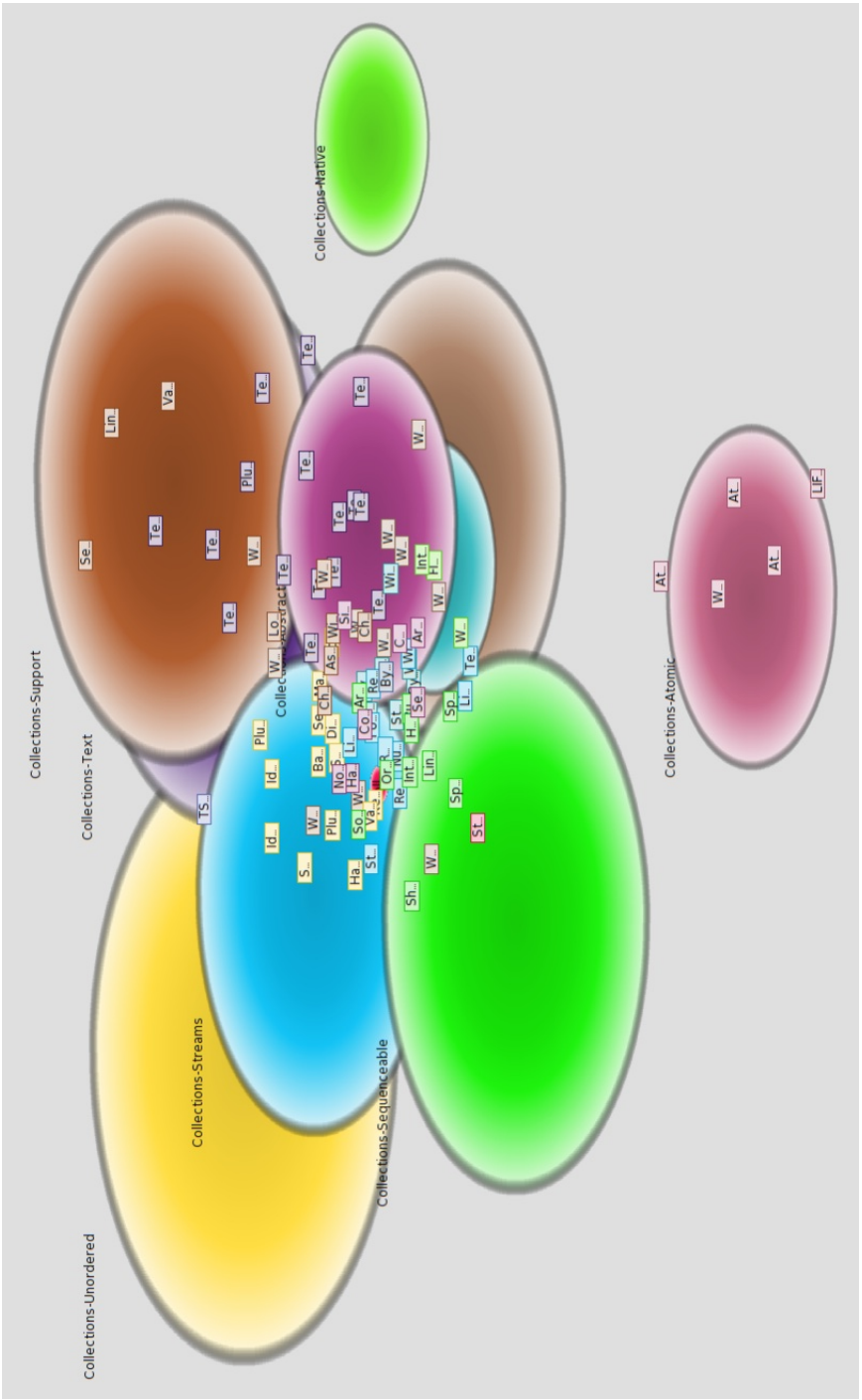

Figure 5.9. The general representation of Collections subsystem of Smalltalk.

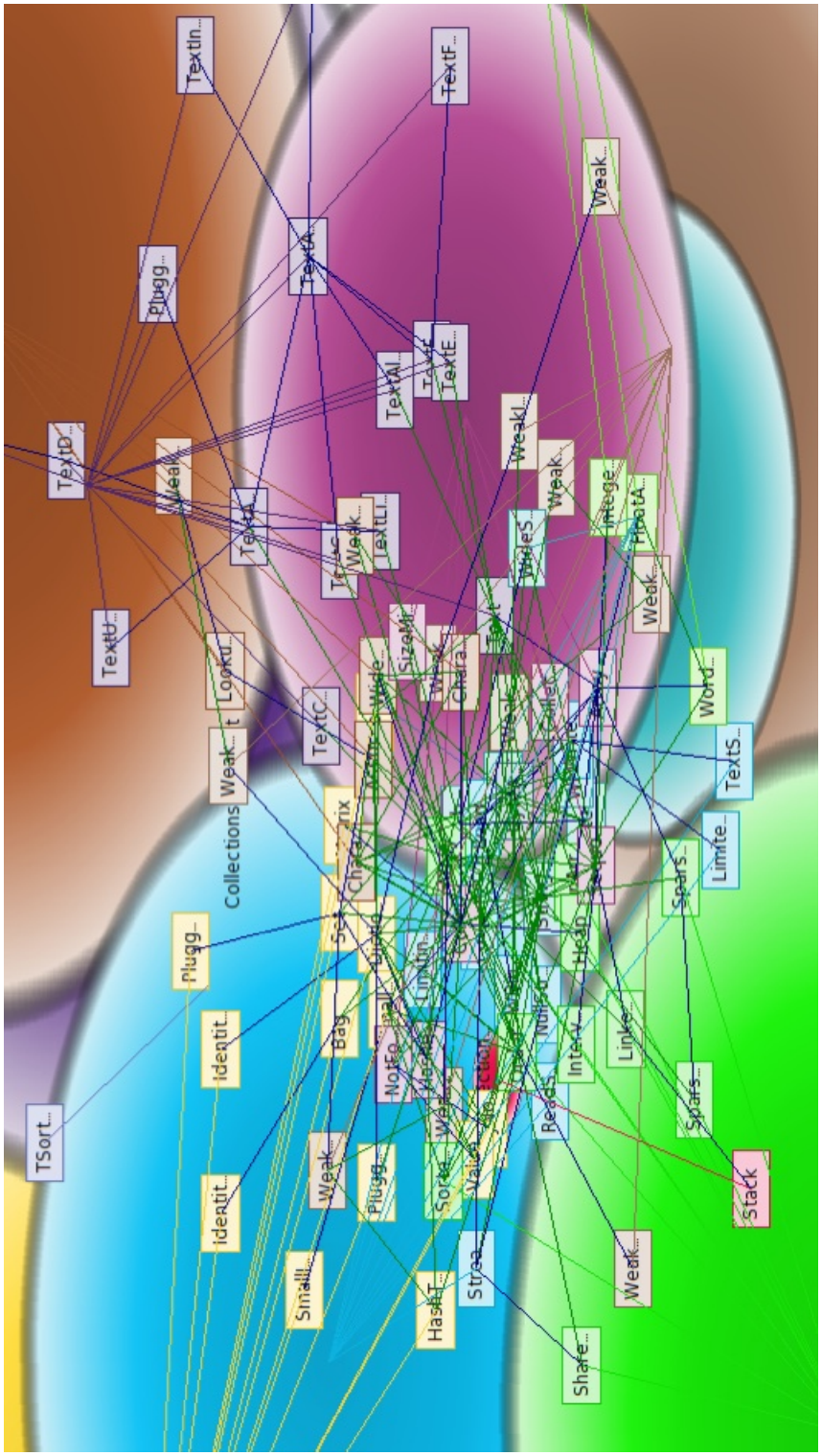

Figure 5.10. A view of the surprisingly high number of edges in the Collections subsystem of Smalltalk.

Chapter 6

Conclusions

This thesis presents our algorithm for visualizing a software system, whose development revolves around the set of goals we defined in Section 1.2.

In this closing chapter we analyze the results obtained with respect to those goals. We retrace the development steps done during the development and implementation of our work in Section 6.1; following it, in Section 6.2 we present a list of the possible directions which our thesis could be developed further into. Finally, to conclude this document, in Section 6.3 we state our impressions regarding the obtained results.

6.1 Summary

The goal of this work is to develop and implement an algorithm able to lay out the different part that compose a system in a way that many different relations between displayed elements are taken in account. To achieve it we created an algorithm which makes use of the space available in the canvas, exploiting the positioning of the elements of the software system being analyzed to suggest the degree of coupling between them.

To create the visualization of a software system our approach is divided into three phases:

1. Analyze the system to extract relevant entities that are both displayed *i.e.,* classes and packages, or used for laying out the visualization *i.e.,* reference, inheritance and package containment relations.

2. Aggregate the information extracted in the first phase to obtain the missing relations; once this is done combine all the information to build a graph, which is used to model the software system.

3. Layout the nodes of the graph using a customized force-directed algorithm. This process is performed in three steps: first each package is considered individually and its classes are positioned, then the package nodes are laid out according to relations between them, and finally all the classes nodes are taken in consideration and positioned according to the edges connecting them.

After completing the algorithm we developed *Hubble*, a visualization tool created to implement our approach and present its benefits. *Hubble* was developed to be used intuitively and to be learned quickly, providing developers a way to create a visualization easy to comprehend, which would help them creating a mental model of the system.

6.2 Future Works

Being this a very broad subject of study, there is a wide range of directions in which our proposed solution could be expanded. Here we define some of the possible path that could be taken for expanding this work and of the functionalities that could be added.

1. **Prevent nodes overlapping**. One problem of the force spring layout we have been using is that it is meant to work with points, and not with extended shapes; this results in the overlapping of some nodes under certain conditions, and thus in not letting users have a graphically pleasant view of the system. We managed to work around the problem by modifying the settings of the algorithm in order to try to prevent overlapping or at least reduce it to the minimum possible, however some overlapping could still be possible in some situations.

   We thought about two different strategies to solve this problem: the first would be implementing a corner stitching layout [Ous82] that would prevent the overlapping of nodes in the visualization phase. This is a simple solution to implement and would ensure good results as we would be sure that no overlapping would occur, it could however cause some minor movement that could cause two classes to move closer even if not related, or vice versa take apart two connected ones.

   The second strategy would be enhancing the algorithm to directly prevent overlap while running the algorithm. This approach is surely harder to realize and, depending on its implementation, could still not prevent all cases of overlapping. The advantage is that, having it directly implemented in the algorithm, the final result would be closer to the optimal solution.

2. **Speed up the algorithm**. Even though the algorithm for laying out the graph has been optimized while working on the thesis, there is still a lot of room for making it better. The first thing that could be done is working on the computation of the force and try to simplify them. Another way of making it faster would be to start cacheing results of some operations that are computed many times and re-structure the algorithm around this cache.

3. **Define a more intuitive way to browse the graph**. Despite being possible to browse the graph trough the zoom and move in the four directions (up, down, right and left), during interaction with the program it came out to be intuitive for the users to click on the window and drag the graph to display a different part of it. Implementing this would let users have a better experience while working with our program.

4. **Visualize class size**. As of now, all classes displayed look the same, except for the name and different color if they belong to different packages. One possible enhancement would be

displaying some properties of a class by showing classes with different sizes, or with thicker border according to the lines of code of the class, or depending on the number of method it has, to make easier for users to recognize which are the main classes of the system.

5. **Provide a detailed class tooltip**. The program already provides a little tooltip showing the name of classes displayed while mouse hovers them, however it could be a nice enhancement to also provide additional information on the class in some ways, i.e. clicking on the class with the right mouse button, giving users the opportunity to grab additional pieces of information, while still having in mind our goal of keeping the use of the application extremely simple and fast.

6. **Show system evolution**. As said in the previous part of the thesis, Hubble uses a graph as basis to display data; this is a strong starting point that could be expanded and used as a tracking tool for the system's evolution.

   Being already able to save snapshots of the systems at different time and to show them at a later time, users could create a history of the lifecycle of the system. Once this has been done it could be possible to integrate Hubble by making it able to create a video showing the evolution of the system based on these snapshots.

6.3 Last Thoughts

Building a very flexible tool that lets users define which data should be taken in account for creating systems' visualizations, and also moving elements after the visualizations have been generated, gives developers the freedom to analyze and explore in different ways various systems which can differ in many aspects. This provides them with more possibilities to approach the problem of creating a mental mode of the software system.

Our intuition of using space for displaying information, which can be grabbed by our brain without spending much effort in it, is a characteristic that prove itself very useful and must always be kept in consideration when developing any visualization tool.

We would also like to point out the importance of displaying multiple relations at once, as this allows a good understanding of the system. Our approach of utilizing space for showing them is a valid approach, however there could be others as well viable, if not better, and we hope that our work is be able to push research in this way.

Bibliography

[ABM+06] Erik Arisholm, Lionel C. Bri, Senior Member, Siw Elisabeth Hove, and Yvan Labiche, *The impact of uml documentation on software maintenance: An experimental evaluation*, IEEE Transactions on Software Engineering **32** (2006), 2006.

[BE96] Thomas Ball and Stephen G. Eick, *Software visualization in the large*, Computer **29** (1996), 33–43.

[Bec00] Kent Beck, *Extreme programming explained: embrace change*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CCI90] Elliot J. Chikofsky and James H. Cross II, *Reverse engineering and design recovery: A taxonomy*, IEEE Softw. **7** (1990), 13–17.

[Cor89] T. A. Corbi, *Program understanding: challenge for the 1990's*, IBM Syst. J. **28** (1989), 294–306.

[DDF+90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman, *Indexing by latent semantic analysis*, JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE **41** (1990), no. 6, 391–407.

[DT97] Eirik Tryggeseth Department and Eirik Tryggeseth, *Report from an experiment: Impact of documentation on maintenance*, Journal of Empirical Software Engineering **2** (1997), 201–207.

[Eic02] Holger Eichelberger, *Aesthetics of class diagrams*, In Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, IEEE, 2002, pp. 23–31.

[Erl00] Len Erlikh, *Leveraging legacy system dollars for e-business*, IT Professional **2** (2000), 17–23.

[Ern10] David Samuel Erni, *Codemap – improving the mental model of software developers through cartographic visualization*, 2010.

[GN98] Emden R. Gansner and Stephen C. North, *Improved force-directed layouts*, Proceedings of the 6th International Symposium on Graph Drawing (London, UK), GD '98, Springer-Verlag, 1998, pp. 364–373.

[HB95] Thomas Hofmann and Joachim Buhmann, *Multidimensional scaling and data clustering*, Advances in Neural Information Processing Systems 7, MIT Press, 1995, pp. 459–466.

[Hun96] Christopher Hundhausen, *A meta-study of software visualization effectiveness*, Journal of Visual Languages and Computing **13** (1996), 259–290.

[JTL00] Vin Silva Joshua Tenenbaum and John Langford, *A global geometric framework for nonlinear dimensionality reduction.*, 2000.

[KN02] Carsten Friedrich Keith Nesbitt, *Applying gestalt principles to animated visualizations of network data*, Proceedings of the Sixth International Conference on Information Visualisation (IVÕ02), 2002.

[Lee01] J. De Leeuw, *Multidimensional scaling*, Handbook of Statistics, North-Holland Publishing Company, 2001, pp. 285–316.

[LS81] Bennet P. Lientz and E. Burton Swanson, *Problems in application software maintenance*, Commun. ACM **24** (1981), 763–769.

[McK84] James R. McKee, *Maintenance as a function of design*, Proceedings of the July 9-12, 1984, national computer conference and exposition (New York, NY, USA), AFIPS '84, ACM, 1984, pp. 187–193.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gĭrba, *The story of moose: an agile reengineering environment*, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA), ESEC/FSE-13, ACM, 2005, pp. 1–10.

[NL05] Andreas Noack and Claus Lewerentz, *A space of layout styles for hierarchical graph models of software systems.*, Association for Computing Machinery, Inc, 2005, pp. 155 – 164.

[Ous82] John K. Ousterhout, *Corner stitching: a data structuring technique for vlsi layout tools*, Tech. Report UCB/CSD-83-114, EECS Department, University of California, Berkeley, Dec 1982.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker, *A taxonomy of software visualization*, Proceedings of the 25th Hawaii International Conference on System Sciences (Kauai, HI, USA), vol. 2, January 1992, pp. 597–606.

[Rei05] S.P. Reiss, *The paradox of software visualization*, Visualizing Software for Understanding and Analysis, International Workshop on **0** (2005), 19.

[Rig11] Francesco Rigotti, *Visualizing software systems and developers activity*, 2011.

[Sea05] Dabo Sun and et al., *On evaluating the layout of uml class diagrams for program comprehension*, 2005.

[TS03] Huang S. Tilley S., *A qualitative assessment of the efficacy of uml diagrams as a form of graphical documentation in aiding program understanding.*, Proceedings of the 21st Annual International Conference on Documentation, 2003, pp. 184–191.

[War04] Colin Ware, *Information visualization: Perception for design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Wet10] Richard Wettel, *Software systems as cities*, Ph.D. thesis, Università della Svizzera Italiana, 2010.

[WS06] Kenny Wong and Dabo Sun, *On evaluating the layout of uml diagrams for program comprehension.*, Software Qual J **14** (2006), 233 – 259.

[XW08] Fenglian Xu and Alex Wood, *Reverse engineering uml class and sequence diagrams from java code with ibm rational software architect*, jun 2008.

[ZSG79] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon, *Principles of software engineering and design*, Prentice Hall Professional Technical Reference, 1979.

[ZZ03] Hongyuan Zha Zha and Zhenyue Zhang, *Isometric embedding and continuum isomap*, In Proceedings of the Twentieth International Conference on Machine Learning, 2003, pp. 864–871.