FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Interactive Class-Diagram Generation and Abstraction

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

BSc. Gàbor Miklos Hernàdi
Registration Number 0725876

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Vienna, 23rd August, 2015

Gàbor Miklos Hernàdi	Franz Puntigam

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interaktive Generierung und Abstraktion von Klassendiagrammen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

BSc. Gàbor Miklos Hernàdi
Matrikelnummer 0725876

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 23. August 2015

Gàbor Miklos Hernàdi	Franz Puntigam

Erklärung zur Verfassung der Arbeit

BSc. Gàbor Miklos Hernàdi
Alserbachstrasse 6 / 12a

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. August 2015

Gàbor Miklos Hernàdi

Danksagung

Ich möchte mich bei meinen Eltern und Brüdern bedanken, dass sie mich sowohl bei dieser Arbeit als auch in den vorangegangenen Studien mich mit Rat und Tat nach Kräften unterstützt haben. Auch möchte ich ein herzliches Dankeschön an Verena Himmelbauer und Rudolf Mildner richten, einerseits für die hilfreichen Rückmeldungen bezüglich dieser Arbeit, als auch dafür dass sie sich als wertvolle Freunde und Studiengefährten erwiesen haben. Ein weiterer Dank gebührt Herrn Professor Puntigam für seine geduldige und stets hilfreife Betreuung bei dieser Arbeit.

Abstract

Developing software systems depend on a good overview. Creating such an overview manually is a lengthy task and the resulting overview generated by tools often contains too much information, causing more confusion than clarity. In order to gain a more suitable view on the system, unnecessary information could be hidden using abstraction techniques. However, a static decision on which data the user desires to be visible or not is not trivial for a machine as it depends on the user's current focus. In this thesis, we will discuss techniques of calculating a layout which takes the relation of classes into account, namely the force directed layout. The proposed prototype modifies the positioning algorithm in order to prevent node-overlapping. Furthermore, this thesis introduces structured elements, which are recursive, visual and logical containers for the basic class nodes. The prototype is interactive and has capabilities to increase visual clarity. These capabilities include collapsing, expanding, highlighting and temporarily removing a set of nodes in or from the current graph. On any structural change the positioning algorithm relayouts the graph, providing a layout as similar as possible to the old one.

Kurzfassung

Bei der Entwicklung von stetig wachsenden Software-Systemen ist Überblick sehr wichtig. Die manuelle Erstellung solcher Visualisierungen kann sehr viel Zeit in Anspruch nehmen, wobei eine automatisierte Erstellung oft in einem überfüllten Bild endet. Unnötige Informationen können ausgeblendet werden, allerdings ist eine ständig gültige Abstraktion nahezu unmöglich, da die Wichtigkeit der einzelnen Elemente vom aktuellen Fokus der entwickelnden Person abhängt. Im Rahmen dieser Arbeit werden Methoden zur Positionierung der Knoten beschrieben, welche die benachbarte Knoten beisammen hält. Hierbei handelt es sich um das sogenannte Force Directed Layout. Der vorgestellte Prototyp modifiziert diesen Algorithmus, sodass keine Knotenüberlappungen vorkommen können. Weiters werden Structured Elements eingeführt, welche rekursive, visuelle und logische Container für die einzelnen Datenknoten. Der Prototyp ist interaktiv und steigert die Klarheit der Übersicht durch folgende Möglichkeiten: Ein- & Ausklappen, hervorherben und zeitweises Entfernen von einer Knotenmenge. Bei jeder strukturellen Änderung berechnet der Positionierungsalgorithmus ein Layout, das dem Alten möglichst ähnlich ist, um den Benutzer bei der Wiedererkennung der neuen Übersicht zu unterstützen.

Contents

CHAPTER 1

Introduction

Whenever software becomes large and complex, creating an overview of the system gets inevitable. Such overviews are used to explain the software, especially to people who were not involved from the very beginning in the project. Developers often draw pictures as humans are in general better at understanding pictures than extracting information from numeric values [1, 2, 3]. The images are usually sketches or, if detail is required, often based on diagrams specified in the Unified Modeling Language (UML) [4]. The UML 2 standard contains a large set of visualization techniques, which are mainly split in the categories "Structure Diagrams" and "Behaviour Diagrams". This thesis mainly focuses on overviews of the former group representing classes and their relationships.

Visualizations of this kind are used to explain the basic idea of the software, without the need of going through every single line of code. However, nowadays software projects tend to have up to several hundred classes already in small or medium sized projects. Thus, an overview containing every tiny piece of information about the project is simply overwhelmingly complex for any viewer.

1.1 Motivation and Problem Statement

An overview representing a given project can be created in several ways. However, starting from scratch there are two possibilities, namely either creating the overview by hand or using a tool that generates one. A human might construct a better overview than an algorithm, but at the cost of time. Even when assuming the developer is very familiar with the whole system, the time needed for building the overview may easily exceed hours, while an algorithm usually requires in the worst case just a few minutes.

With this significant difference in mind, many tools capable of generating such overviews were developed. The quality of the resulting overviews of those tools might not be as good as if made by hand, but they are calculated in a fraction of time. The lack of quality can often be compensated with interactive features. These features may provide different views of the system, exceeding the capabilities of static overviews.

However, tool generated overviews of classes and their relations often get cluttered by the sheer amount of classes. Visualizations created manually are also likely enhanced by not containing less important classes. The decision if a class is important is in most cases no trivial task for an algorithm. An improvement of such algorithms can be achieved by prefiltering the set of classes for the generator, but this is still very time-consuming. However, the importance might also change with the focus of the viewer.

To visualize a given code base, a developer can either create an overview manually, which is very time-consuming, or he / she can use an automated tool, which often generates a cluttered result. The aim of this thesis is to provide the developer with a tool, that automatically generates a basic overview and is further adaptable to support the developer's current focus.

The prototype proposed in this thesis is able to interactively visualize an overview over given object oriented code. The required data to generate the visualization is gathered through reverse engineering, where the classes and relations in between them are used to calculate a force directed layout. Doing so, the layouting algorithm positions each node, so that no overlapping occurs, yet related or communicating nodes are still close to each other. Additionally, this thesis introduces the concept of structured elements, which are recursive, logical, and visual containers for encapsulating nodes. Finally, the tool provides analytic methods for further analysis. These methods allow the user to define more or less important classes so that they can be highlighted or hidden.

1.2 Thesis Structure

Chapter 2: State of the Art contains topics relevant to this thesis. The chapter gives an overview about three topics: Software visualization, abstraction of class diagrams and analysis methods.

Chapter 3: Theoretical Background discusses some issues with reverse engineering, explains the used data model and goes into detail of the core concept of the prototype, namely the force directed layout.

Chapter 4: Methodology mentions the overall approach of developing the prototype. This includes used tools and frameworks, programming language and a summary of the development phases.

Chapter 5: Implementation explains the structure of the prototype and how techniques and concepts from the previous chapter were used. The techniques were extended with overlapping-prevention, encapsulating classes into containers representing sub-systems and a mechanism to define sets of classes.

Chapter 6: Evaluation shows the output generated by the prototype, includes performance tests and discusses the limitations of the current implementation.

Chapter 7: Conclusion provides, as the name suggests, the conclusion over this thesis.

CHAPTER 2

State of the Art

The task of visualizing and analyzing a given code base can be divided into three major sections. The first one is software visualization, giving the user a visual representation about their system. The second topic deals with the problem of the sheer amount of visible elements, as too many of them might clutter the overview. Last, but not least, we will discuss techniques which allow the user to use the rendered visualization for further analysis in order to understand the system or some of its properties.

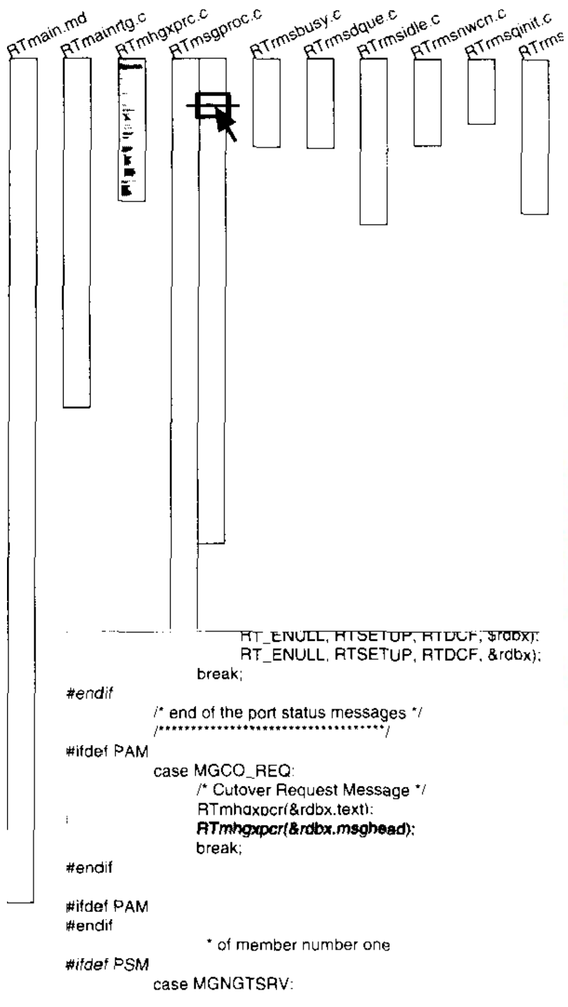
2.1 Software Visualization

There are fundamentally different approaches to software visualizations. Some of them try to give the user an overview about the source code itself, others have additional information they can rely on as object orientation or hierarchical structures of the code. Relation-centered approaches contain well known examples like graphs and *UML* diagrams. There are also some interesting techniques extending previous visualizations with metrics extracted from the code base.

In the following, we will discuss different visualizations across the previously mentioned topics. The structure and chosen techniques in this section are inspired by “Visualization of the static aspects of software: A survey” [5].

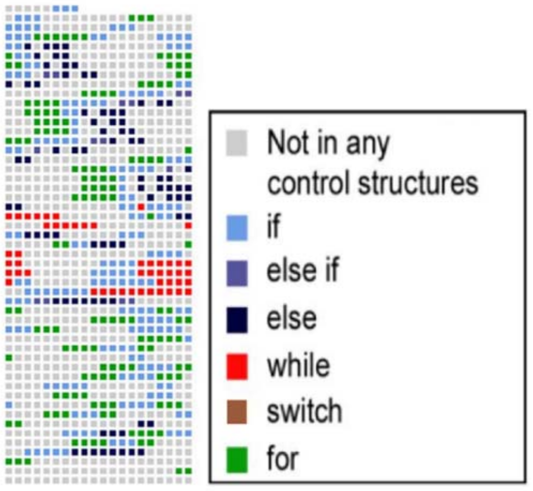
2.1.1 Source Code

Although the text editor itself could be interpreted as a visualization of the software, representing the actual source code has been researched for a long time. First implementing approaches providing an overview included an intuitive “zoom” effect, allowing the viewer to literally see more of their code or color various types of structural elements differently. The latter approach is nowadays better known as syntax highlighting.



```
                    RT_ENULL, RTSETUP, RTDCF, $rdbx);
                    RT_ENULL, RTSETUP, RTDCF, &rdbx);
                break;
#endif
            /* end of the port status messages */
            /***********************************/
#ifdef PAM
            case MGCO_REQ:
                /* Cutover Request Message */
                RTmhgxpcr(&rdbx.text);
                RTmhgxpcr(&rdbx.msghead);
                break;
#endif

#ifdef PAM
#endif
                 * of member number one
#ifdef PSM
            case MGNGTSRV:
```

(a)



(b)

Figure 2.1: (a) *SeeSoft* showing multiple source-files with a zoomed out effect (top), and a magnifying view of the hovered section (bottom). Image from [6]. (b) Alternative view of *SeeSoft*. Every line of code is represented as a colored block. Image from [5].

SeeSoft

SeeSoft [6] is one of the few visualizations which represent the source code itself. Figure 2.1a [1] shows the organization of files and code lines. Every file is represented as a vertical bar. Every row of a file is mapped to its bar as a line with just a few pixels as height and a width proportional to the length of the corresponding line of code. Using this proportional width, the indentation is preserved. This technique from the early 1990's can nowadays be described as a zoom-out technique. The source code gets shrunk

[1]The original application was colored and in higher resolution such that every vertical bar was filled with horizontal lines as the third bar indicates

to the height of one pixel. The amount of lines of the source code file corresponds with the height of the bar itself. For example, the fourth file in Figure 2.1a had too many lines of code, creating a bar which reaches the bottom of the visualization. However, *SeeSoft* is able to create multiple bars for each file to display the entire content of the given file. Multiple files can be displayed side by side, all of them vertically starting at the same baseline. This way the developers have a rough overview over every, or at least over the most important files in their system.

Each line can be colored differently. The usage of colors is configurable depending on whether statistical or structural information shall be displayed. Statistical information is obtained from the version control system, including information about the author, the last modification date, how often it is executed, version and feature number, and so on [6]. Structural information can be mapped based on flow control statements such as conditional jumps (if, else, switch, case), loops (for, while, do-while) or other units of code (comment blocks or functions).

Eick et al. describe in their paper that the user is able to directly manipulate the source code by manipulating the display, although this claim is mainly based on “direct manipulation techniques, in particular updating the screen in real-time in response to mouse actions” [6]. However, this approach is based on the intuitiveness of the representation, as the developers immediately recognize their own code by structure because it is the same as if they were looking at the code from the distance. With an enabled “read” button a box is shown at the bottom of the screen while the mouse moving along a bar representing the zoomed out version of the source code, as Figure 2.1a illustrates. The source code which was abstracted inside the content of the box is then visible as readable code in an additional “magnifying” window.

On the very left of the source code bars a key shows the color code. The default code when displaying the last modification date is based on the colors of the rainbow, red for the newest and blue for the oldest modifications. Interactivity is achieved by hovering the mouse over a specific color in the color code bar. The lines which were modified on the selected date are still visible whereas all other modifications lose their color. By clicking-techniques this “on over” effect can be changed into a permanent one, allowing the user to use the mouse for further exploration.

Very short lines of code tend to be hardly visible or not visible at all. An additional button deals with this problem by ignoring the length of the line. This feature makes the visualizer drawing all rows the same size, using the full width of the given bar.

An alternative and more compact visual representation of *SeeSoft* is shown in Figure 2.1b. One line of code is mapped to one colored block. Figure 2.1b shows a block structure displaying different structural elements of each line. Although this way the information about the length of lines is lost, but the whole source file can be displayed in a more space filling format. Colored units (blocks instead of narrow lines) are now easier to detect and to distinguish from their neighbors. Using this representation and a color code based on modification times as mentioned earlier allows the user to detect single lined modifications in one source file much easier.

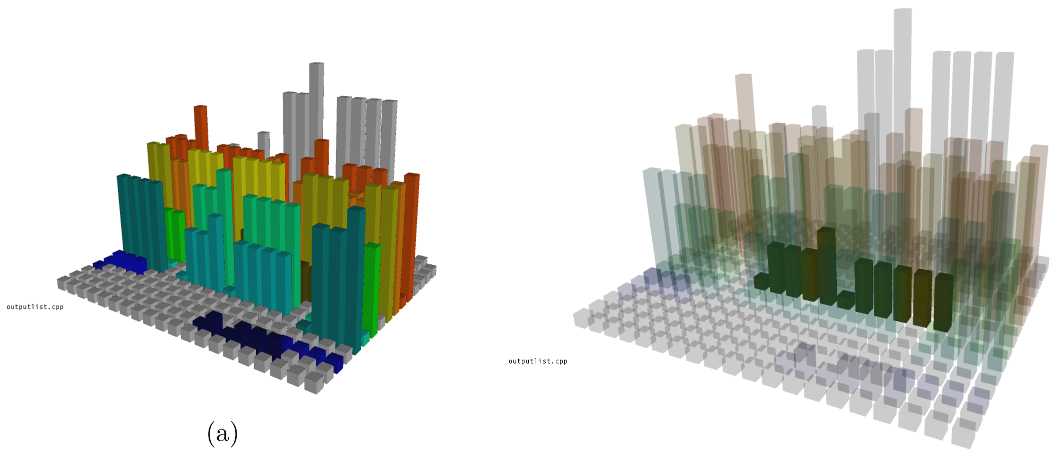


Figure 2.2: (a) *sv3D* rendering of Doxygen's outputlist.cpp source file. One cylinder per line of code. Colors represent the surrounding function. Height is based on hit count of the line. (b) *sv3D* filtering one function. Other lines (cylinders) at 85% transparency. Images from [7].

sv3D

The *source viewer 3D* (*sv3D* [8]) enhances the approach of *SeeSoft* to the third dimension. The additional dimension allows the visualization to provide additional information without cluttering the view. Figure 2.2a shows the same kind of block view as Figure 2.1b does with a color code for the structural information (conditional jumps and loops as the key in Figure 2.1b shows). The third dimension shows the nesting level of each line. Additionally to the use of the third dimension, the tool can rotate, zoom and move the visualization as the user wishes. This zoom functionality allows the user to focus on his / her current need.

The authors of *sv3D* mention that *SeeSoft*-like tools (as well as *sv3D* itself) can be used for various tasks, such as: Fault localization [9], visualization of execution traces [10], source code browsing [11], impact analysis, evolution, complexity and slicing [12]. *sv3D*'s ability to use the third dimension allows it to surpass other approaches, like *Tarantula* [9] where brightness as additional source of information confused the user instead of providing useful insights.

Due to the use of the third dimension, the navigation has to be intuitive. When the user gets literally lost in space, he / she has to use view-reset functionalities and start all over again. *sv3D* supports various 3D navigation techniques, like zooming and panning at variable speed, as well as rotation, scaling and translation of each represented source file [8]. Moreover, the tool supports a number of filtering methods which makes uninteresting units transparent as shown in Figure 2.2. The user can configure transparency, color, shape and texture for the used filter.

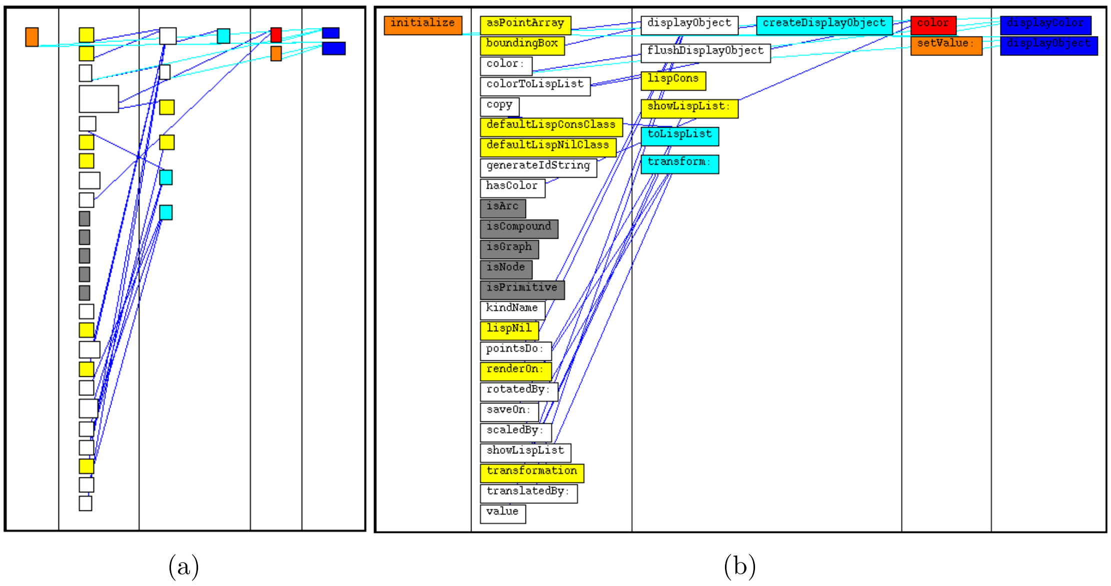


Figure 2.3: *Class blueprint* view of *JunOpenGL3DGraphAbstract*. (a) Mapping metrics for width and height of methods and attributes (b) Displaying names. Images from [13].

2.1.2 Class-Based

There are only few approaches that visualize the structure of a class itself. One of them is *class blueprint* [14], whose goal is a "semantically augmented visualization of the internal structure of a class, which displays an enriched call-graph with a semantics-based layout" [13]. This approach splits the methods and attributes of a class into five layers, namely initialization, external interface, internal implementation, accessor and attribute layer. The same order of layers can be seen from left to right in Figures 2.3a and 2.3b.

Class blueprint provides an automatic assignment of methods and attributes to the mentioned layers. A method or attribute will be assigned to the given layer depending on the rule set shown in Table 2.1.

Both the nodes and the edges use color codes for more detailed information. There are three types of edges. Both invocation of a method and invocation of an accessor are colored blue. Direct accesses to an attribute are cyan. The nodes are split into eight types, for example abstract methods as cyan, delegating methods as yellow, attribute access is colored red, attributes are constantly blue and so on.

The tool can visualize the same view in different ways. Figure 2.3 shows two examples of the very same source code. Figure 2.3a does not contain any method or attribute names, but instead the height and width of each node are calculated based on a given metric. In Figure 2.3b the dimensions are derived from the length of the corresponding method and attribute name.

Class blueprint's layouting works by assigning the nodes to a layer, depending on the rule set mentioned earlier. In the first three layers a horizontal tree layout algorithm

Layer name	Assignment rules
Initialization	• The method contains the substring "init". • The method is a constructor. Methods like Java's static initializer should also be assigned to this layer, but are not taken into account by *class blueprint* as their meta model does not cover such structures anyway.
External Interface	• The method is invoked by a method from the initialization layer. • The method has a public or protected modifier (if the language supports such modifiers). • The method is not invoked by other methods within the same class. This layer excludes accessor methods as those are collected in a separate layer described later.
Internal Implementation	• The method has a private modifier (if the language supports such modifiers). • The method is invoked by at least one method of the same class.
Accessor	• The method is a getter or setter. Such methods have the only purpose of returning the value of a private variable (getter) or assign the given parameter to a specific private variable (setter).
Attribute	• All attributes of the current class.

Table 2.1: *Class blueprint*'s rule set assigning methods and attributes to the corresponding layer [13].

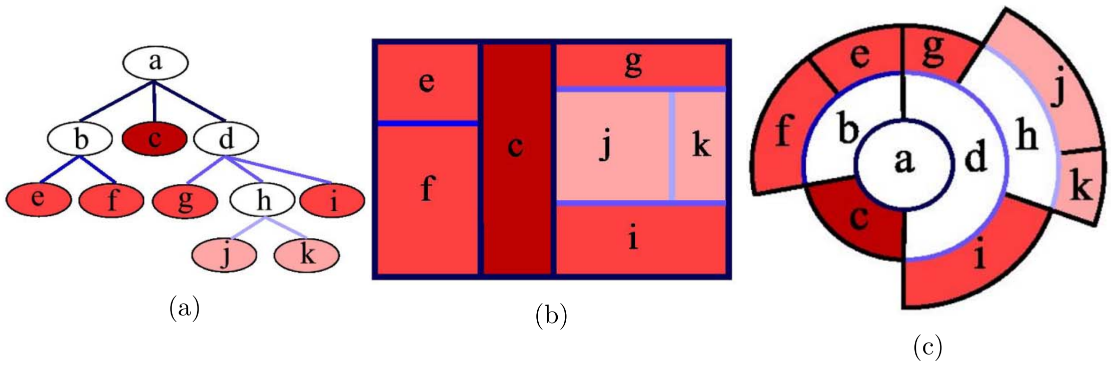


Figure 2.4: Different visualizations of the same system. (a) Tree view (b) Treemap (c) Sunburst. Images form [5]

for the exact positioning is used. That means, if a method invokes another method, the invoked method will be positioned right of the invoker and the two methods are connected with an edge with a corresponding color. A vertical line layout is used on the inside the last two layers. Further, if a method calls another method from the same layer, the call is visualized as a linked list, as visible in the third, the internal implementation layer of Figure 2.3.

This visualization technique allows the viewer to receive information about the class in a very short time. For example, nodes in the internal implementation layer which do not have any edges are considered as dead or unreachable code [2]. Classes used for data storage can also be seen very easily as those may have some nodes in the first (initialization) layer, none or just a few in the next two (external interface and internal implementation) layers, but many nodes in the last two layers (accessor and attribute). More examples of visual patterns can be found in the mentioned paper [14].

2.1.3 Hierarchial Structures

Figure 2.4a shows the *Tree view* of a hierarchical system. The red nodes are leaves, each depth level with a different shade, while the white nodes are internal nodes. The colors are consistent in the three figures of Figure 2.4. The *Tree view* is a very general yet intuitive approach to visualize a given hierarchy. However, the drawbacks are the limited space for displaying information about each node as well as the inefficient space usage. Unbalanced or large hierarchies will lead to overviews with much wasted space.

There are many subtly different implementations of the *Tree View*, which most likely differ in the positioning of a node's children. One method is to draw the tree as traversing it with a breadth-first approach. A naive implementation may position every child side by side, directly next to each other. The problem is, that if the grandchildren are not taken into account, the next row of children gets unrelated to the current level. The connecting edges will be hard to follow, thus the tree will loose of its clarity. As this

[2]We do not take into account special invocations like Java's reflection framework

issue keeps getting worse with each iteration, other approaches try to calculate the needed space from bottom up. This way the parent of a set of children can be positioned horizontally centered above them. However, both of these approaches tend to consume unnecessarily much space. Alternative approaches were developed which try to position the next children towards the horizontal center if possible in the given context [15].

Treemap

Johnson and Shneiderman introduced the *Treemap* as a “space-filling approach to the visualization of hierarchical information structures” [16]. Figure 2.4b shows the same system as its left neighbor. This technique makes use of all available space. The recursive algorithm alternately cuts the given space into pieces vertically and horizontally. The amount of pieces depends on the number of children of the current node. This way, the internal nodes, colored white in Figure 2.4a, are not visible in the *Treemap*, but the path of each leaf is still preserved.

Beside an efficient space filling algorithm, the original approach also considers interactivity. The user is able to change display properties as color mappings as well as boundaries themselves. However, the algorithm still depends on a given weight of each node to be able to calculate the final size of the bounding box. This weight should be seen as a measure of importance or degree of interest [16].

Originally *Treemap* was designed for directory-like tree visualizations. That means, there is a strict distinction between internal nodes (e.g. directories) and leaves (e.g. files). Thus, the lack of visibility of the internal nodes is acceptable when the user is only interested in the actual files. However, as an alternative to Figure 2.4b, a nested version of *Treemaps* exist, where the internal nodes have also a visible (thus selectable) border [16]. In the non-nested version the missing information about the internal nodes is still accessible as a visualization of properties when clicking on a specific leaf node. The additional property view also showed the leaf’s unique path containing a reference to every traversed internal node.

Although each node has a set of visual properties, such as color (hue, saturation, brightness), texture, shape (within the corresponding bounding box), border, blinking, and so on, color is considered as the most important [16]. However, the paper also states, that the number of different codings has to be limited due to human perception. This limitation increases the necessity of interactivity.

The authors claim that their introduced algorithm is able to traverse any tree structure and draws the corresponding *Treemap* in O(n) time[3]. The procedure is done in two major steps: first, the node draws itself within the given bounding box. The dimension of the bounding box is based on its properties (shape, color, weight, et cetera). In the second step the node splits its available space into portions for each of its children. The split also includes the alternation between splitting vertically or horizontally. Afterwards, the node recursively calls the first step for each child. The core Think C code is available in the authors paper [16].

[3]In assumption that all properties (weights, names, and so on) are previously computed or assigned

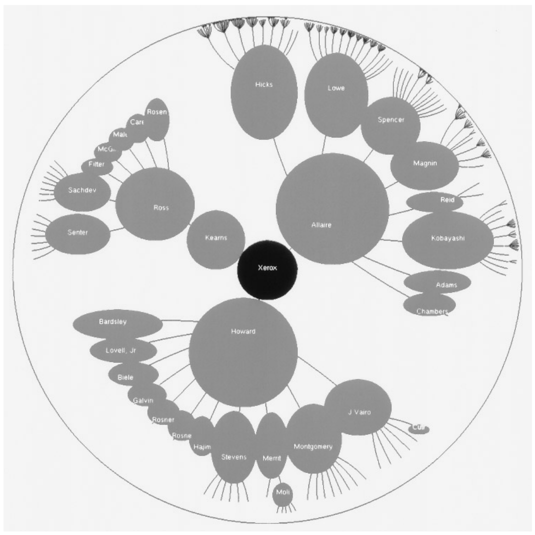


Figure 2.5: Hyperbolic Browser [17]

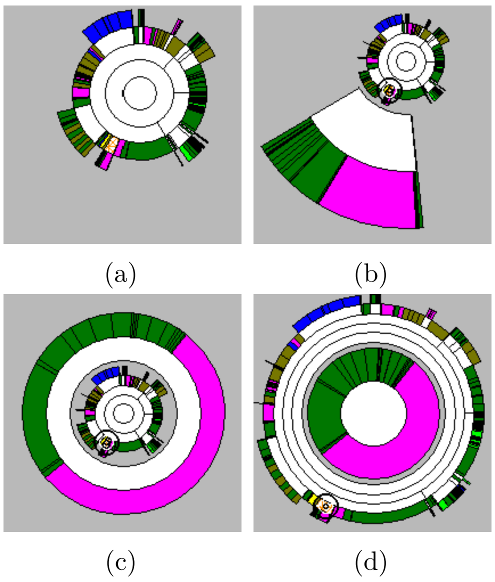


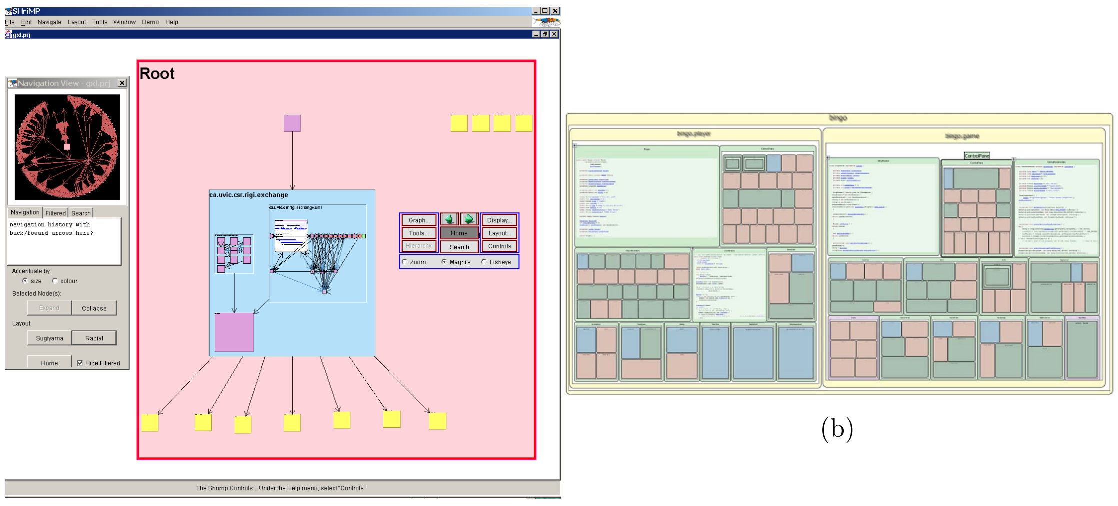
Figure 2.6: *Sunburst*'s zooming options [18]

Sunburst

Figure 2.4c shows the *Sunburst* [18] visualization of our example hierarchy. *Sunburst* is a radial approach of the *treemap*. Instead of splitting the available space in rectangular boxes, *Sunburst* starts in the center with a circle as the root node. Every child is limited to the arc of its parent. The children of the root node (nodes b, c and d) result in a full circle, whereas the children of b (e and f) are limited to the arc of b. As every node is visible in *Sunburst*, the overall nesting is more intuitive, thus the understanding of this visualization is easier to learn [5].

Sunburst is also inspired by the *Hyperbolic Browser* [17], which is shown in Figure 2.5. The currently selected node is always in the center of the visualization. When the user wants to change the focus in the browser, the next active node moves to the center, pushing the currently centered node outwards and pulling the selected node's neighbors towards the center. As the name of this approach suggest, this visualization uses hyperbolic geometry, which means that the farther objects (nodes) are from the center, the smaller (e.g. unimportant) they get.

Figure 2.6a shows the initial state of a given system. The user selects a dense area on the bottom left (about 7 o'clock). One improvement takes the selected area, duplicates and enlarges it in an intuitive animation, resulting in Figure 2.6b. Figure 2.6c takes the approach similar to the logic of expanding root's children. The (now selected) children are creating a full circle, but with the overall view of the system in the center, like the root node would be. The third improvement is a kind of inverted version of the previous one. Instead of building the selected area as a new outer circle, the involved nodes are shown in the very center of the visualization, as visible in Figure 2.6d.


SHriMP
File Edit Navigate Layout Tools Window Demo Help
Root
ca.uvic.csr.rigi.exchange
Navigation Filtered Search
navigation history with back/foward arrows here?
Accentuate by: size colour
Selected Node(s):
Expand Collapse
Layout:
Sugiyama Radial
Home Hide Filtered
Graph... Tools... Hierarchy Home Search Display... Layout... Controls
Zoom Magnify Fisheye
The Shrimp Controls: Under the Help menu, select "Controls"
bingo
bingo.player
bingo.game

(a) (b)

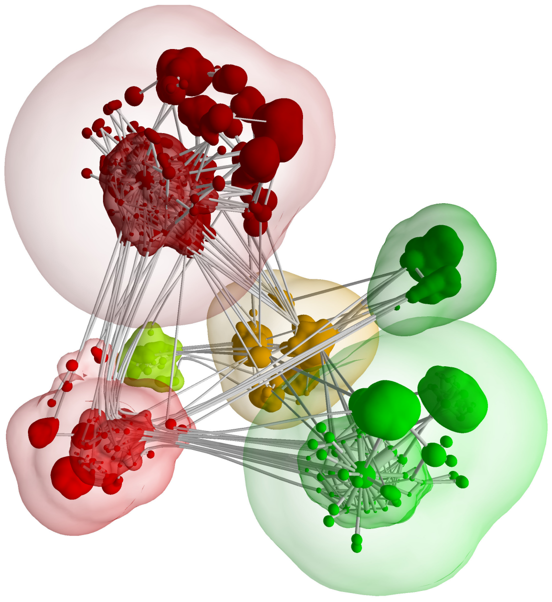
Figure 2.7: (a) *SHriMP* visualizing Java code with multigraph techniques. Image from [19]. (b) *SHriMP* combining *Treemap* with source code view. Image from [5].

SHriMP

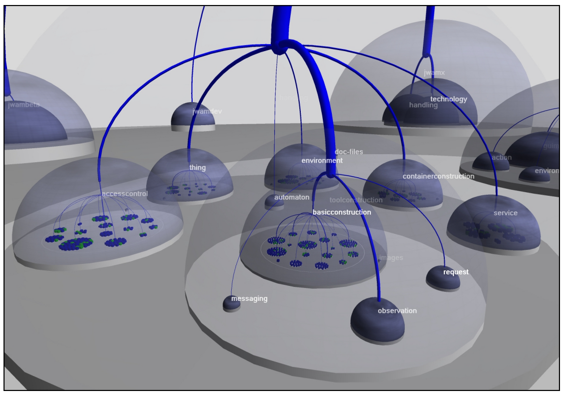
The *Simple Hierarchical Multi-Perspective* (*SHriMP* [19]) visualization technique was developed to provide multiple perspectives of the given information space while exploring it. *SHriMP* combines several visualization techniques, such as a *Treemap* layout, as well as higher level (multi-) graph layouts as fisheye-views [20] and pan and zoom effects. One of the key features of *SHriMP* is that these visualization techniques are fully interchangeable. Figure 2.7a shows an example where browsing Java code is split into the graphical view with multiple granularity, such as source code view (white box in the center) and a hyperbolic view of the overall system on the left. Figure 2.7b shows the tool with a combined view of the hierarchical presentation of classes as a *Treemap* with the more detailed source code view of specific classes. Each color representing either packages (yellow), classes (light green), methods (green and blue) or attributes (red).

The navigation of *SHriMP* through the system is achieved with several techniques, including simple geometric zooming (scaling of all elements without context awareness), semantic zooming (non-leaf will open, revealing more details about its content) and a "context+detail" approach (fisheye-view [21]). Although *SHriMP* was originally designed for visualizing software systems, it has been improved to be usable for navigating through a variety of knowledge domains. Besides software visualization the tool is also suited for visualizing knowledge engineering and flow diagrams.

Experiments [22] have shown that users of *SHriMP* found the overall layout logical and were especially content with the ability of switching between the various views of the tool. However, the experiments also showed some disadvantages of *SHriMP*, namely the lack of search functionalities, as well as the lack of parsing C / C++ macros (users of *SHriMP* looking for variables or constants had to investigate all header files manually).



(a)



(b)

Figure 2.8: (a) Clustered graph. Image from [23]. (b) Hierarchical net. Image from [24].

2.1.4 Relationship Centered

As the name of this section suggests, the main focus is not primarily the data containing objects, but the relation and their types. There are different types of relations. Speaking of object-oriented code, well known types are simple method calls, inheritance, implementing interfaces, direct field accesses and so on. An object could theoretically have arbitrarily many relations, whereas such relations could also be divided into uni- or bi-directional relations. We will now discuss several approaches how visualizations can help the user to focus on the relation of the objects, components or sub-systems.

Graph

Graphs in general are well suited for representing relations, mapping the items as nodes and the relations as edges. However, one of the biggest problems when visualizing graphs occurs when the number of nodes or the density increases. If the amount of nodes is too high, i.e. the graph is too sparse, not all of the nodes can be displayed, whereas if the graph is too dense it gets cluttered making further investigations almost impossible.

On the other hand, utilizing the properties of spare graphs can improve the navigation and orientation significantly. For example, edges can be given a direction (indicating a one way relation), weight (representing the relatedness of the connected neighbors), colors, thickness and so on. Nodes themselves can vary in size, color (including transparency), shape, texture, content, et cetera. Also the bare position of nodes can play a role in understanding a system. Two nodes may be closer together to each other if they are in

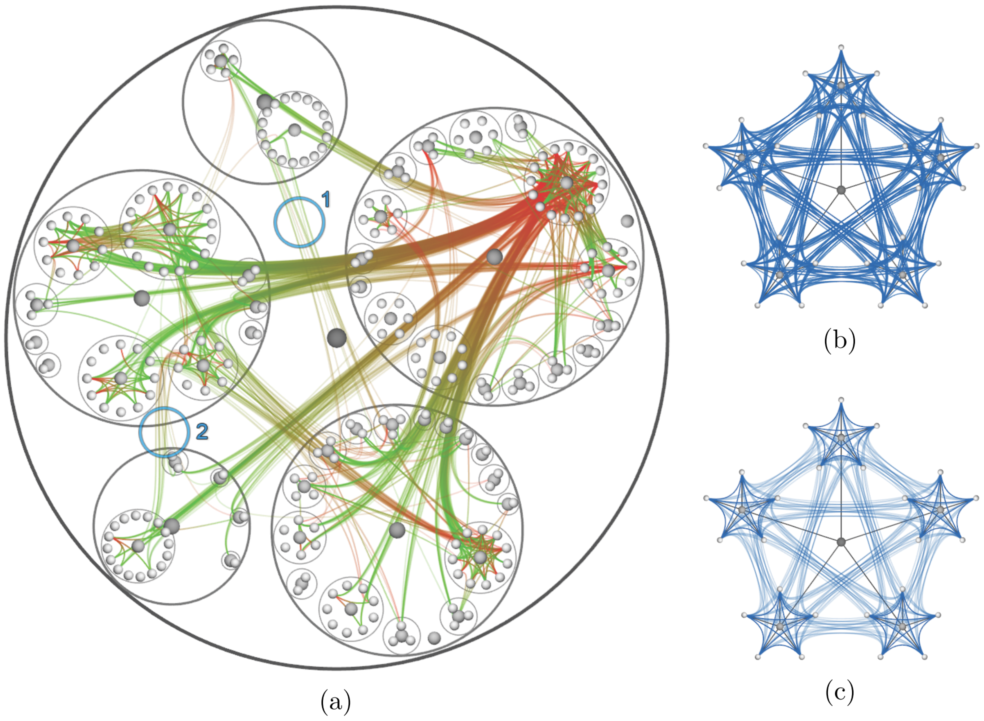


Figure 2.9: (a) Edge bundling in a clustered graph (b) Edge bundling without alpha blending (c) Edge bundling with alpha blending. Images from [25].

the same package, even if they do not share a connecting edge. Using these principles, clustered graphs [23] try to encapsulate logically correlated nodes into super nodes, as shown in Figure 2.8a. The hierarchical structure of a system, internally represented as a tree, which is a special case of a graph, can also be visualized intuitively as a three dimensional net [24], as Figure 2.8b shows.

Hierarchical edge bundling [25] is a technique which bends the edges, bundling them if their targets are locally close. Figure 2.9a shows an example of a clustered graph with edge bundling. The two blue circles mark the relations between two super-nodes which are very spare, indicating low correlation between the super-nodes. Additionally the gradient indicates the caller (green) and the callee (red), resulting in a *Call Graph* (which we will discuss in Section 2.3.1).

Figure 2.9b shows another example of edge bundling. The overall structure of the graph is clearly visible, but as the edges overlap some nodes and edges, the finer structure of each cluster is less clear. Figure 2.9c changes the opacity of the edges [25], while keeping the bundling effect from Figure 2.9b. The shorter the edge, the less transparent it is. The idea is, that “since short curves only occupy a small amount of screen space, they tend to become obscured by long curves” [25]. Additionally the rendering order of the edges is also influenced by the length of the edge. With this technique both, the overall as well as the inner structure of the system are clearly visible.

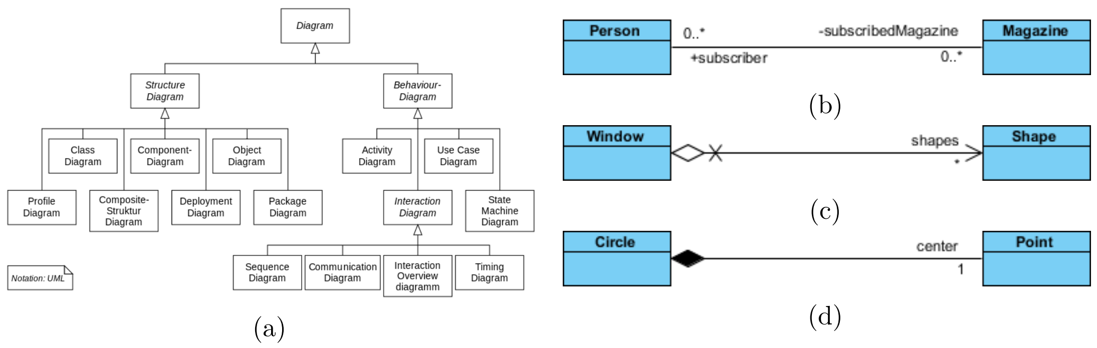


Figure 2.10: (a) *UML* diagram types represented in *UML* class diagram notation. Image from [W1]. (b) Examples of the relation types of class diagrams: association (top), aggregation (middle) and composition (bottom)

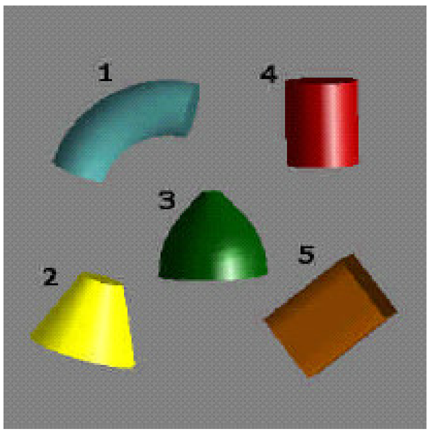
UML

The *Unified Modeling Language* (*UML*) [4] might be the best known graph-based visualization of software systems. *UML* is a collection of diagrams and representation techniques with the aim of visualizing relations and / or the state of a system or a well defined subsystem. Figure 2.10a shows the different types of diagrams, whereas the figure itself is in *UML* class diagram notation. As we can see, *UML* is divided into two kinds of diagrams, namely structural and behavior diagrams. Both of them are again extended by diagram types with more specific purposes.

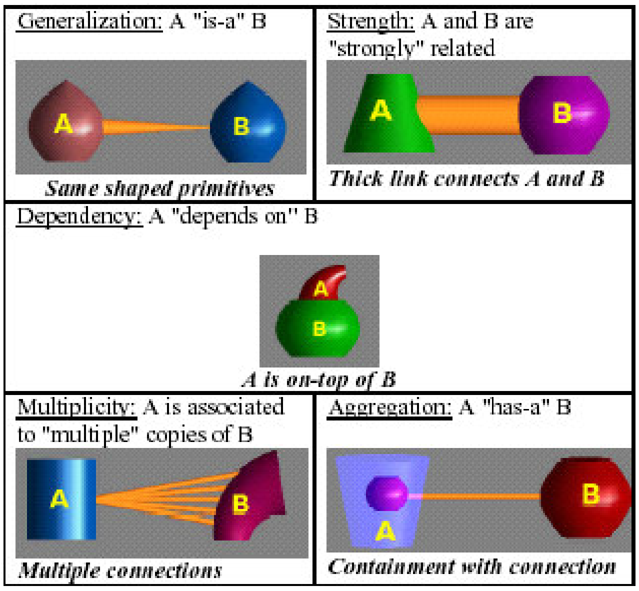
A class diagram is a graph where the nodes are represented as rectangles containing the name of the class and optionally its attributes and operations. The edges are relations between the classes and are distinguished in inheritance, generalization, dependency as well as in association, aggregation and composition.

Figure 2.10b shows an association connecting the classes `Person` and `Magazine`. Every association can be extended with additional information, such as navigation direction, multiplicity and names (for variables and the association itself). Navigation types are indicated with an arrow (navigable), no arrow (not specified), or a cross (not navigable). Multiplicity, i.e. the number of addressed instances, which supports beside positive numbers and zero an asterisk for arbitrarily many instances. For example, Figure 2.10b shows a bi-directional association, where both classes have zero to many references of each other. `Person`'s variable name is "subscribedMagazine" and `Magazine` knows the related `Persons` as "subscriber". The $+$ and $-$ signs prefixing the reference names indicate the reference's visibility, in our case public and private respectively. Others would be # (protected), / (derived, can be combined with others) or $\sim$ (package).

Figure 2.10c shows an aggregation, which extends the association. This type of relation is also often called as "part of" relation. Aggregation means that both classes can exist without each other, but `Window` gets the important displaying data from the `Shape` instances. However, to decide from a given code, whether a relation is an association or an aggregation is quite hard as it strongly depends on the semantics.



(a)


Generalization: A "is-a" B
Same shaped primitives
Strength: A and B are "strongly" related
Thick link connects A and B
Dependency: A "depends on" B
A is on-top of B
Multiplicity: A is associated to "multiple" copies of B
Multiple connections
Aggregation: A "has-a" B
Containment with connection
A
B

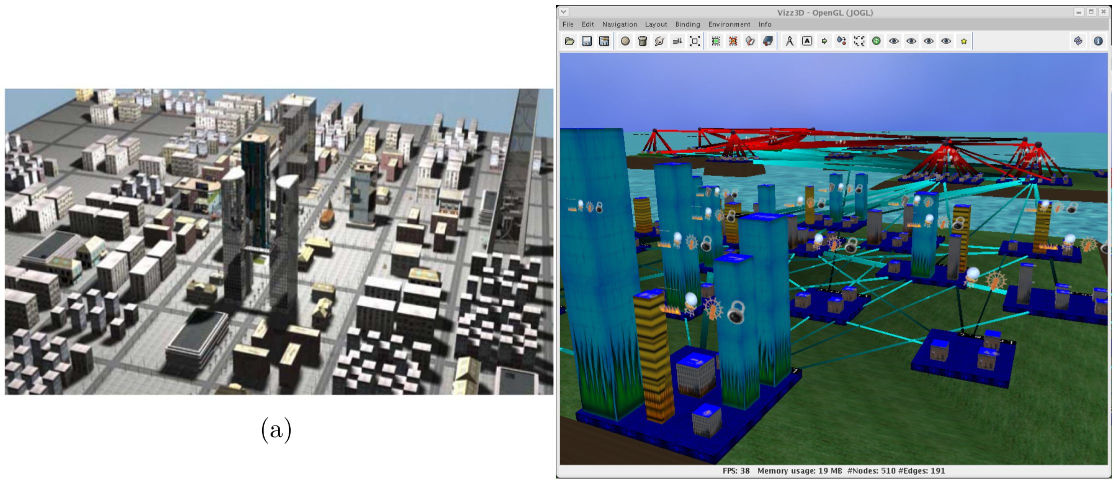
(b)

Figure 2.11: (a) Primitive *Geon* objects from Biederman's theory [26] (b) Visualizations of the different connection types of *UML*. Images from [27].

Another extension of association is the composition. The main difference between composition and aggregation is the lifecycle of the composed instances. Figure 2.10d shows the relation of the classes `Circle` and `Point`, where `Circle` uses a `Point` instance for its center. That means, that whenever an instance of `Circle` is deleted, the "center" instance is also deleted. Thus, the lifecycle of this `Point` instance is determined by the `Circle`. In the other relation types, both instances can be created and destroyed independently from each other.

Originally *UML* was supposed to be a 2D visualization. However, some works [28, 29, 30] tried to render the *UML* in 3D, facing this way the problems of edge crossings and available space. For example, Casey et al. [27] built a tool which replaces the elements of an *UML* class diagrams with *Geons*, which are 3D objects with easily distinguishable shapes and colors, as shown in Figure 2.11a. Figure 2.11b shows, that the tool implemented different visualizations for the different connection types. Some features are that objects sharing the same shape have an "is-a" relation, multiplicity are easily recognizable, and thickness of the connecting edge indicates the relatedness.

Gutwenger et al. [31] researched the aesthetic of *UML* class diagrams in special and created the `GoVisual` [31] library. Their work is based on the results of Purchase et al. [32], which states that the most important aspects of aesthetics are minimizing crossings and bends, orthogonality, horizontal labels and joined inheritance arcs. *GoVisual* implemented and extended this criteria list with "Uniform direction within each class hierarchy" and "No nesting of one class hierarchy within another" [31].



FPS: 38 Memory usage: 19 MB #Nodes: 510 #Edges: 191

(a)

(b)

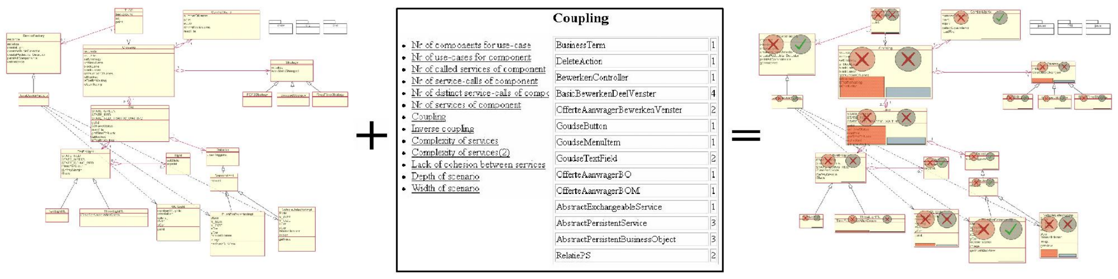
Figure 2.12: (a) *City Metaphor* viewing the software system as a single city. Image from [34]. (b) *Cities Metaphor* visualizes multiple connected cities. Image from [35].

Real World Metaphors

Two dimensional visualizations often have a problem with limited space. One of the easiest ways to face this issue is to expand the space to the third dimension. However, this extension comes with some problems. The most common challenge is the user interface for navigation, which is important to avoid disorientation. The problem is that most users have input devices providing 2D input which has to be mapped to a 3D environment [33].

The *Real World Metaphor* aims to create an environment the user is already familiar with. This environment could be a single city (*City Metaphor* [34, 36, 37, 38]), multiple cities (*Cities Metaphor* [35]), a solar system [39, 40] or any other well known type of hierarchy. The most important aspects of *Real World Metaphors* are that the metaphor has to be familiar to the user and it should contain a clear granularity. The granularity is used for the hierarchical structure of the software. In Figure 2.12a we see one large city. The blocks of buildings representing the classes, the buildings themselves the methods. Cities may use a city per class and a building per method or a city per package and the city metaphor as a finer granularity as shown in Figure 2.12b. Solar systems consist of stars (classes), planets (methods) and maybe moons for anything more detailed.

As mentioned before, an intuitive navigation has an important role in 3D visualizations. A common technique is to set up a 3D environment, containing a light source, a camera and 3D objects. The navigational input of the user is passed to the camera. However, there are different approaches. Some of them give the user unlimited control, allowing navigation through the whole 3D space [24, 41, 42]. Others restrict the movement in order to reduce disorientation [43]. Another interesting method is to provide a fixed 2D minimap in a corner of the screen, always allowing the user to jump to a specific location with clear navigation capabilities [44].



Coupling

- Nr of components for use-case
- Nr of use-cases for component
- Nr of called services of component
- Nr of service-calls of component
- Nr of distinct service-calls of compo
- Nr of services of component
- Coupling
- Inverse coupling
- Complexity of services
- Complexity of services(2)
- Lack of cohesion between services
- Depth of scenario
- Width of scenario

BusinessTerm	1
DeleteAction	1
BewerkenController	1
BasicBewerkenDeelVenster	4
OfferteAanvragerBewerkenVenster	2
GoudseButton	1
GoudseMenuItem	1
GoudseTextField	2
OfferteAanvragerBO	1
OfferteAanvragerBOM	1
AbstractExchangeableService	1
AbstractPersistentService	3
AbstractPersistentBusinessObject	3
RelatiePS	2

Figure 2.13: *UML* diagram (left), Metric information (center), *MetricView*s visualization (right). Image from [45].

2.1.5 Metric Driven

Visualizations can be extended with metrics extracted from the corresponding code based. Some of them are extending well known *UML* diagrams by taking the standard notation and displaying either on top or behind the *UML* elements the visual representations of quantified metrics. The combination of metrics with their corresponding classes enable the viewer to locate classes matching a given set of metrics. They could indicate that some classes contain too much logic, or have the single purpose of encapsulating data.

MetricView

The core idea of *MetricView* [45] is visually summarized in Figure 2.13. *MetricView* takes a *UML* diagram and adds an additional layer of visual representation of the *UML* element's metrics. Beside class diagrams, the tool can extend sequence, state, use case and collaboration *UML* diagrams. An *UML* element can be assigned arbitrary many metrics. *MetricView* supports boolean and numeric metrics [45]. Boolean values are either shown as red crosses or green check marks. Integer values have several visualization options, like 2D rectangles (color coded ranging from blue to red), 2D height bars (y dimension), 2D circles (radius), 2D pies (circle arcs), 3D bars (z dimension) and 3D cylinders (z dimension) [45].

By adding an additional layer on top of the *UML* diagram, the metrics are visually covering information. Therefore, *MetricView* can configure both alpha values, one for the overlaying metrics and the other for the *UML* diagram. Making the visible metrics transparent, the user can access the information provided by the *UML* diagram again, moreover, by increasing the metric alpha and decreasing the *UML* alpha, the user can focus more on the actual diagrams, thus controlling their current focus between those two aspects. Besides the alpha values, the tool supports restricting the area in which the visual representation of the metrics are drawn, allowing the user for example to shift all metrics in one corner of the specific *UML* element, revealing the underlying content.

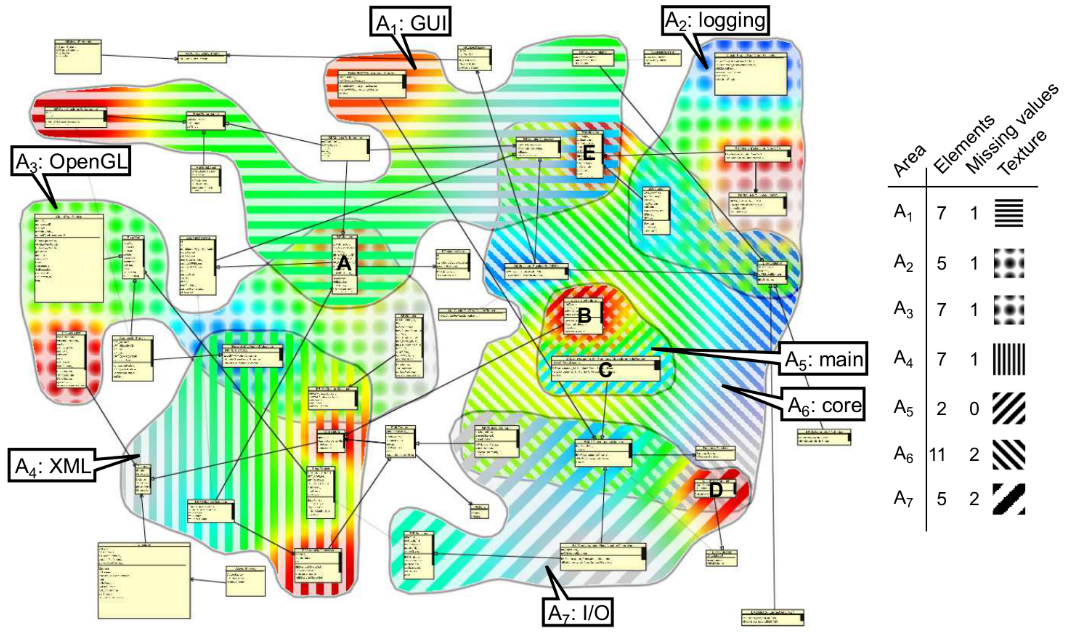


Figure 2.14: *UML* diagram extended with Areas of Intrest. Image from [46]

Area of Interest

The *Area of Interest* (AoI) approach from Byelas et al. [46] lets the user define collections of *UML* elements (areas). The tool needs a mapping of how much each element is involved in their respective *AoI*. This mapping is later converted into a color coding that ranges from red (high value) to blue (low value), or alternatively gray when no value can be calculated.

Figure 2.14 shows an example of 45 elements partitioned in seven *AoI*s. "The legend shows, for each area A_i, the number of classes it contains, the number of classes that have missing values for that area's metric p_i (due to the fact that we were unable to reliably estimate the percentage of code involved in each aspect), and the texture used to show the area. We notice several facts. Few classes participate in two aspects, and none takes part in three. This indicates a good functional modularity. The only class strongly involved in two aspects is B, part of the *main* and *core* areas. Since B is actually the system's entry point, this strong involvement is not a problem. Class E participates strongly in *core* (red in A_6) and weakly in *GUI* (blue in A_1). E is the *main* window, so its weak involvement in *core* and strong in *GUI* is correct. Class D is strongly I/O-related (A_7), and also part of the core (A_6). However, its code is quite complex, so we were unable to assess how strongly it belongs to the core (missing metric of D in A_6)." [46]

The tool provides five easily distinguishable textures, and the above mentioned color code. However, when more than three *AoI*s overlap, the visualization gets messy and hard to understand [46].

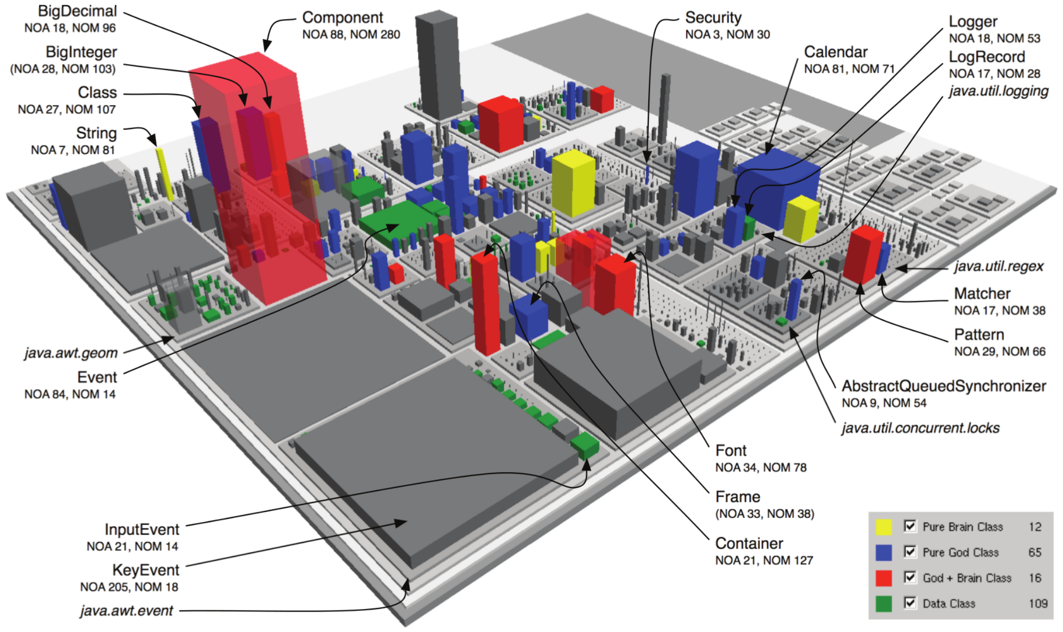


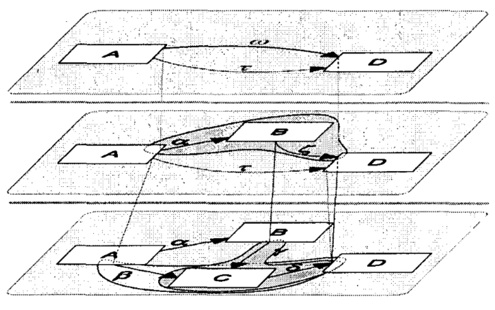
Figure 2.15: *CodeCity* visualizing the JDK 1.5 core. Image from [47].

CodeCity

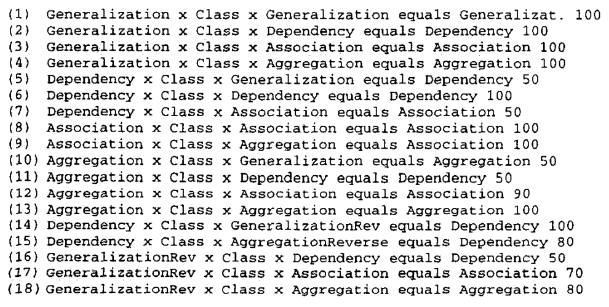
CodeCity [47] visualizes a given code base with the *City Metaphor* approach, with packages as nested districts and classes as buildings. Although all buildings have a quadratic floor area, their properties (size, position and color) are coupled with the properties of the classes. Figure 2.15 visualizes the JDK 1.5 core packages and its classes and interfaces (i.e. "the entire java namespace"[47]). The number of methods (NOM) and the number of attributes (NOA) are mapped to the buildings height and dimension respectively. This mapping allows the viewer to easily locate different types of classes. For example, the `KeyEvent` class is a wide but flat building, as it has only a few methods but many attributes. The `Component` class also catches the eye with its many methods. An opposite of the `KeyEvent` class (in terms of NOM and NOA) is the `String` class (located left from the `Component` class), with just a few attributes but many methods.

As the key in Figure 2.15 shows, *CodeCity* can map different colors to a predefined set of categories. The visualization highlights classes yellow which are classified as *Brain Class*, blue for *God Class*, red for a combination of the previous two and green for *Data Class*. Such classifications can be constructed with simple rules. For example, the *God Class* classification is defined as a class which uses more than a "few" attributes of other classes, its *Weighted Method Count* [48] is classified as "very-high" [49] or higher and the *Tight Class Cohesion* [50, 51] is less than a third.

Another interesting feature of *CodeCity* is its method-level view. Classes are still represented as buildings, but instead of one big block, the building is visually built from smaller blocks, one block for each method. Hereby, the overall height of the building is



(a)



```
(1)  Generalization x Class x Generalization equals Generalizat. 100
(2)  Generalization x Class x Dependency equals Dependency 100
(3)  Generalization x Class x Association equals Association 100
(4)  Generalization x Class x Aggregation equals Aggregation 100
(5)  Dependency x Class x Generalization equals Dependency 50
(6)  Dependency x Class x Dependency equals Dependency 100
(7)  Dependency x Class x Association equals Association 50
(8)  Association x Class x Association equals Association 100
(9)  Association x Class x Aggregation equals Association 100
(10) Aggregation x Class x Generalization equals Aggregation 50
(11) Aggregation x Class x Dependency equals Dependency 50
(12) Aggregation x Class x Association equals Association 90
(13) Aggregation x Class x Aggregation equals Aggregation 100
(14) Dependency x Class x GeneralizationRev equals Dependency 100
(15) Dependency x Class x AggregationReverse equals Dependency 80
(16) GeneralizationRev x Class x Dependency equals Dependency 50
(17) GeneralizationRev x Class x Association equals Association 70
(18) GeneralizationRev x Class x Aggregation equals Aggregation 80
```

(b)

Figure 2.16: (a) Relation abstraction from the most concrete (bottom) to the abstracted (top) layer. (b) A subset of rules for relation abstraction. Images from [52]

still influenced by the number of methods, whereas the number of attributes influences the horizontally aligned method-blocks composing one layer of the building. With this representation the tool is able to show design flaws on the method level, instead of just marking the whole class containing such a flaw.

2.2 Abstraction

Diagrams and overviews present a simplified version of an actual software system. The simplification should hide unimportant details from the viewer, letting him focus on the important parts of the given system. Such a simplification is often called abstraction and can be achieved in several ways. Common examples hide or collapse multiple elements to one super-element as already mentioned in Section 2.1.4. However, the difficulty is the decision whether an element is important or not.

2.2.1 Automated Abstraction

Egyed presents in the paper "Semantic abstraction rules for class diagrams" [52] a technique which distinguishes between classifier and relation abstraction. The former method is based on existing hierarchical structures (like packages, classes, states, et cetera) which can be grouped and collapsed to a simplified view. The previously existing relationships of the grouped elements are inherited by the abstracted element. Figure 2.16a shows an example of a relation abstraction. In the first step (from the bottom to the center layer), the edge from B to C, the node C itself and the edge from C to D are collapsed to a composite edge, connecting B directly with D. As the node C was abstracted, the edge from A to C gets redirected to D. In the next iteration (from the center to the top layer), the triple of node B and its incoming and outgoing edges, whereas the latter is already a composite edge, are abstracted to a new edge also connecting A and D.

The approach has to determine next the type of the newly created edge(s). Speaking of *UML* class diagrams, different types of edges may lead to different abstraction. Because of this behavior of the approach, the author argues that this kind of abstraction is

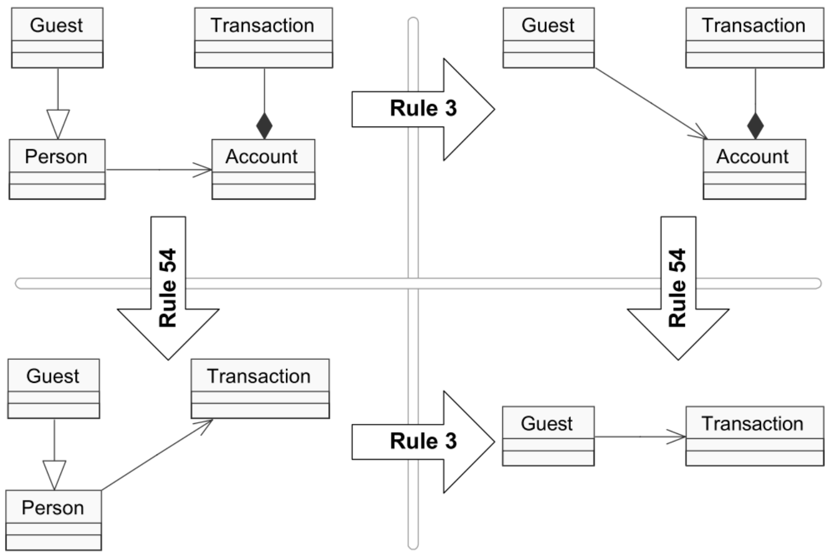


Figure 2.17: Symmetric serial abstraction. Image from [53].

superior to the classifier abstraction, as it allows “semantic abstraction richer than a pure classifier-based decomposition” [52]. Moreover, the relation abstraction does not need a strict hierarchy of classifiers.

The actual type of the edge is determined by a predefined set of rules. Figure 2.16b shows some of those rules which the author could derive. The original paper [52] introduced 49 rules of relational abstractions. This set of rules was later extended to a sum of 121 rules [53]. Each rule (one line in Figure 2.16b) has three parts. The first is the input pattern (left from “equals”), followed by the output pattern (right of “equals”) and finally a reliability indicator (trailing number). The input pattern has to contain at least two relations and a classifier, whereas the output pattern has to be at least one relation. Both, input and output patters, may be arbitrarily complex, but the output has to be simpler (in terms of containing relations) than the input in order to guarantee determinism.

Figure 2.17 gives an example as how the proposed method can be applied multiple times, also on composite edges. *Rule 3* has the input pattern of “GeneralizationRight - Class - AssociationRight” (the appendix “Right” or “Left” indicates the direction of the arrow in the *UML* diagram) and the output pattern “AssociationRight”. *Rule 54* consists of “AssociationRight - Class - [Agg]Association” as input and “AssociationRight” as output. The position of “[Agg]” (which is an abbreviation for *UML*s aggregation) indicates on which side of the relation the aggregation is in the diagram.

2.2.2 User Supported Abstraction

In “Tool-supported compression of UML class diagrams” [54], the authors describe the method of collapsing and re-expanding nodes of a given directed graph in detail. They

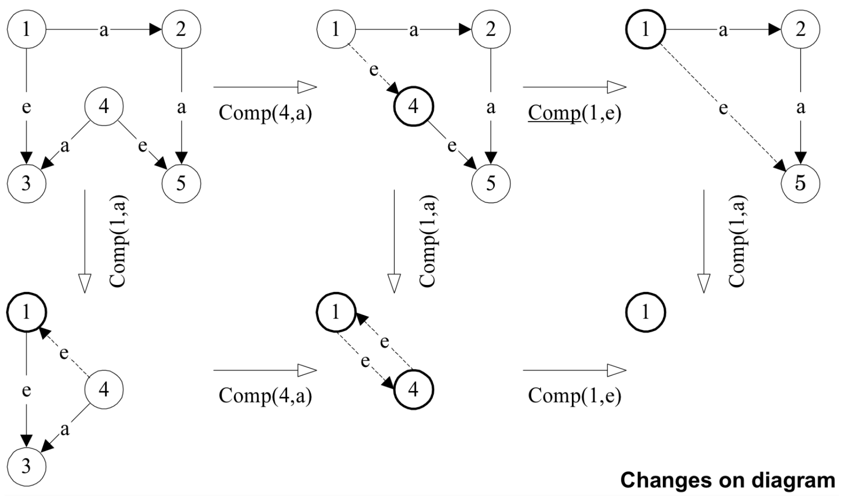


Figure 2.18: Abstracting a directed graph from top left to bottom right. Image from [54]

define that every node can be contained in an other node (whereas they further distinguish between *contained* and *container* node). The usual definition of a directed graph (a tuple of a set of nodes and a set of edges) is also extended by an additional set of edge labels, representing the *UML* based relation types. Figure 2.18 shows an example of the proposed method. Beginning with the top left graph, we have multiple options which eventually yield in the same result (bottom right). At the translation from top left to center left, we tell the algorithm to abstract[4] node 4 with all relations of type 'a'. Thus, node 3 gets contained in node 4, whereas the latter also inherits the edge from node 1 to 3. Node 5 is not affected from this step, as its relation type does not match the given parameter 'a'.

The authors applied this graph based method on class diagrams. The goal of their tool was to give the user a rather coarse diagram which can be extended depending on the user's current interest. Therefore, the question arises when is an abstraction valid. On the one hand, this could be answered quite simply with that the approach has to keep the semantic relations. On the other hand, this means that an extending class will inherit all associations of its super-class. Although this is technically correct, the authors argue that in most cases not the most concrete version of a class is interesting but the most abstract one. Therefore, the tool breaks the semantic validity. Instead of keeping the semantic validity, the tool is looking for relations "which somehow mean abstraction" [54], focusing on inheritance (in reverse direction), aggregation and implementation of interfaces.

[4]The authors call this step "compress", indicated by calling the function "Comp"

2.3 Analysis

Visualizations of software systems have usually the purpose to provide an overview and to improve the understanding of the given system. Although a good visualization already helps a lot, users may want to investigate the internal structure of the system in more detail, ideally related to the currently focused parts. This is where analysis techniques can be used. Such techniques may be very simple as “find class by name” which centers or highlights the found classes. Other techniques can be more complex, for example, finding dependencies or control flows of classes. In this section we will discuss two of the more complicated, but very powerful techniques, namely call graphs and dominator trees.

2.3.1 Call Graphs

Call graphs are constructed following all possible calls a method can make. There are two major types of call graphs, namely statically and dynamically constructed call graphs. The difference comes from the dynamic types of the classes. By statically analyzing the given code base, the actual class which will process a call cannot always be decided. The callee might be addressed via an interface, or the method could be simply overridden by an extending class. Thus, tools which create call graphs by static code analysis often simulate a call to every possible implementation.

In “Application-Only Call Graph Construction” [55] the authors give an example of a simple “Hello, World!” Java program which results in a call graph that contains more than 23,000 edges. As already mentioned, the reason for this is the dynamic binding. To overcome this problem, many approaches create analysis scopes. One popular approach [W2] is to simply ignore every method which is out of the program’s scope. However, this has the drawback that call-back methods registered in library classes are ignored too. Thus, the tool will miss methods inside the given scope which are indeed reachable.

```java
public class Main {
    public static void main(String[] args) {
        MyHashMap<String,String> myHashMap = new MyHashMap<String,String>();
        System.out.println(myHashMap);
    }
}
public class MyHashMap<K,V> extends HashMap<K,V> {
    public void clear() { }
    public int size() { return 0; }
    public String toString() { return "MyHashMap";}
}
```

Listing 2.1: Example Java program from [55]

The prototype tool *Call Graph Construction* (*CGC*) [55] creates a single node representing all methods which are outside of the current scope. This allows the tool to follow method calls, whether they are in a given scope or not and create dashed edges back to the program’s scope if necessary. Listing 2.1 shows a simple Java program which creates a new instance of `MyHashMap` and prints the object to the standard output stream.

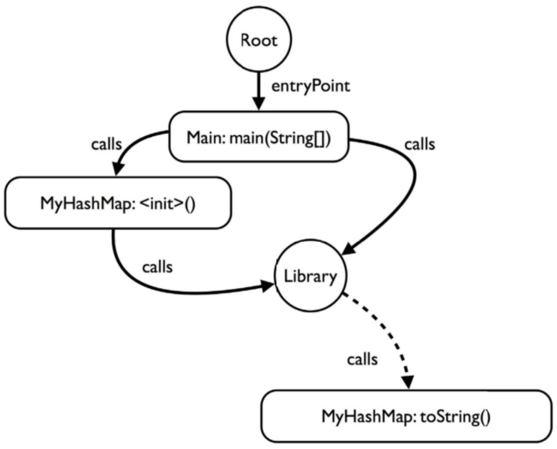


Figure 2.19: Generated call graph based on Listing 2.1. Image from [55]

The `MyHashMap` is a simple extension of the `HashMap` class, overriding the `clear()`, `size()` and `toString()` methods. However, Figure 2.19 shows the call graph which the *CGC* generates of the previous Java code. We see, that the `main` method makes a call to the constructor of `MyHashMap`[5], as well as a call to the *Library* node. This call is in fact the `println(Object)` call on `System.out`, as everything outside our program's scope is considered as *Library*. However, as the `println(Object)` internally calls the `toString()` method of the given parameter, thus, the *Library* also calls a method within our scope, namely our overridden `toString()` of `MyHashMap`.

Moreover, the authors argue, that the division of scope at the application's boundary (not including the standard libraries) is not arbitrary. They make use of the *separate compilation assumption*, which basically says that libraries had to be compiled without the knowledge about the current application's classes and interfaces. Apart from dynamic exploration of the system with *Reflection* for instance, this assumption seems to be realistic. The tool also has capabilities to trace an application class's canonical name. If such a string leaves the application scope, *CGC* assumes that the corresponding class might get instantiated from the *Library*. More details can be found in their paper [55].

As mentioned before, the dynamic type of classes (thus, dynamic methods) is only known at run-time. In "A framework for call graph construction algorithms" [56] the authors state that the creation of a call graph can be performed in two ways: Either with pessimistic or optimistic assumptions about the specific types. A too optimistic approach may lead to an unsound result as the approach could assume that there is only one entry point (i.e. the main function), so paths are not explored which are not reachable from that entry point. A pessimistic approach on the other hand could assume, that every call may be received by every possible candidate. This assumption will lead certainly to a sound result, but will contain unnecessarily many edges.

[5]Internally in Java all constructors are named `<init>`

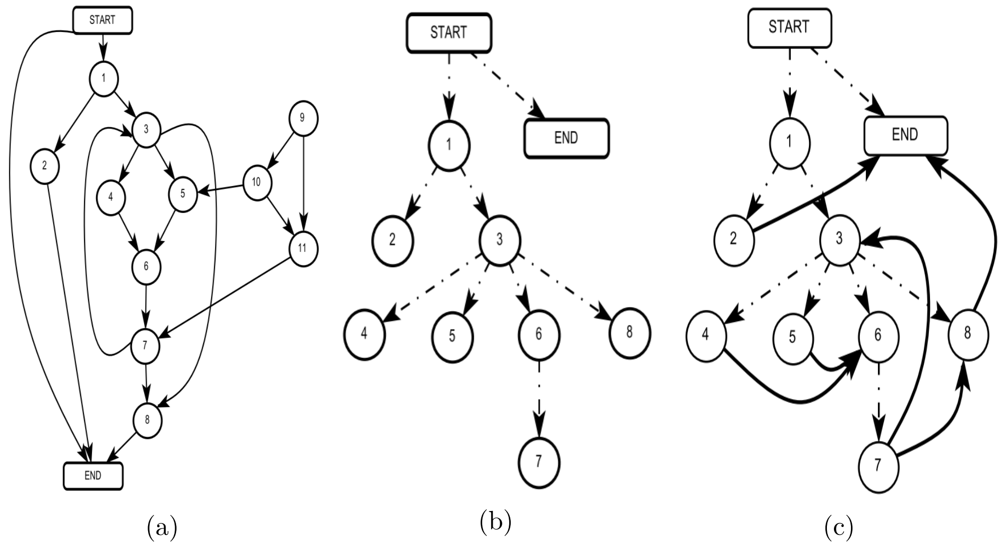


Figure 2.20: (a) A flow graph with its (b) dominator tree merged into a (c) *DJ-graph*. Images from [57].

2.3.2 Dominator Tree

Figure 2.20a shows a flow graph with several nodes, a *start* and an *end* node. Not every node has to be reachable from *start*. Figure 2.20b shows the corresponding *dominator tree*, which contains only the nodes reachable from *start*, thus nodes 9, 10 and 11 are not part of it. A node x *dominates* a node y if and only if all paths in the flow graph, from *start* to y has to pass through x. If x dominates y and $x \neq y$, x *strictly dominates* y [57]. This means that every node from Figure 2.20b dominates all of its successors. Applying this technique to class diagrams, the user could get an insight into the purpose of the class. Classes which dominate many other classes might be controller or key classes.

However, as the creation of a dominator tree is time-consuming, approaches have been developed for incrementally updating a dominator tree. Most approaches distinguish between reducible and irreducible flow graphs. In order to distinguish these two types of graphs, we first have to know that an edge from x to y is called a *backward edge* if y dominates x, or a *forward edge* otherwise. A flow graph is reducible "if the set of all forward edges induces an acyclic graph" [58], or irreducible otherwise. The importance of this property becomes visible, as many approaches [58, 59] are only able to calculate the dominator tree on reducible flow graphs. In the paper "Incremental Computation of Dominator Trees" [57] Sreedhar et al. present an algorithm which is capable of handling both types of graphs. This can be achieved by not relying on properties of the flow graph itself (which are not valid in irreducible flow graphs), but on properties of a so called *DJ* graph. A *DJ* graph is a mixture of the dominator tree and its corresponding flow graph as shown in Figure 2.20c. The dashed edges are from the dominator tree, whereas the additional solid edges come from the original flow graph. This *DJ* graph now has properties which are valid in both cases, reducible as well as irreducible flow graphs.

CHAPTER 3

Theoretical Background

The aim of this thesis is to help users to create an abstract class overview. Therefore, we will consider the desired overview as a graph with classes as nodes and relations as edges in this thesis. In order to generate the overview we need to extract the needed information from a given code base. This step is better known as reverse engineering. The extracted information has to be stored in a meta-data model. Once a graph structure is created from the previously gathered data, the nodes have to be assigned specific positions. This assignment will take the relatedness of classes into account.

3.1 Reverse Engineering

Gathering the information from a given code base can be done in two ways: parsing the source code or the compiled code. In case of Java both methods can be achieved easily with appropriate frameworks. The Eclipse [W3] project provides the `ASTParser` (Abstract Syntax Tree Parser) class, where a visitor interface has to be implemented and registered to the `ASTParser`. Given a source file, the parser analyses the input and calls the visitor for every event. These events are for example, start and end of a method, variable declaration, et cetera. Each call will contain all needed information. For example, in case of method start parameters will include the return type, name of the method, the method's parameter names and types, declared exceptions, and so on. An example of the `ASTParser` is shown in Listing A.1. The `ASTParser` engine parses the source file and calls the overridden event-handling methods. By gathering the information contained in the events, a custom data structure can be easily extracted.

However, parsing the source code can lead to ambiguity, as the example code in Listing 3.1 shows. In this scenario the class `MyClass` is a self-written class, whereas `Library` is an imported class from a library whose source code is not available. Parsing the source code alone leads to ambiguity as the extractor cannot decide which method is called.

```java
public class MyClass{
    public static void overloadedMethod(SomeType arg){
        // ...
    }
    public static void overloadedMethod(OtherType arg){
        // ...
    }

    public static void main(String args[]){
        MyClass.overloadedMethod(Library.getSomething());
    }
}
```

Listing 3.1: Source Code parsing - ambiguity method calls.

The problem of ambiguity can not occur when parsing bytecode, as the compiled code contains the information about the return type of `Library.getSomething()`. For Java there are some good bytecode parsing frameworks available, such as *Javassist*[W4], *ASM*[W5], *BCEL*[W6] and others. *ASM* uses the same approach as the `ASTParser`, that means the classes as `ClassVisitor`, `MethodVisitor`, et cetera, have to be extended. The *ASM* engine will call a specific method for each event, similar to the `ASTParser`. *Javassist* and *BCEL* parse the given class file internally and returns an Object which provides getter methods like Java's internal `java.lang.Class` does.

3.2 Data Model

The prototype has to map the parsed project into a suited model for further processing. The used meta model is shown in Figure 3.1. The model is strongly inspired by Java's standard *Reflection* interface. In order to avoid naming conflicts with the *Reflection* classes `Class`, `Method`, et cetera, the prototype's meta model classes all start with a capital D, resulting in class names as `DClass`, `DMethod`, et cetera.

We begin first with the `AbsData` class, which is a super class of all other meta model classes. It provides an id for internal purposes (`equals`, `hashcode` and `compare` methods), as well as a general purpose `Hashtable` (`userData`) allowing arbitrary storage space for all model instances.

The `DProject` contains a project name and an array of `DClasses`, containing all `DClasses` of the given project path. A relation to `DPackage` does not exist because the selection which packages should be contained is difficult. The problem is that Java allows a project to create classes in arbitrary packages, also in packages used by libraries. Thus, a package may contain `DClasses` of the project as well as from other sources.

`DClass` contains all extractable information about the given class. As most of the information is self-explanatory, we will focus on the less obvious attributes. The `DClass` stores both versions of its name, the full qualified name[1] (`name`) as well as the bare class name (`simplename`). The attribute `bCode` is a string representation of the bytecode of

[1]Class name prefixed by the full path of packages

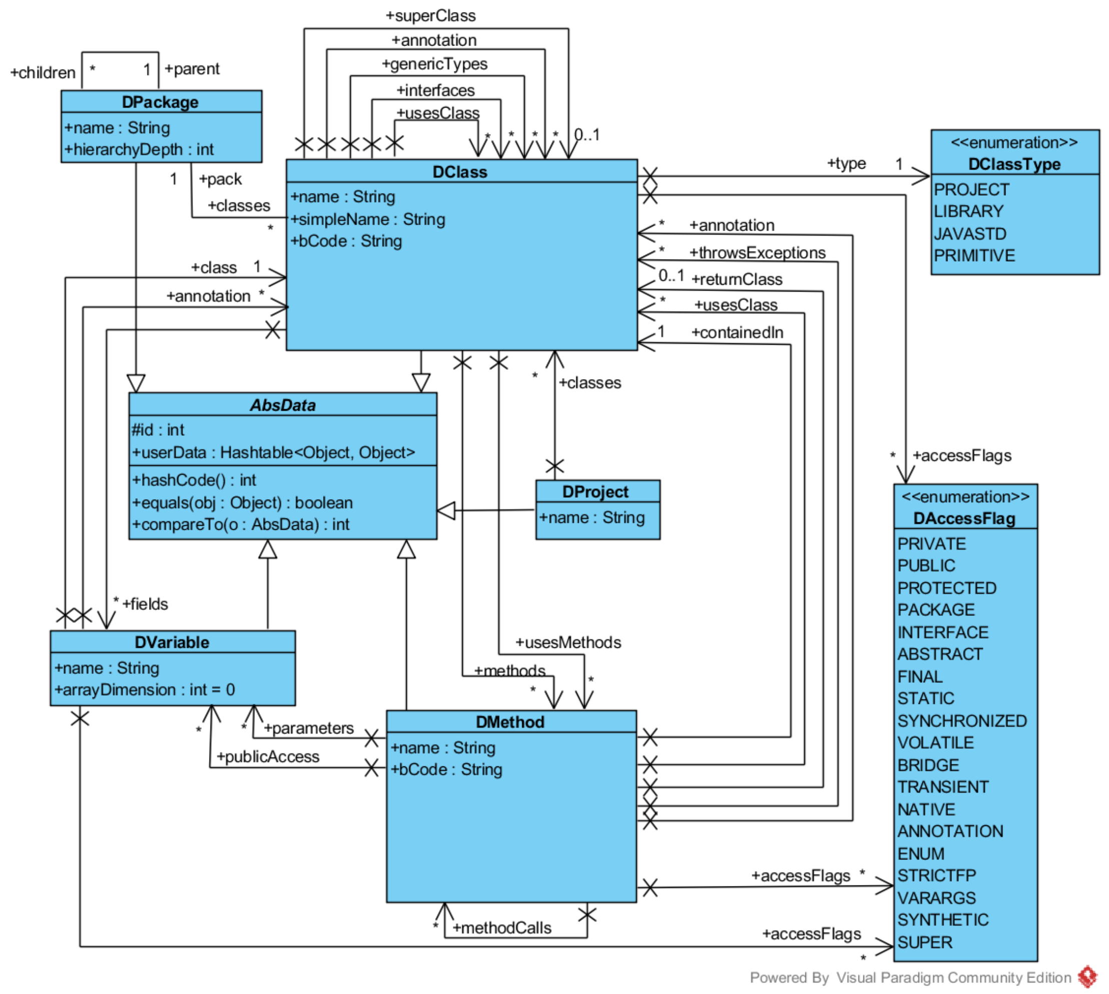


Figure 3.1: Core Data Structure of the Prototype.

the whole class. The usesClass and usesMethods arrays are calculated based on the class' methods (including the static block initializer, inline class variable instantiations as well as constructors) and have the sole purpose of improving the performance of some criteria (discussed in Section 5.3.1).

The type attribute refers to the DClassType enumeration. However, the enumeration item PRIMITIVE is only used for the primitive data types, which are for simplicity reasons also capsuled in DClass instances as well as their corresponding java.lang.Integer et al. versions.

DPackage stores its own name, its parent, its children as well as its directly contained classes. The hierarchyDepth is also used for performance improvements for later purposes.

Although most of the items of the DAccessFlag enumeration are well known, the enumeration also contains the Java intern keywords, where some of them are automatically generated by the Java compiler.

`DMethod` also has a `bCode` attribute as `DClass`. The `bCode` of the `DMethod` instance is a subset of the `bCode` from `DClass` where the current `DMethod` is stored in. The other mentionable attribute `usesClass` is an array which represents the classes whose methods are called by the current method.

The last class of the meta model package is the `DVariable` which is used as the data type for the field in `DClass` as well as for parameters in `DMethod`.

The `Node` class is worth mentioning as it connects (in addition to the `DProject`) the meta model data structure with the rest of the system. The `Node` class is used by the graph visualization engine and the layouting algorithm. Thus, it acts as a bridge between internal calculation data structures and visualization information. However, those aspects are capsuled strictly in sub classes, ensuring the separation of concern.

3.3 Definitions and Assumptions

In the next section, we will need some notations and discuss some assumptions. In this thesis, we will denote a graph as $G = \langle V, E \rangle$ where V is a set of nodes, and E a set of edges. Further, we assume that E has only simple edges such that only two vertices are connected with one edge. The coordinates of node i are denoted with x_i and $\|x_i - x_j\|$ denotes the vector distance of the two nodes. An existing edge between two nodes i and j is denoted as $i \leftrightarrow j$.

3.4 Positioning

When visualizing a class overview the viewer will likely expect the shown classes to be positioned dependent on their coupling to each other. Classes interacting with each other should be closer together, whereas classes which have nothing in common (no calls or any dependencies) should be further apart from each other. When creating such an overview manually, whether the creator is familiar with the system or not, the correlation of classes can be taken into account. If generated automatically, this feature of relatedness based positioning becomes a challenge.

This thesis inspects the usage of force directed graph layouting algorithms in order to solve this problem. To be more specific, we will focus on the technique described by Hu [60], which is based on the approach by Fruchterman and Reigold [61]. In the basic approach the nodes of the graph are considered as electrical charges, which are pushing each other away. Simultaneously, the edges are acting like springs pulling the connected nodes towards each other. With this simple idea the algorithm is able to untangle a regular mesh where the initial position of all nodes are completely random, as shown in figures 3.2a to 3.2c

Equations (3.1) and (3.2) are showing the basic force calculation for the repulsion and the attraction respectively. As Equation (3.1) shows, the repulsive force is defined such that every node influences every other node, leading to a worst case scenario of

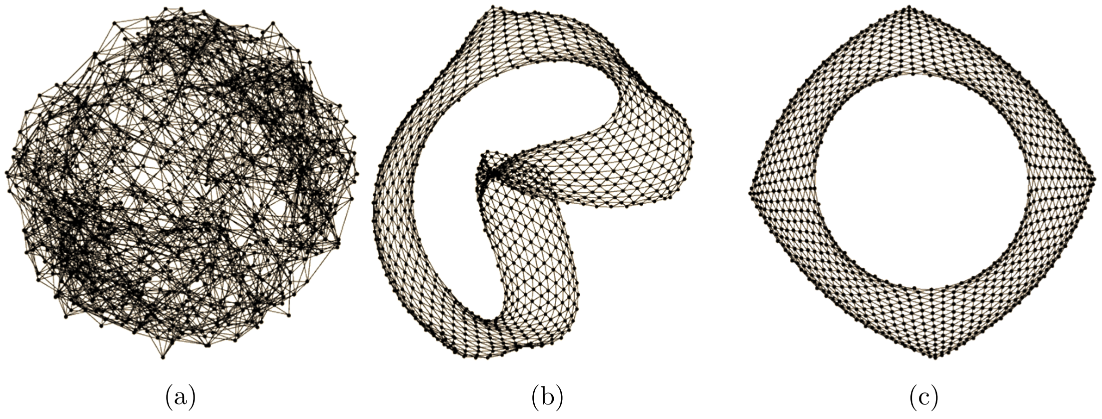

(a) (b) (c)

Figure 3.2: Force directed algorithm untangling mash with random initial positions. Images from [60].

n^2 calculations (with n as the amount of nodes). This problem is also known as the "N-Body Problem". Barnes and Hut[62] presented a technique which can lower the calculations to $O(n * log(n))$, and is discussed in more detail in Section 3.4.3.

$$f_r(i,j) = \frac{-C * K^2}{\|x_i - x_j\|}, \quad i \neq j, \quad i,j \in V \tag{3.1}$$

$$f_a(i,j) = \frac{\|x_i - x_j\|^2}{K}, \quad i \leftrightarrow j \tag{3.2}$$

Equation (3.3) shows the combined force of the repulsion and attraction, where the parameters K and C represent the optimal distance and the relative strength of the repulsion respectively. Hu [60] argued that the two parameters influence the overall size of the resulting graph, but have no effect on the stability of the algorithm.

$$f(i,x,K,C) = \sum_{i \neq j} \frac{-C * K^2}{\|x_i - x_j\|^2} * (x_j - x_i) + \sum_{i \leftrightarrow j} \frac{\|x_i - x_j\|}{K} * (x_j - x_i) \tag{3.3}$$

3.4.1 Termination

Iterative algorithms require a termination condition. Hu [60] uses the Equation (3.4) to calculate the energy of the system, where x is the vector of all coordinates. Hu's algorithm terminates as soon as the energy difference from the current iteration compared to the last iteration is below a threshold. However, the problem is that the algorithm might get in a cyclic state, where the nodes literally cycle around each other. If the movement per iteration is large enough (which is easily the case), the energy difference will never undercut the threshold, thus the algorithm will never terminate.

$$Energy(x,K,C) = \sum_{i \in V} f^2(i,x,K,C) \tag{3.4}$$

Simulated Annealing [63] is a well-known technique for iterative algorithms for eventually stabilizing, thus terminating. This approach multiplies every movement with a steadily decreasing factor. For example, if the algorithm's current iteration negates the effects from the last iteration, using simulated annealing the state will eventually not be able to completely nullify its previous state's effect. This factor is often referenced as the temperature of the system, where a high temperature allows the nodes large movements, whereas a low temperature cools the overall movement, eventually reaching the point of convergence.

However, the problem is, that this steadily decreasing factor limits the overall iterations to an almost constant value. Constant iterations tend to stabilize in local minima, considering the overall energy of the system. Hu extended the aforementioned approach. Whether the energy of the current iteration is higher or lower than in the last iteration, the *Adaptive Cooling Scheme* decreases or increases the temperature respectively. To prevent cyclic behavior, the temperature is only increased if the energy has been consequently lower for several steps.

3.4.2 Consistent Layout

Panas et al. [35] also extended a force directed layouting algorithm with the intention to make the overall result more consistent. Their approach combined the base layouting with a hierarchical algorithm. The hierarchy they used is the directional nesting structure of the source files in the file system. They put all files from the same folder close to each other. With this technique the force directed layout will not get completely deterministic, but it has a start layout which is very consistent, even if new nodes are added. In their paper they claim that the results of the computation of consistent starting layouts worked well, creating predictable visualizations.

3.4.3 Optimizations

In any force directed layouting algorithm there are two major factors influencing the overall performance, namely the calculations for repulsion and attraction. The algorithm can only be improved by either improving the basic equations, or by reducing the number of calculations. The former is out of scope of this thesis. However, for the latter there are two major optimization techniques, boosting the overall performance significantly.

Quadtrees

A naive implementation of a force directed layout will calculate the repulsive forces for each n^2 tuples. Barnes and Hut provided in their paper [62] a way of reducing the n^2 to $O(n * log(n))$. The approach is dividing the given space into four equally sized quadtrees[2], as Figure 3.3 shows.

[2]The original approach divided the three-dimensional space into eight octrees. However, as we are considering only two-dimensional graphs we will refer to this structure as quadtrees.

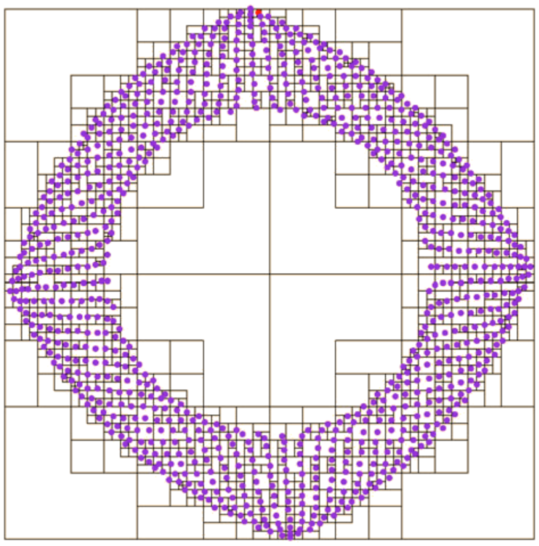


Figure 3.3: Quadtree dividing space in equally sized segments recursively. Image from [60].

The recursive quadtree structure will either store four children of its own type, or a data node, making the current quadtree element a leaf. While adding the nodes, every quadtree (leaves as well as non-leaves) keeps track of how many nodes they and their children are holding.

This structure allows the layouting algorithm to calculate the repulsion from one node to a set of nodes at once. This is done by considering the given set of nodes as one large super-node. The result of the default calculation between the node and the super-node is simply multiplied by the mass of the super-node, which is the amount of nodes in the quadtree (and its children, recursively). This method is less accurate than calculating all n^2 tuples, but if the distance is “far enough”, this inaccuracy is negligible. The term “far enough” is defined [62] as follows: Let l be the length of the current quadtree, D the distance between the current node and the quadtrees center of mass and θ an accuracy constant ~ 1. If the Inequality (3.5) holds, the nodes inside the quadtree are considered as “far enough” and are calculated as one super-node. Otherwise the node will be processed against all children of the quadtree recursively.

$$l/D < \theta \tag{3.5}$$

Multilevel Approach

Another optimization of the force directed layout is the multilevel approach [64]. This technique contains three steps: graph coarsening, layouting the coarsened graph and refining. The idea is, that graphs with just a few nodes are cheap to position. When they have a stable position, the graphs are refined again. The new nodes are already close to their ideal position (considering the global minimum of the system’s energy).

Graph Coarsening can be done in many ways. One method is to collapse edges, where the adjacent nodes of an edge are collapsed to a super node. Super nodes additionally store a weight, representing the amount of nodes they contain. Edges also have weights, but their value only changes if both collapsed nodes share the same neighbor. The new edge to that neighbor gets the sum of weights of the collapsed edges assigned. Using *maximal matching* for selecting the edges to be collapsed prevents selecting the same node twice in one coarsening step. Other methods use already existing edge weights, preferably collapsing the heaviest or lightest first, or to try to equalize the inherited edge weights over the multiple coarsening steps. Yet another approach is to calculate the maximal independent vertex set, which is a set where no two nodes of the set are adjacent to each other.

Layouting the Coarsened Graph has to be distinguished in two cases. The first case is used for the most coarsened graph (that is, the graph with the least nodes and edges), which has to be given an initial layout to start working with. In case of a connected graph, the coarsening can be stopped when there are just two nodes left. These two nodes can be positioned randomly. If the graph is not connected, we can consider the given graph as two separate graphs where both are internally connected graphs. If the current graph is not the most coarsened graph, we can take the current positioning as initial layout for our force directed algorithm.

Refining reverses the effect of coarsening the graph. Every created super node of the current coarsening level is replaced by both nodes it was holding. However, before we can continue with the re-layouting of the refined graph, we have to adjust these positions. When replacing a super node with its two children, there will be zero distance between the children. This would cause a zero division in the repulsive force. Even if we apply a minimal gap between the two nodes, both children will have a huge repulsive force against each other causing large movements in the next iteration. Such large movements could destroy the inherited layouting information from the parent graph. We will later discuss that this problem does not apply to this thesis's prototype. However, the problem is discussed in detail in Hu's paper[60].

CHAPTER 4

Methodology

The basic goals of this work are to generate a class overview and to provide the user with abstraction and analysis capabilities. This chapter focuses on the used methods, approaches and exposed changes made during the development.

4.1 Language, Tools and Libraries

The proof of concept prototype is written in Java 7. Although the prototype should be able to work with Java 8 too, at the time of developing only a few bytecode parsing libraries supported that version of Java and most of them were still experimental. Therefore the prototype was constrained to Java 7.

4.1.1 Tools

For most of the development the IDE Eclipse Luna (4.4) [W3] was used. The source code was managed with Git 1.9.5 [W7] as the client and GitLab 6.5.1 [W8] as a server application. As an organization tool and an issue tracker Redmine 2.5.1 [W9] was used. Although the prototype is not based on them, the open-source tools Gephi [W10] and D3 [W11] influenced the prototype's final implementation of the positioning algorithm.

4.1.2 Libraries

For parsing the source code the prototype has an implementation based on ASM 5.0.3 [W5] and for bytecode BCEL 5.2 [W6]. The visualization is powered by the JUNG 2.0.1 [W12] framework. Profiling was performed several times during the development, mostly for performance issues. For this task the *JVisualVM* was used, which is a standard tool shipped with every JDK distribution.

4.2 Origin

One of the main challenges of this work was clear from the very beginning. The concrete positioning of the nodes alone almost defines the quality of the overview. Thus, the first research iteration had the goal to find an adequate layouting algorithm which suits the requirements such as positioning related classes close and non-related classes far away from each other. The research resulted quite fast in the force directed layout. However, development showed that there were decent changes needed in a usual implementation of such an layouting algorithm to meet the requirements of this work.

After a calculated positioning, the user should be able to tell the system which classes are more or less important. Depending on the user's configuration, a cumulative rating should indicate an importance for each class. The user then can hide or show classes rated as important or not. For this rating categories are provided. Each category has among others an importance and can be freely defined which classes should match the category or not. For convenience the user can bundle some categories in filter sets, providing the option to enable or disable a whole set of categories at once.

4.3 Development Phases

Developing the proof of concept prototype can be split into six phases. The first few phases are mostly about building the framework and gathering the needed data. Phases four and five are adding some of the core features, whereas the last phase adds multi-threading and contains some performance tests.

4.3.1 Parsing

Everything in this prototype is based on the data to represent. Therefore, in this first phase, the first parser was implemented. This parser was a source code parser based on Eclipse's `ASTVisitor` [W13]. However, parsing the source code alone leads to ambiguity, as the parser does not have the information about the return type of a function if the return value is not explicitly stored in a typed variable. This ambiguity was discussed in more detail in Section 3.1. As a consequence the parser was modularized, and a second implementation was added, this time bytecode based using BCEL 5.2 [W6].

The gathered data was stored in a Java's `Reflection`-like data structure, which is described in more detail in Section 3.2. As the overall aim of the tool is a visualization of classes and their relations, the data was converted into a graph (classes as nodes, any kind of relation as edge).

In order to visualize the resulting graph, the JUNG 2.0.1 [W12] framework was used. However, the first phase only included some experiments with the visualization framework, as the actual positioning of the nodes was tricky.

4.3.2 Spring Layout

This still very experimental phase made use of JUNG's implementation of the force directed layout, which is also known as `SpringLayout`. First visualizations were achieved in this step. The main idea of using the force directed layout looked promising. Although most nodes were overlapping each other, only a few edge crossings were existent. This could be investigated with the integrated navigation and zoom capability of the JUNG framework.

In the same phase the JUNG's default implementation of the force directed layout was extended. This first extension was a very simple coupling-indicator, namely the package distance of the two nodes. As Java's package structure is a tree structure, the distance was calculated quite easily. The idea was that classes in the same package should be closer together as to others. Although this coupling-indicator was removed in a later extension (with a new implementation of the force directed layout), it was the first idea of clustering the nodes based on properties of the corresponding class.

4.3.3 Criteria and Categories

Based on the findings of the previous stage, the concept of categories was introduced. The purpose of categories was to at least visually cluster nodes (more details in Section 5.3.2). To be able to define which nodes should be assigned to which category, criteria were used. These criteria and the combination of those in form of criteria-expressions are discussed in more detail in Section 5.3.1.

This phase also included the first attempt of collapsing and expanding a defined set of nodes. This set was based on categories, which was changed later. Beside the collapsing capabilities, the categories had a simple yet visually beneficial effect on the nodes, namely to color the nodes based on the category with the highest priority. Another look at the rendered overview has shown that by creating categories based on the major systems of the prototype (graphical user interface, data model, positioning algorithm, et cetera), the differently colored categories were pleasingly clustered.

Although the main purpose of categories changed in a later phase, the categories are still used in the final state of the prototype. Criteria and categories are the base for every analyzing utility the prototype supports. Those include simple data property queries as *Name* and *InPackage* but also some more sophisticated as *CallsClass* or *DominatesClass*. More information about those can be found in Section 5.3.1.

4.3.4 Reimplementing the Force Directed Layout

The `SpringLayout` had the advantage of being already provided by the framework. With just a few adaptions it calculates iteratively the desired positioning. However, the performance of the algorithm was quite bad. After some research, two other implementation candidates were found, namely the D3 [W11] written in JavaScript and the Java-based Gephi [W10] project. Both of them already implement the N-Body optimization. Calculating the attraction or repulsive forces between n bodies, this optimization considers a set of nodes (bodies) which are “far enough” away from the current node as one super-node with the cumulative mass. This basically reduces the n^2 calculations to $O(n * log(n))$. This optimization was discussed in Section 3.4.3). Both, Gephi and D3 had this optimization already implemented, yet with slight differences. Gephi's implementation is a traditional approach, adapted for the purpose of the force directed layout. The implementation uses the Barnes and Hut opening criteria (discussed in detail in Section 3.4) and returns a force vector for the requested node. The D3's implementation on the other hand takes an additional function which gets called back for each visited quadtree-node. If this callback function returns false, the quadtree-traversing goes one level deeper, else it assumes the callback function already took care of all contained nodes. D3 calls the quadtree's visit function with a combination of a force directed layout and the Barnes and Hut opening criteria. This approach has the advantage that it allows other algorithms to efficiently traverse the quadtree too. Some other “addons” for the D3 quadtree are clustering or collision prevention.

For the prototype, both versions were adapted. Although the D3 implementation has some great features considering the reusability of the quadtree, the prototype is still based on the Gephi's implementation. The major reason for this is performance, which could be the result of the Java-nature of Gephi, whereas D3 had to be translated first. Another reason is that Gephi already had the distance calculation nicely separated in distinct classes. This made adaptions as the rectangular distance calculation (see Section 5.2.1) much easier. However, in this phase there were already several (differently adapted or translated) versions of the force directed layout tested. Therefore, also the layouting mechanism was modularized, and can be easily exchanged.

As the force directed layout is an iterative algorithm, an additional layer was introduced. This layer gives the user of the tool control over the iterations. The animation can be stopped and single-stepped forward. This utility feature lays between the JUNG framework and the rest of the system. That means, this feature will work any iterative positioning algorithm. This utility proved itself very useful when debugging the force directed algorithms and for deeper understanding how those are working internally.

4.3.5 Structured Elements

The fifth phase finally extended the force directed layout with structured elements. These elements are treated the same as the data nodes which are containing class information. This property leads to major changes and adaptions to the force directed layout, which we will discuss in more details in Section 5.2. Beside the addition of structured elements, this phase also included a collision prevention and resolution, also discussed in Section 5.2.

As with this phase the user customized structured elements are used splitting the system into parts, the categories lost their ability to collapse and expand. This functionality was shifted into the structured elements.

Performance was also considered in this step. Although the positioning calculation is still the slowest part of the system, the querying of criteria expressions turned out to be much slower as expected. This was especially true when the expressions got long and complex. Therefore, the previous binary tree implementation got replaced with a two layered nested B-tree implementation (more details in Section 5.3.1).

4.3.6 Multi-Threading and Tests

The previous phase can be considered as the final step for the core functionality regarding the theoretical parts. As the positioning algorithm was still quite slow, which could not be further improved in the scope of this work, the prototype got multi-threading support. Although this does not make the algorithm itself faster, the overall performance gets boosted. Tests and their results regarding the performance of single- or multi-threaded can be found in Section 6.2.2.

As most parts of the system was modularized and loosely coupled, also the multi-threading is very customizable. The layouting function returns a nested data structure with gives the algorithm total control which chunks should be executed in parallel and where to join all executions before progressing to the next calculations. More details about this feature can be found in Section 5.4.2.

4.4 Method Adaptations

During the development some of the planned approaches had to be adapted or extended. For example, at the beginning the categories had a collapsing and expanding feature. This caused a problem, as one class could be part of multiple categories by design. Collapsing the first category works flawless, but when the second category contains an already collapsed node, in which “super-node” should that node be assigned? The fifth phase solved this problem by transferring this feature to the introduced structured elements. This data structure has a strict hierarchical nature, forcing the data nodes to be assigned to exactly one parent element.

This change made the concept of category-based importance rating obsolete. Although in the final version the categories still have an assignable importance value, this rating only influences the color of the data node.

Another large adaptation was already mentioned in the first phase. While developing the parser, the source code approach seemed more viable for debugging purposes. However, as mentioned a major problem occurred, which forced a reimplementation of the parser based on bytecode.

CHAPTER 5

Implementation

The prototype proposed in this thesis provides an interactive visual overview over object oriented code. The required data to generate the visualization is gathered through reverse engineering. The classes and relations in between them are used to calculate a self-organized and flexible layout. The concept of structured elements, which is introduced in this thesis, enables the user to organize the classes in recursive, logical, and visual containers. Categories let the user highlight or hide a configurable set of classes for further analysis.

The implementation of the prototype can be split into several parts. These are reverse engineering the given code, converting the extracted data into a graph, and calculating a modified force directed layout. The graphical representation continuously animates the intermediate steps of the layouting process. Finally the user can tweak the result by using the capabilities of configurable categories.

5.1 Reverse Engineering

As mentioned in Section 3.1, there are two possible approaches of gathering information from the project, either by parsing the source or the compiled code. The prototype was initially based on the bytecode parsing framework *BCEL*. However, the current stable *BCEL* version 5.2 unfortunately does not support Java annotations. As the prototype was developed with a modularized concept in mind, BCEL could quickly be exchanged with another parsing library. This decoupled parser theoretically also allows us to parse non-Java code, although the used data model is highly inspired by Java's internal structure, as discussed in Section 3.2. If non-Java object oriented code is parsed, thus not all fields of the model are used, criteria which are depending on the missing fields might not work (as the "InPackage" criteria which is explained in Section 5.3.1).

The abstract parser expects the implementation to create instances of `DClass`, `DMethod` and `DVariable` (for instance variables as well as for method parameters) and fill those with appropriate data. The parser automatically extracts the `DPackages` based on the `DClass.name` which has to be the full qualified class name and assigns the `DClassType` to each `DClass`. Because the order in which the classes and methods are parsed (thus created) is not determined, the implementation is expected to use the `AbsData.userData` hash map where the key is the purpose of the reference (method invocation, parameter types and names[1], super class, implemented interfaces, field signatures, and so on) and as value a list of string based Java standard signatures. These signatures will be parsed in a later step to build the references. The used hash map will be cleared at the end of the parsing.

The abstract parser performs the following steps:

1. Find the root package in the file system. The parser can be called with an arbitrary class contained in the project. This step points to the root of the project for further parsing.

2. Traverse all folders recursively and let the implementation create the data objects based on each found file.

3. Link the super classes and the interfaces. This is needed in later steps which have to take inheritance into account.

4. Build the method parameters. The methods themselves were already created by the parser implementation in step 2. Now we build their parameters in order to distinguish overloaded methods (same name, different parameter count and / or types).

5. Link all methods and class references. This step may create copies of overridden methods and modify those. The references of those copied methods might be relinked to point to overridden methods.

6. Build convenient arrays like `DClass.usesClass` or `DClass.callsMethods`.

7. Build package tree structure and link all directly contained `DClasses` to the `DPackage`. This step also includes an enumeration of package depths.

8. Clear all hash maps as the user might use those later.

9. Initialize all not yet initialized arrays with zero length arrays. This is also for convenience so the developer can skip null pointer checks.

However, some limitations still remain. The parser only supports static parsing, which means that the code can only be inspected with its static typing.

In preparation for the next step, the gathered data objects have to be converted into a graph. Such a conversion is straight forward as all classes are mapped to nodes one-to-one. Edges are treated more configurable as it is useful for later analysis if the edges are represented either like in a class diagram, or a call graph.

[1]Parameter names are only available if the source code was compiled with the corresponding flag.

5.2 Positioning of Nodes and Structured Elements

An intuitive way of positioning classes in an overview is by their relatedness, as mentioned in Section 3.4. Classes with high communication or relations should be close to each other. Additionally to the usage of a force directed layout, the prototype introduces structured elements. A structured element is a collection of classes which are selected by a criteria-expression. Criteria, criteria expressions, categories and structured elements are discussed later in Section 5.3.1.

The prototype's implementation of Hu's approach [60] was initially strongly inspired by the Gephi [W10] project. The prototype faced two major challenges while adapting this implementation in order to be compatible with structured elements. First, the original approach considers the nodes as points, rather than two dimensional shapes (in our case simple bounding boxes). This may lead to overlapping nodes which drastically lowers the clarity of the overview. Second, structured elements can get much larger as simple nodes by design. Both of this problems will be discussed in next sections.

5.2.1 Rectangular Distance Calculation

The prototype considers nodes as rectangular objects. Therefore, we cannot use the Euclidean distance of the center points of our nodes. Instead, the prototype calculates the rectangular distance by calculating the four distances, assuming the second node to be above, left, right or below the first node. If only one of these distances is positive, the algorithm returns the distance and the corresponding direction. If two of the distances are positive, the second node is diagonally shifted to the first node. Then the algorithm calculates the resulting distance with the Euclidean distance formula between the two corner points. Three or four positive distances are mathematically not possible (a node cannot be above and below the other at the same time without overlapping [2]). In case of overlapping nodes, none of the four distances are positive. Here the algorithm returns the maximum of the distances with the corresponding direction. By returning the maximum distance (i.e. the closest to zero as all distances are negative), we choose the direction in which the overlapping can be resolved with the least needed iterations.

However, when a negative distance values is returned, the outcome of the repulsive and attractive formula will be inverted. For short distances (regardless whether positive or negative) the repulsive force will overwhelm its counterpart. As a consequence the already overlapping nodes will move even closer to each other. Therefore, the prototype simulates a very small distance (prototype uses 0.01, for comparison the optimal distance is set to 100) instead the negative value but keeps the calculated direction. Thus, the attraction force is almost nullified, whereas the repulsive force gains much more strength, resolving the overlapping eventually. Listing B.1 contains a pseudo code of the prototype's rectangular calculation. The pseudo code refers to rectangular position cases, depending how the two given rectangles are positioned relative to each other. Table B.1 shows

[2] We assume positive width and height values for the rectangles

a complete list of possible combinations, where the blue rectangle (first parameter) is considered as the origin.

As mentioned in Section 3.4.3, a low distance cause a huge movement which might destroy an almost stable layout. This effect is avoided with *Gephi* project's implementation of Hu's approach. The algorithm normalizes all movements, thus allowing only small movements. However, as this normalization increases the number of needed iteration, the prototype multiplies after the normalization the current temperature to all movements. This has the effect, that overlapping nodes will cause huge movements, which get normalized, suppressing originally small movements. Thus, the prototype first resolves every overlapping and afterwards all other nodes can move to their desired position.

5.2.2 Structured Elements

Structured elements act as a logical and a visual container, keeping the contained nodes and nested structured elements together and foreign elements out. Another desired behavior is that nodes with edges leaving the structured element should be positioned closer to the border than nodes without such edges.

As mentioned earlier, a structured element easily gets much larger by design than a data node representing one single class. However, structured elements are also considered as nodes with the visual behavior that they are always painted to the background. Therefore, they also have bounding boxes representing a cumulative bounding box of all their children plus a small padding. Further, the size or position of a structured element cannot be directly changed, only by changing the properties of its children.

In Section 3.4.3 we discussed a multilevel approach where parts of the graph got substituted by coarsened graphs. The prototype adapted this approach, using the structured elements. Every structured element is treated as a coarsened graph. Each node is affected by their neighbors and the neighbors of their parents, whether they are data nodes or structured elements. Unlike the multilevel approach, the prototype does not need a refining step.

However, as discussed in Section 5.2.1 the prototype has to adapt the calculation algorithm and might adjust the calculated distance as well. The adapted algorithm iterates over every data node (as only those can move), repulsing against every neighbor within its own parent structured element. This includes all sibling data nodes as well as sibling structured elements, but excludes the data nodes within the sibling structured elements. Within the same iteration, the data node is repulsed against its grandparents structured elements (without their data nodes), and their great-grandparents structured elements, and so on.

When calculating the repulsive forces, we have to distinguish the following cases. In every case the first node is a data node.

1. The second node is also a data node.
2. The second node is a structured element and a sibling of the first node, i.e. the two nodes have the same parent.

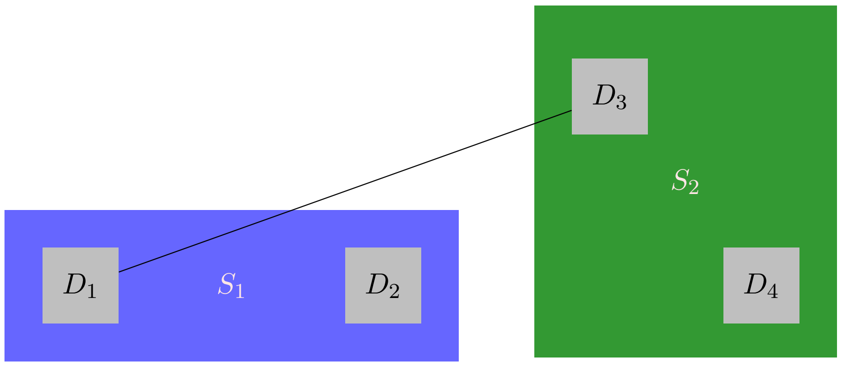


Figure 5.1: Data node D_3 replused by non-sibling structured element S_1, but attracted by D_1.

3. The two nodes do not have the same parent (this implies that the second node has to be a structured element)

For the first two cases, the calculation using the rectangular distance formula does not change. The problem in the third case is shown in Figure 5.1. The data node D_3 (first node) is repulsed by the structured element S_1 (second node). D_3 is pulled towards S_1 as D_3 is connected to D_1. In this scenario the nodes D_1, D_2 and D_4 are considered as not movable (due to other not shown pull and push forces or by being locked by the user). If we do not adjust the distance, the positioning algorithm moves D_3 above S_1 and from there, further to the left. As D_4 does not move, S_2's size will extend to keep D_3 contained, resulting in a collision between S_1 and S_2. As structured elements will never repulse each other because only data nodes are movable (if not locked), this collision might never get resolved.

The prototype's algorithm overcomes this problem as follows. First, it calculates the distance between the two structured elements S_1 and S_2. This distance (with direction) is used instead of calculating the distance between D_3 and S_1. If the distance is lower than a threshold (in the prototype the same value as the optimal distance), it neglects the y-axis (or x-axis if the scenario is rotated at 90 degrees) in the distance calculations. This simulates that D_3 has the same height as its parent S_2, preventing D_3 moving above S_1. If the distance between the structured elements is higher than the threshold, the two structured elements are considered to be far enough from each other, such that no collision might occur in the current iteration. In this case the real distance (between D_3 and S_1) is calculated and used for the movement calculations.

Another adjustment in the distance is the subtraction of all padding from the bounding boxes of the crossed structured element borders. This is calculated by the tree distance of the first data node to the second node, multiplied with the used padding constant. This becomes important, when the sum of padding gets higher than the optimal distance.

For a more convenient overview, all nodes have a weight, proportional to their children. As a consequence, structured elements containing many nodes will have a larger margin to their siblings as nodes with just a few children. Moreover, data nodes can be considered as leaves, not able to store children. Thus, they can never have a margin as large as well filled structured elements.

5.2.3 Indeterminism

As mentioned earlier, the force directed layout pushes nodes away from each other. The strength is based on the distance, and the movement vector is simply the delta of the nodes positions. However, when two nodes are on the exact same position, the distance and the movement vector are both zero. In this case, the overlapping cannot be resolved. During the layouting process, such a scenario is quite unlikely, because before the nodes move to the same position, they should have repulsed each other. Yet another cause that this is unlikely are the edges of the nodes. If the nodes have at least one different edge, this edge pulls the corresponding node to a slightly different position, thus resolving the overlapping. However, in the beginning of the layouting process the nodes have to be assigned a starting position. It is quite common to lazily initialize the positions to the origin, or the center of the drawing panel. Thus, all nodes are on the very same positions. Even the edges cannot pull them apart, as the other endpoint of the edge is on the same position too.

To overcome this problem, most implementations add a small random movement to one or both nodes. Although the prototype also has the random movement for the case described above, it calculates the initial layout with a cheap radial algorithm which is package based, strongly oriented on the approach described in Section 3.4.2. The deeper the class is in the package structure, the further it gets pushed away from the center. As the Java language allows multiple classes located in the default package, those classes are not positioned in the very center, but on the most inner circle of the radial layout. However, even if this radial layout is deterministic it only influences the force directed layout and does not guarantee that two render processes to be fully equal.

5.3 Analysis

When the layouting algorithm has provided a stable layout, the user may want to further investigate the system. The user is assisted in this with the analysis functionalities of the prototype, which are based on two mechanisms: Categories and structured elements. We already have discussed the impact of the structured element on the positioning of the data nodes in the previous section. Although both are based on criteria and composed criteria expressions, they have different impact on the resulting visualization. Before we discuss the purpose and the influence of these mechanisms, we will take a look on criteria and criteria expressions, as both categories and structured elements are based on them.

5.3.1 Criteria and Criteria Expressions

A criterion is a validation mechanism, checking if a given data node matches the criterion or not. Table 5.1 shows the criteria which are currently implemented by the prototype. Input parameters marked with an asterisk are considered to be string based regular expressions. For unused parameters (null values) the corresponding property is considered as matching.

Each criterion is atomic and returns either true or false. A criteria expression can combine multiple criteria. The prototype currently supports the three basic operators, && (and), || (or), ! (not) as well as parentheses. The prototype's implementation of the expression tree is performed as a disjunctive normal form, but instead of restricting the conjunctions to atoms, the prototype also supports nested expressions. This is achieved with a list of criteria expression lists. The elements contained in the outer list are linked with logical disjunctions, whereas the elements of the nested lists are connected with logical conjunctions. This data structure allows the prototype to discard all parentheses while it supports a fast iteration without requiring the tree to be balanced.

The core method of the criteria receives an instance of `DClass` and returns a boolean value, whether the parameter matches the criteria or not. However, as some criteria need preparations in order to efficiently calculate the result (i.e. call graphs and dominator tree), every criteria can override the `reInitialize` method which is called whenever an edge or node has changed. This method receives the current `DProject` instance, containing every gathered information about the system to analyze.

5.3.2 Categories and Structured Elements

Categories are mainly used to visually highlight specified data nodes. A category consists of a name, a priority, background- and text-colors, a criteria expression and a flag if it is enabled or not. One data node may be assigned to multiple categories, in which case the category with the highest priority will influence the data nodes colors. If two or more categories have the same priority, the first of them is used. The suppressed categories may still be used by criteria like *DominatesCategory*.

Further, categories have the boolean property "Remove matching Nodes". As the name suggests, every data node matching the criteria expression of the category will be removed from the graph. This allows the user to clear the overview from cluttering nodes. Especially in combination with the "Not" criterion, the user can define everything outside of their current focus. Then, by enabling the "Remove matching Node" property, the user can focus on their very specific part of the graph, with a strongly reduced amount of data nodes.

Structured elements have a name, a color, a parent and a criteria expression but are internally handled differently from categories. Except for the "Remove matching Nodes" property, a category has merely visual purposes, whereas internally a structured element extends the same `Node` class which the data node is extending. Also, as already mentioned, structured elements greatly contribute to the overall positioning of the data nodes.

Unlike categories, a data node can only be assigned to exactly one structured element. If multiple sibling structured elements have a matching criteria expression, the first one takes the data node. If no structured elements criteria expression matches the data node, a default root structured element accepts every data node, ensuring that all data nodes have a valid parent. Structured elements may also have nested structured elements as

Name	Description
CalledByClass	checks if the class is called by a given class*.
CallsClass	checks if the class calls a given class*.
ConntainsByteCode	checks if the class contains given bytecode*.
ContainsField	checks if the class contains a given field. The field can be described with the following list of parameters: • A list of `DAccessFlags`. • A data type*. • A name*. • An optional integer of the dimensions. Default value is 0.
ContainsMethod	checks if the class contains a given method. Method description contains: • A list of `DAccessFlags`. • A return type*. • A name*. • Arbitrarily many parameter types* and names* (tuple-wise).
DominatesClass	checks if the class dominates the given class*.
DominatesCategory	checks if the class dominates all classes within the given category.
ExtendedBy	checks if the class is extended by the given class*. Recursive checks can be enabled using the second boolean parameter.
Extends	checks if the class is extending the given class*. Recursive checks can be enabled using the second boolean parameter.
HasAccessFlag	checks if the given class* or interface* has a given `DAccessFlag`.
HasClassType	checks if the class's or interface's `DClassType` (as defined in Section 3.2) matches the given parameter.
ImplementedBy	checks if the interface is implemented by a given class*.
Implements	checks if the class implements a given interface*.
InPackage	checks if the class or interface is in a given package*.
Name	checks if the class's or interface's `name` attribute matches the given parameter*.

Table 5.1: Prototype's supported list of criteria.

children with the limitation that they cannot contain themselves or an ancestor as a child, preventing endless recursion.

5.4 Performance

As mentioned in Section 3.4.3, increasing the performance of the prototype can be achieved in different ways. However, improving the base calculation equations is out of scope of this thesis, therefore we will focus on other techniques to increase the performance. The prototype currently implements two such techniques. The first is the use of quadtrees, which should decrease the number of needed calculations. The second is parallel computation within the same iteration.

5.4.1 Quadtrees

The quadtree technique, as described in Section 3.4.3, had to be adjusted for the prototype because the algorithm uses two dimensional shapes instead of points, as mentioned in Section 5.2. However, to efficiently use the quadtree approach, each object has to be assigned to exactly one quadtree. Therefore, the adaption still took the point of mass of each node (whether data node or structured element) as decision. Thus, the creation of a quadtree structure is the same as usual. However, the adapted version will return a different value when asked of its height or width. While adding objects, each quadtree calculates a bounding box of its elements, whether the quadtree stores the element itself or passes it to one of its children.

This adaption makes it possible to add structured elements to the quadtree, where the structured element might overlap into all four quadtrees. The further calculation using the Barnes-Hut opening criteria stays the same, when using the adapted width and height of the quadtree.

5.4.2 Parallelization

In general implementations of the force directed layout split the calculation into three major steps: Calculating the repulsive and the attractive forces and moving all nodes. Some implementations move the nodes as soon as a calculation of repulsion or attraction is computed. However, this technique is not suitable for parallelism due to race conditions, because the prototype needed some additional steps for cleaning utility lists or performing further operations after the move step. Parallelization is achieved by returning a list of lists of `Callable` objects. The elements of each nested list will be executed in parallel (by using the `ExecutionService.invokeAll` method), but are joined again before progressing to the next element of the outer list. These parallel barriers give a decent control over the parallel execution without the need of explicit synchronizations. For controlling the amount of working threads, the prototype uses the `FixedThreadPool` implementation of the `ExecutorService`.

The following list shows the steps that are executed in parallel, including the big O notation about the iterations, where n is the amount of data nodes and e the amount of edges. The horizontal lines between the items are representing the aforementioned parallel barriers.

repulse* Calculates the repulsive forces. Stores the result with `node.movement.addRepulsiveForce(force).` $O(n^2)$, with quadtrees $O(n * log(n))$

pull* Calculates the attracting forces. Stores the result with `node.movement.addPullForce(force).` $O(e)$

centricGravity* Calculates the centric gravity. Stores the result with `node.movement.addCentricForce(force).` $O(n)$

---

calculateMaxForce Accumulates the three lists from the previous steps, searches for the maximum force of all nodes and calculates the current energy of the system (needed later for adaptive cooling). $O(n)$

---

move* Applies the forces to each node. $O(n)$

---

postAlgorithm Performs the adaptive cooling as described in Section 3.4. $O(1)$

All steps marked with an * may return a list of callable objects as the working space may be split into multiple chunks of configurable chunk sizes. Due to race conditions, it is very important that the three main steps (`repulse`, `pull` and `centricGravity`) store their result in different variables inside the `node.movement`. This separation is enough for `repulse` and `centricGravity` but not for `pull`. The problem with the latter is, that distinct edges may point to the same node. Thus, it still may happen that parallel threads try to update the same `node.movement.pullForces` variable. To solve this problem, the prototype extended the `node.movement` with an additional `ThreadLocal<ForceVector>` and a list of pull force vectors. At the first access of the `ThreadLocal` each thread creates a new force vector, storing it in the `ThreadLocal` and adds the created force vector to the list of pull force vectors. This prevents race conditions on updates on pull forces and also supports the accumulation of those without the need to access all threads after they finished (which might be tricky using `ExecutionService` where we do not have direct access to those threads).

The two methods, `calculateMaxForce` and `postAlgorithm` do not return a list of `Callables`, thus have to strictly be calculated in a single thread. The `postAlgorithm` has a constant speed of $O(1)$, thus it does not need parallelization. The former method on the other hand could be parallelized with a *Map-Reduce* approach. However, such an approach is out of scope of this thesis.

CHAPTER 6

Evaluation

The evaluation is split into four parts. First, we will show the prototype with some screenshots where the prototype visualizes its own code base. Further, we will take a look at the overall performance of the prototype's implementation. We will also discuss optimizations and the influence on the overall performance. Next, we will discuss the prototype's limitations and finally future work which can enhance the current implementation.

6.1 The Prototype

Figure 6.1 shows a screenshot of the prototype, displaying 247 nodes (238 data nodes, 9 structured elements) and 852 edges representing its own code base. The big yellow box in the background is the root structured element. This root structured element has beside the cyan, red, light blue and the pink structured elements (latter is quite small, in the center of other three) also some data nodes as direct children. Every structured element corresponds to a (sub-) system of the prototype, for example the cyan on the top left contains classes responsible for the graphical user interface (labeled "GUI", as visible in the tree structure on the left). The small pink structured element in the center is a separate structured element for the "Controller", which consists of only one class with the same name and purpose.

The screenshot represents the overview of the overall system. Although it is almost impossible to read the labels due to the zoom-out effect, the overall connectivity of the components is clearly visible. For example, we can easily see that the controller class is linked in the whole system, especially strong with the "GUI" structured element. If we zoom in further, we can see that most of the edges are going towards the controller, meaning that many of the other classes have references to the controller, whereas the controller itself has just a few references to other classes.

Classes without inter-component connections are also influenced by the force directed layout. As they do not have a force pulling them to the border of their structured

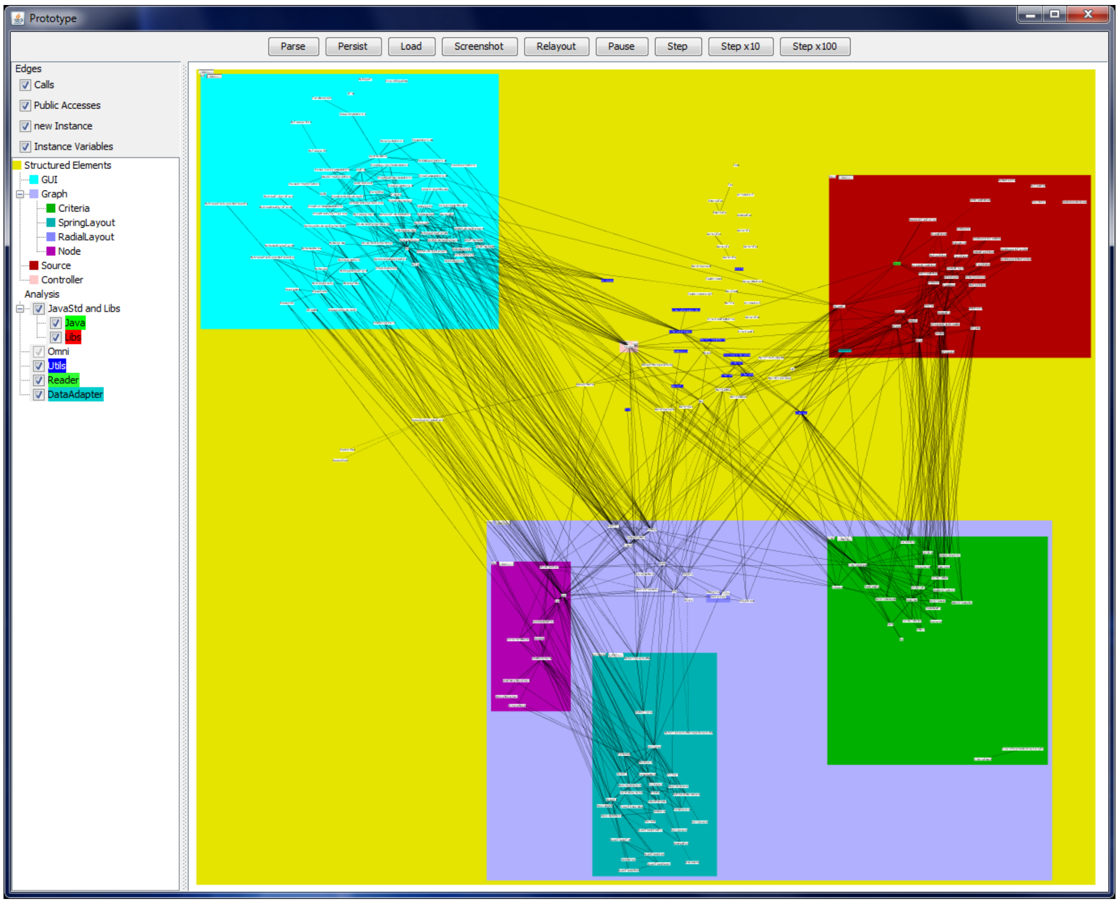


Figure 6.1: Protoype showing the overview of its own code base

elements, they get pushed further inside their structured element, or, if possible, towards the outside of the whole overview. Further, we can see some classes at the top center completely decoupled from the rest of the system. In fact, that are classes which were used in the early stages for testing some interesting bytecode.

The prototype is interactive, allowing the user to move all structured elements and data nodes and to add, edit[1] and remove[2] structured elements and categories at run-time. When changing properties of structured elements or categories the positions of the nodes are kept after anything changes. The prototype re-calculates the layout with the current positions as starting position, as some nodes might got added or removed, changing the available space for the other nodes. This helps the algorithm to stay as close to the current stable overview as possible.

We can, for example, extend the "Source" structured element in Figure 6.1 by

[1]Editing the root structured element and the *omni* category is limited to the color

[2]Root structured element and *omni* category cannot be removed

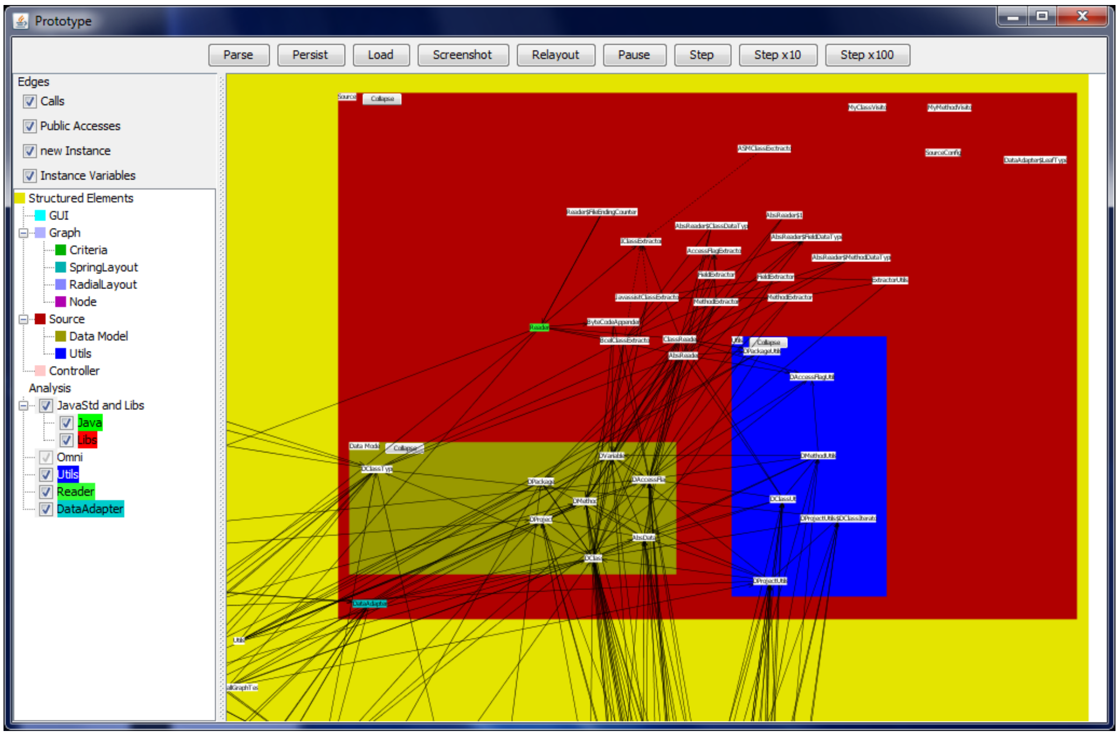


Figure 6.2: "Source" structured element extended by two new nested structured elements

two nested structured elements, representing the core data structures (as described in Section 3.2) and utility classes used by the classes in the "Source" structured element. Figure 6.2 shows the result, where we can see that most of the relations to the "Source" component actually access either the "Data Model" (dark gold) or its "Utils" (dark blue) structured elements. The "Source" component has only the green highlighted `Reader` (extracting information from byte- or source-code) and the teal highlighted `DataAdapter` (converting the data model into a graph) classes communicating with the rest of the system.

In the top left corner of Figures 6.1 to 6.3 we have some configuration methods controlling which edge types should be visible or not. Below them we have a tree structure showing the structured elements as well as the filter sets and categories. Structured elements show both, their names as well as their background color represented by an icon. Starting with the "Analysis" element (which is a bare separator element itself) first the filter sets are listed followed by the categories. The checkbox at each category is a convenient shortcut for enabling or disabling the specific category. Filter sets are only containers allowing us to enable or disable a set of categories at once.

As mentioned in Section 5.3.2, categories have the "Remove matching Nodes" property. The two categories "Java" and "Libs" in Figure 6.2 have criteria expressions matching all classes which are contained in the Java standard libraries or in any third party libraries

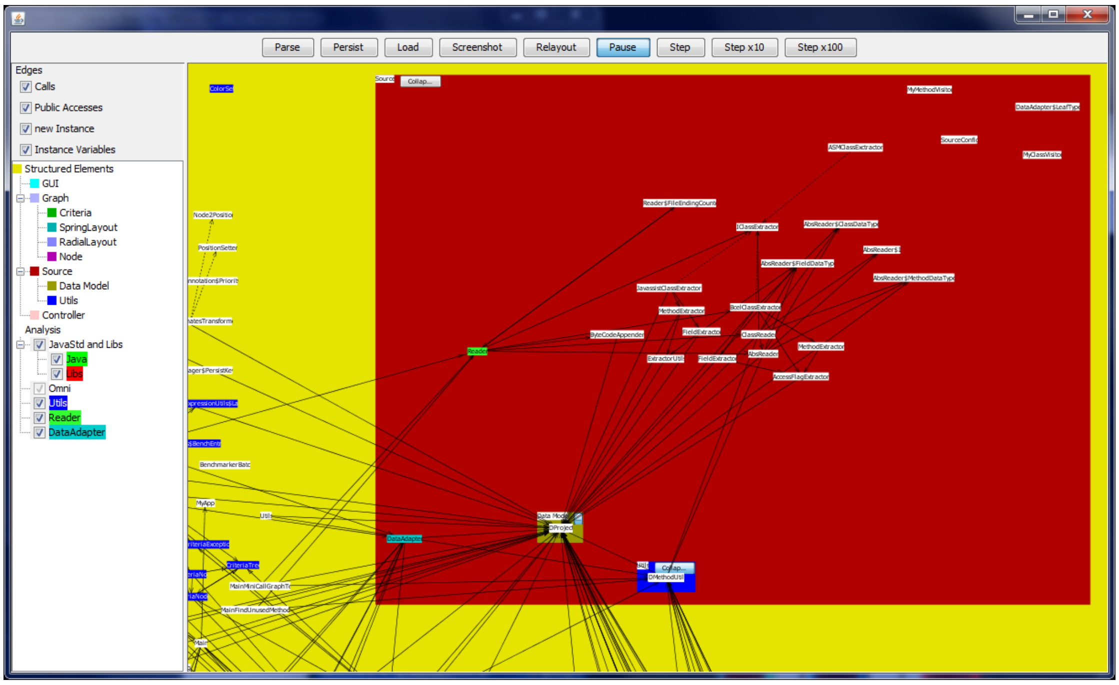


Figure 6.3: The nested structured elements of "Source" collapsed for increased visiblity of the remaining elements

respectively. Both categories have the "Remove matching Nodes" property enabled, leaving only project specific data nodes in the overview.

Figure 6.3 demonstrates the collapse feature of structured elements. By collapsing the "Data Model" and "Utils" structured elements, our previous statement that the "Source" component communicates only through the highlighted structured elements and nodes with the rest of the system, gets even more visible.

Generally speaking the prototype gives an overview about the whole project or a well defined part of it. The structured elements as well as the data nodes can be configured with colors (latter using categories), the user can use his / her own color code for each sub-system. The positioning of each node is influenced by the number of contained children, the margin of structured elements grows with their own visual size. This makes the border even clearer between sub-systems.

Experience during development has shown, that the prototype does provide a convenient overview. Further, it reacts to every structural change with a newly calculated layout which tries to stay as close to the original layout as possible. Moreover, as the prototype animates the steps of the recalculation, the user recognizes the new layout much faster as he / she literally can see where the nodes are moving.

Structured Elements \ Quadtrees	Disabled	Enabled
Disabled	100%	196%
Enabled	259%	271%

Table 6.1: Improvement of runtime performance of structured elements and quadtrees with 1 working thread and 1024 chunk size.

6.2 Performance Tests

All performance tests were executed 100 times each. The corresponding tables are based on the median values of those 100 runs. All tests were executed on the following machine:

- AMD Phenom II Mobile, 4x2.0 GHz
- 8GB DDR3
- Windows 7 64Bit Professional SP1
- JRE 1.8.0 u25 (although the prototype runs Java 1.7)

As stated in Section 5.4, the prototype implemented two optimizations, which we are now evaluating. The overall performance is influenced by two factors, regardless of active optimizations. Those are the time needed to calculate one step of the force directed algorithm, and the amount of iterations. The performance per step can be strongly optimized with various techniques, as for example graph coarsening or quadtrees, whereas the former technique was implemented using structured elements. However, during the development and the final performance tests, the second factor varied about several thousand iterations, usually about two or three thousand.

6.2.1 Quadtrees

The quadtree optimization was tested with a single working thread and a chunk size large enough to take all calculations in single chunks, thus minimizing the overhead. In Section 3.4.3 we stated that quadtrees can reduce the repulsive calculations from $O(n^2)$ to $O(n * log(n))$. Table 6.1 shows, that with only the quadtree optimization active, the iteration performance was about twice as fast as without any optimizations. This performance is far away from $O(n * log(n))$, but we have to consider two additional factors. First, the quadtree has to be build from scratch in each iteration. Second, quadtrees only boost the performance of the repulsive force and neither of the pull force nor the other calculations.

However, enabling the structured elements proved to improve the performance nonetheless. The reason for this is the graph coarsening effect. Instead of potentially calculating all n^2 repulsions, we can repulse a data node with a non parental structured element immediately, without looking at quadtrees if the other node is far away from us. Also,

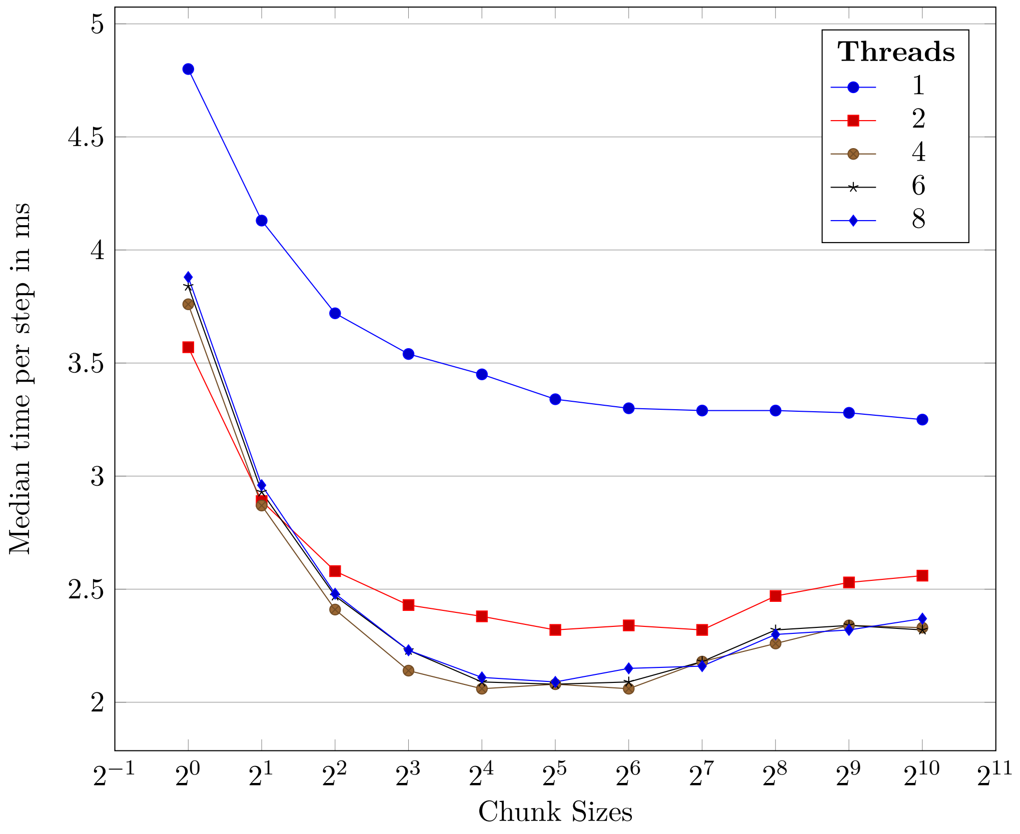


Figure 6.4: Benchmark results of different working threads and chunk-sizes. Each data point is a median time of 100 test runs.

we do not have to build the structured elements from scratch each iteration. The overall performance gain of this optimization depends on the amount of used structured elements. We believe, that the better the user configures the structured elements representing sub-systems, the higher the performance gain will be.

The combination of the two optimizations also boosts the overall performance slightly. The effect is quite small, which is understandable as the quadtree's effectiveness is based on the size of the prepared graph. By assigning the nodes in structured elements, and applying the quadtrees on the direct children of each structured element, the performance boost of the quadtree is reduced.

6.2.2 Multithreading

The executed performance tests were run with 247 nodes (238 data nodes, 9 partially nested structured elements) and 852 edges without the quadtree optimization. Chunk sizes higher than 1024 would have the same result as 1024 as every node and every edge will be assigned to the same array.

Figure 6.4 shows the results of the executed performance tests. We see that low chunk sizes always perform badly. The reason for this is the huge overhead, assigning a thread just a few iterations of the task at once. When the chunk size gets too large, the overall task cannot be dispatched to all available threads equally, resulting in idle threads lowering the overall performance. As expected, the single threaded version performs the best, when the chunk size is large enough to capsule all needed calculation in one chunk per task.

In this tests, we did not use the quadtree optimization. The reason is the splitting the calculations into chunk sizes. The worst case scenario would use chunk sizes of 1, and multiple working threads. Chunk size 1 means, that the thread gets one data node to repulse against every other node in the current structured element. Thus, the per iteration created quadtrees are only used for a single data node. However, efficiently solving this problem is out of scope of this thesis.

6.3 Limitations

We find that the prototype, as a proof of concept tool, has answered the core question of this thesis, whether a tool can identify, with the help of the user, (currently) less important classes and hide or abstract them. However, there are still some limitations that the prototype did not take into account.

6.3.1 Coupling of Classes

We stated in Section 3.4 that an intuitive overview would position classes with high coupling close to each other. Although the force directed layout tries to pull those nodes together, the current implementation of the prototype does not have edge weights, for example based on a quantification of coupling. Thus, two classes will get positioned as close to each other, whether one of them calls the other only with a single statement, or both calling each other eagerly.

6.3.2 Instability

Although the adaptive cooling technique (see Section 3.4) was implemented by the prototype, there is still a chance for the graph to enter a state which can never be stabilized. We believe this is partially caused by the additional centric gravity. However, this happens only when many data nodes are in the same structured elements on the same level (i.e. too many siblings). This may cause a small set of data nodes (or multiple sets) to steadily rotate around their common center. The adaptive cooling only keeps track of the change in energy, which never reaches the state when the system should start cooling.

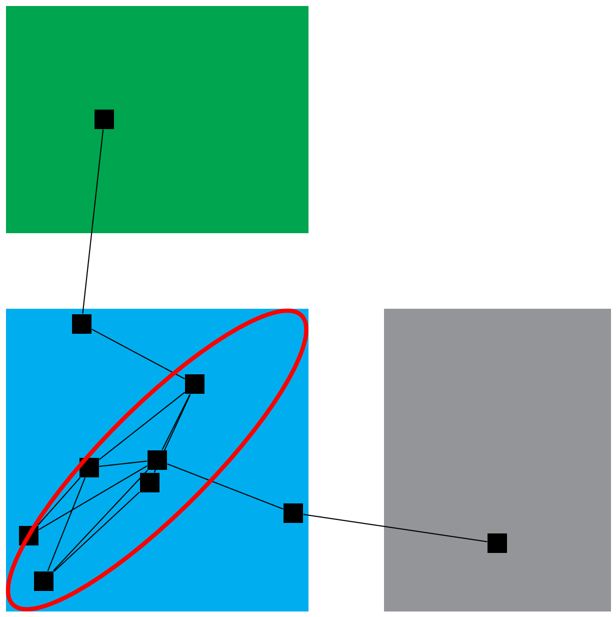


Figure 6.5: Unused reserved space in structured elements

6.4 Future Work

In this section we give an overview about possible future work, starting with general topics and heading to implementation improvements.

Displaying Methods and Attributes Inspired by *Class blueprint* [13], the prototype could provide an option to display a more detailed layer of the represented code. Each class-node could be replaced by a structured element (or a little more restricted version of it) and the actual connecting nodes would be methods and attributes instead of classes. This surely has the disadvantage of a highly increasing amount of nodes, thus not just performance will suffer but also clarity. However, this technique could boost the deeper understanding of the class without needing to read the actual code.

Data Flow Analysis Following a given data flow is a very powerful analysis technique. Call graphs provide an overview over the possible flow of a call, but they do not distinguish if a given data chunk is passed to the called graph or not. For example, for security reasons a developer of a system wants to follow all string based input variables the user can enter. A prime example could be to find possible attack vectors for malicious input, for example, with the purpose of SQL Injection. If the tool could highlight all classes and methods which are in contact with such untrusted data, the developer can focus on improving the security of those classes and methods.

Use Reserved Space for Structured Elements A data node gets repulsed from all its sibling data nodes and sibling structured elements, as well as from its parent's sibling structured elements (excluding the contained data nodes), the grand parent structured elements, and so on. However, in Figure 6.5 we see three structured elements. The cyan (bottom left) structured element's data nodes are all pushed inside the red ellipse, except the data nodes with connections to other structured elements. On the one hand, it is a nice side effect that only the latter type of data nodes are at the structured element's border, possibly representing interfaces or communication classes, while the rest of the classes are used for internal purposes. The reason for this inhomogeneous positioning is that all data nodes repulse each other and the other structured elements (green and gray).

The mostly unused space rapidly grows when the number of data nodes in the cyan structured element is increased and the structured element grows. The red ellipse will maintain its shape and scale with the growth of the structured element.

This problem can probably be faced by adapting the repulsive force in a two step calculation. First, we calculate the data node's position without the repulsive forces of the structured elements outside of its parent. If the first step's calculated position of the node is inside of the parent's bounds, we do not need any further calculation. Otherwise, we recalculate the positioning, but this time we include the previously skipped repulsive forces. This would allow a node to extend its parent's border when the second step also results outside of the parent.

The previously desired effect that the algorithm positions the interfaces or communicating classes close to the border can be ensured by limiting the boundary of the first step tighter than the structured element's bounds.

Structured Elements and Quadtrees We discussed the implementation problems of combining these two techniques in Section 6.2.2, yet we believe that the problems which were discussed are efficiently solvable. However, when combining structured elements with quadtrees, structured elements containing too many data nodes, which would have to calculate n^2 repulsions would perform much better due to the quadtree. Thus, the user would get the best performance whether they configure the structured elements in a balanced fashion or not.

Coupling Influenced Edge Weights Quantification of coupling between two classes is not an easy task. Yet, if the tool could provide such a calculation the edge weights could easily be adapted for the force directed layout, improving the overall layout.

IDE integration Modern IDEs like Eclipse, IntelliJ's IDEA, NetBeans and others are often used by developers for writing software projects. Connecting the tool with such an IDE could improve the efficiency of the developers. Example usages are highlighting the data nodes of the currently opened class(es) and their direct neighbor data nodes. Selecting an edge or data node in the overview could, for example highlight the corresponding source code.

Combining Compiled and Source Code Compiled code does not contain all the information the source code provides, and vice versa. For example, in Java, if the code is not compiled with specific flags, the parameter names are not extractable from the compiled code. Comments (including “Todo”-notes, information about the author, Javadoc, et cetera), default values, some annotations, and so on, are also not stored in the bytecode. When combining information from the compiled code and the source code (if available), such information can be recovered. As the tool itself is supposed to give an overview about a system which is currently in development, the assumption that the source code of at least the current project’s workspace is available, is reasonable.

Attribute: Parsed from File The file from which the `DClass` got its data could be also stored in `DClass`. The file can be a compiled file (i.e. “DClass.class”) or it could be inside a container file (i.e. “library.jar”). In the latter case, this information can be used with additional criteria for detailed categorization of the used libraries.

Automatically Suggest new Structured Element Structured elements represent a sub-system. These sub-systems are often also encapsulated in the structure of the source code as well. In Java, we know this structure as packages, which are directly mapped to the folder structure on the file system. Suggesting structured elements is easily possible with the *InPackage* criteria, where the user only has to select the listed packages.

Another approach is to let the user select arbitrary data nodes. If the selected data nodes are in the same structured element, the tool can calculate a criteria expression matching those data nodes, encapsulating them in a nested structure element. An option for this capability could be to either restrict the structured element only for those data nodes (minimal matching), or allow similar, not yet added elements (maximal matching).

In graph theory, we know the concept of strongly connected components. Applying this approach to our class overview, the tool could make suggestions for such strongly connected components. The user has only to verify if the suggestion really represents one of the sub-systems.

CHAPTER 7

Conclusion

While developing a software system, overview is always important. This thesis provides a prototype, which is capable of providing such an overview. The thesis is split into several parts. After discussing related topics, the theory behind the prototype's approach is explained in detail. The following chapters describe development and implementation of the prototype itself respectively. Finally, the evaluation chapter shows the outcome of the generated visualization, provides benchmarks as well as an overview of the current limitations, and describes possible future research topics.

7.1 Theory

The prototype is mainly based on three important concepts. The process starts with gathering information about the given project by reverse engineering the project's bytecode. Parsing the source code proved to have potential ambiguity, which is why the prototype indeed supports it, but by default uses bytecode parsing. As the prototype's data model already represents the information of classes and relations, a conversion into a graph is straight forward.

The core concept is the positioning of the single data nodes. For this a modified force directed algorithm is used. The approach taken considers the nodes as equally charged electrical particles, which are pushing each other away. The relations are acting as springs, pulling connected nodes together. With this basic idea a layout with very few edge crossings (if possible) can be calculated, which intrinsically takes the connectivity of the data nodes into account. This alone already provides a convenient visualization of classes and their relations as communicating or related classes are close to each other.

As in this iterative algorithm every node has to be calculated against every other node, the repulsive forces alone need n^2 calculations. Based on the quadtree optimization, the number of these calculations can be lowered to $O(n * log(n))$. To preventing the algorithm to run infinitely, an adaptive cooling scheme was implemented, based on Hu's work [60].

7.2 Implementation

One of the major adaptions to the presented techniques the prototype made was on the force directed layout. The positioning algorithm had to be modified in order to prevent overlapping. The prototype calculates the rectangular distance instead of relying on centric distances. When two nodes get visually too close to each other, the repulsive force simply overwhelms the attractive force and thus prevents overlapping.

This modification works on all rectangular nodes, regardless of their height and width. This allows the prototype to introduce structured elements, that are recursive visual and logical containers for data nodes. As the weight of every node is based on its children, well filled structured elements have a large margin to their siblings while simple data nodes will stay relatively close to each other. The user defines a criteria expression (which may contain only a single criteria) in each structured element describing which data nodes should be contained or not.

The same criteria expressions can also be used for categories to highlight and hide data nodes. Highlighting is done by letting the user define the fore- and background color of the matching data nodes. When a category has the "Remove matching nodes" option enabled, the affected data nodes are temporarily removed from the graph, which also increases the runtime performance. The latter feature allows the user to define his / her current focus.

The chapter also describes parallelization techniques implemented in the prototype. These techniques give the layouting algorithm the capability to control the parallelization in a very fine granularity, using implicit parallel barriers. The prototype also provides iteration control over the iterative algorithm by pausing and single-stepping through the algorithm.

Based on experience during the development, the prototype also modularized the parsing and the layouting subsystems. The atomic criteria are also easily extendable, which makes the prototype very flexible and dynamic.

7.3 Evaluation

The executed performance benchmarks have shown, that the prototype's concept of structured elements greatly boosts the overall runtime performance. The iterative algorithm is animated, so that after a few seconds the user already sees a rough positioning of the system. Furthermore, the collapse and expand capabilities of each structured element also help the user's clarity and support him / her specifying the current focus.

To summarize, the prototype fulfills all specified requirements. The interactive nature of the prototype supports the user at any time with flexible focusing and analyzing the given project based on structured elements and categories through criteria expressions.

APPENDIX A

Abstract Syntax Tree Parser Example

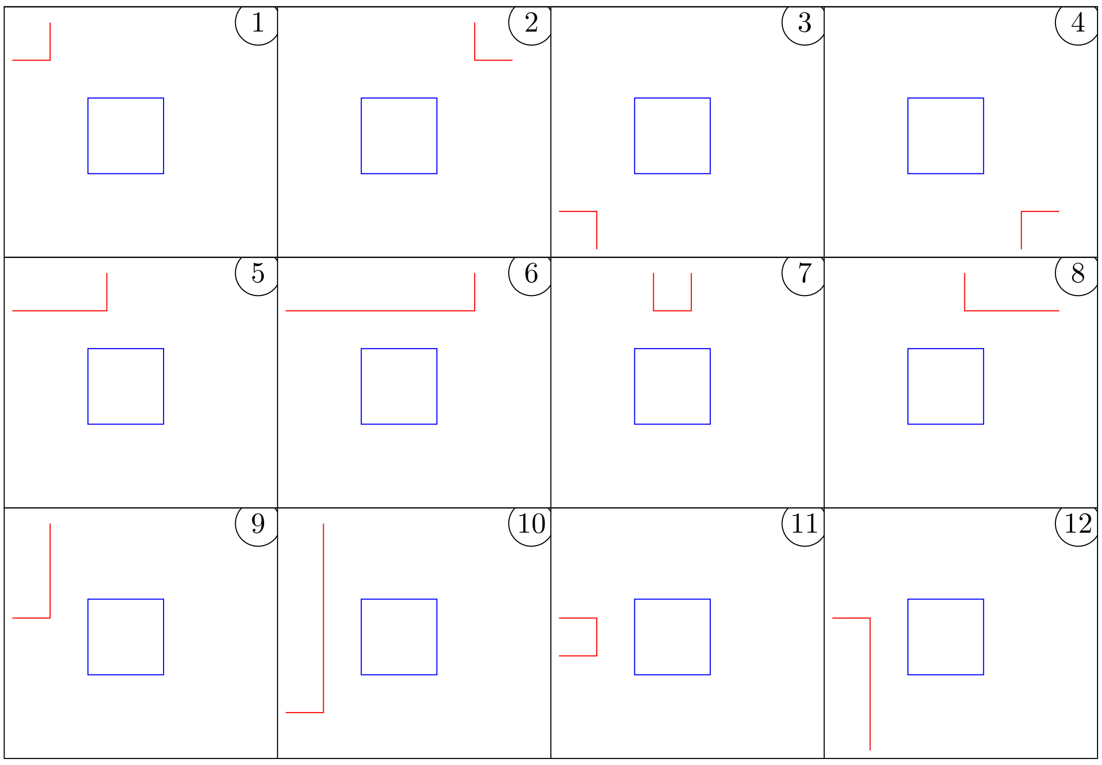
```java
String src = ... // get the source code

// configures the ASTParser for J2SE 7 or below
ASTParser parser = ASTParser.newParser(AST.JLS4);
parser.setSource(src.toCharArray());
parser.setKind(ASTParser.K_COMPILATION_UNIT);

final CompilationUnit cu = (CompilationUnit) parser.createAST(
   (IProgressMonitor) null);

cu.accept(new ASTVisitor() {

   @Override
   public boolean visit(VariableDeclarationFragment node) {
      SimpleName name = node.getName();
      this.names.add(name.getIdentifier());
      System.out.println("Declaration of '" + name + "' at line "
                  + cu.getLineNumber(name.getStartPosition()));
      return true; // continue with next element
   }

   @Override
   public boolean visit(MethodDeclaration node) {
      String name = node.getName().getIdentifier();

      List parameters = node.parameters();
      for (Object parameter : parameters) {
         if (parameter instanceof SingleVariableDeclaration) {
            SingleVariableDeclaration var = (SingleVariableDeclaration)
               parameter;

```

```java
                String varName = var.getName().getIdentifier();
                String simpleType = this.getSimpleType(var.getType());
            }
        }
        return true;
    }

    private String getSimpleType(Type type) {
        if (type instanceof SimpleType) {
            return ((SimpleType) type).getName().getFullyQualifiedName();
        }
        if (type instanceof PrimitiveType) {
            Code primitiveTypeCode = ((PrimitiveType) type).getPrimitiveTypeCode();
            return primitiveTypeCode.toString();
        }

        /*
            ...
            further "type instanceof ..." cases
        */
        throw new UnknownTypeException(type);
    }
});
```

Listing A.1: ASTParser Example

APPENDIX B

Rectangle Distance Cases



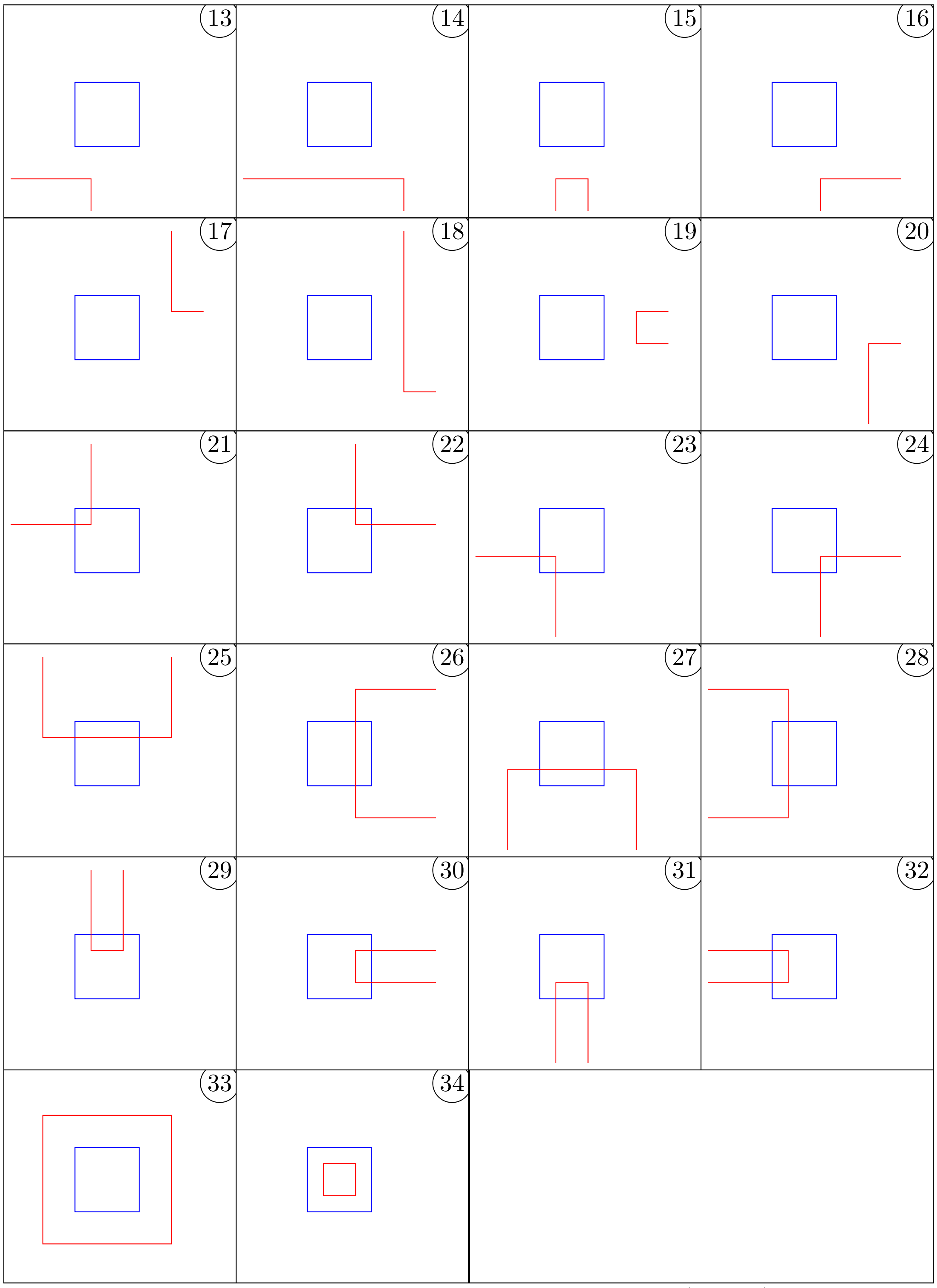


Table B.1: All cases of two rectangle locations with possible (partial) overlapping

```java
public Tuple<Float, Direction> getRectangleDistance(
            float x1, float y1, float w1, float h1,
            float x2, float y2, float w2, float h2) {
    float left = x1 - (x2 + w2);
    float top = y1 - (y2 + h2);
    float right = x2 - (x1 + w1);
    float bottom = y2 - (y1 + h1);

    int direction = left >= 0 ? 1 : 0;
    direction += top >= 0 ? 2 : 0;
    direction += right >= 0 ? 4 : 0;
    direction += bottom >= 0 ? 8 : 0;
    switch(direction){
        case 1: return new Tuple<>(left, WEST); // r-cases 9,10,11,12
        case 2: return new Tuple<>(top, NORTH); // r-cases 5,6,7,8
        case 3: // r-case 1
            return new Tuple<>(sqrtDist(left, top), NORTHWEST);
        case 4: return new Tuple<>(right, EAST); // r-cases 17,18,19,20
        case 6: // r-case 2
            return new Tuple<>(sqrtDist(right, top), NORTHEAST);
        case 8: return new Tuple<>(bottom, SOUTH); // r-cases 13,14,15,16
        case 9: // r-case 3
            return new Tuple<>(sqrtDist(left, bottom), SOUTHWEST);
        case 12: // r-case 4
            return new Tuple<>(sqrtDist(right, bottom), SOUTHEAST);
        case 0: // overlapping, r-cases 21-34
            float closest;
            Direction dir;
            if(left > top){
                closest = left;
                dir = WEST;
            } else {
                closest = top;
                dir = NORTH;
            }
            if(closest < bottom){
                closest = bottom;
                dir = SOUTH;
            }
            if(closest < right){
                closest = right;
                dir = EAST;
            }
            return new Tuple<>(closest, dir);
// switch-cases 5,7,10,11,13,14 and 15 are mathematically not possible
    }
}
public float sqrtDist(float dx, float dy){
    return sqrt(dx*dx + dy*dy);
}
```

Listing B.1: Rectangular distance calculation algorithm. "r-cases" refer to Table B.1

Bibliography

[1] Irving Biederman. “Recognition-by-components: a theory of human image understanding.” In: *Psychological review* 94 (1987), pp. 115–147. ISSN: 0033-295X. DOI: `10.1037/0033-295X.94.2.115`.

[2] D Marr. “Vision: A Computational Investigation into the Human Representation and Processing of Visual Information”. In: *Phenomenology and the Cognitive Sciences* 8 (1982), p. 397. ISSN: 15687759. DOI: `10.1007/s11097-009-9141-7`.

[3] I Spence. “Visual psychophysics of simple graphical elements.” In: *Journal of experimental psychology. Human perception and performance* 16 (1990), pp. 683–692. ISSN: 0096-1523. DOI: `10.1037/0096-1523.16.4.683`.

[4] OMG. *UML 2.4.1-Infrastructure Specification.* Tech. rep. 2011.

[5] Pierre Caserta and Olivier Zendra. “Visualization of the static aspects of software: A survey”. In: *IEEE Transactions on Visualization and Computer Graphics* 17 (2011), pp. 913–933. ISSN: 10772626. DOI: `10.1109/TVCG.2010.110`.

[6] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. “Seesoft–A tool for visualizing line oriented software statistics”. In: *IEEE Transactions on Software Engineering* 18 (1992), pp. 957–968. ISSN: 00985589. DOI: `10.1109/32.177365`.

[7] A Marcus, L Feng, and J I Maletic. “Comprehension of software analysis data using 3D visualization”. In: *Program Comprehension, 2003. 11th IEEE International Workshop on.* 2003, pp. 105–114. DOI: `10.1109/WPC.2003.1199194`.

[8] J.I. Maletic, a. Marcus, and L. Feng. “Source Viewer 3D (sv3D) - a framework for software visualization”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 6 (2003), pp. 812–813. DOI: `10.1109/ICSE.2003.1201299`.

[9] J.a. Jones, M.J. Harrold, and J.T. Stasko. “Visualization for fault localization”. In: *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada* (2001), pp. 71–75.

[10] SP Reiss. “Bee/hive: A software visualization back end”. In: *IEEE Workshop on Software Visualization* (2001), pp. 1–5.

[11] WG Griswold, JJ Yuan, and Y Kato. “Exploiting the map metaphor in a tool for software evolution”. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001* March (2001).

[12] T Ball and SG Eick. “Software visualization in the large”. In: *Computer* 29.4 (1996), pp. 33–43.

[13] Michele Lanza. “Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization”. PhD thesis. 2003.

[14] Stéphane Ducasse and Michele Lanza. “The class blueprint: Visually supporting the understanding of classes”. In: *IEEE Transactions on Software Engineering* 31 (2005), pp. 75–90. ISSN: 00985589. DOI: 10.1109/TSE.2005.14.

[15] C. Wetherell and a. Shannon. “Tidy Drawings of Trees”. In: *IEEE Transactions on Software Engineering* SE-5.5 (Sept. 1979), pp. 514–520. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234212.

[16] B. Johnson and B. Shneiderman. “Tree-maps: a space-filling approach to the visualization of hierarchical information structures”. In: *Proceeding Visualization ’91* (1991). DOI: 10.1109/VISUAL.1991.175815.

[17] J Lamping and R Rao. “The hyperbolic browser: A focus+ context technique for visualizing large hierarchies”. In: *none* (1996), pp. 33–55.

[18] J. Stasko and E. Zhang. “Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations”. In: *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings* (2000). ISSN: 1522-404X. DOI: 10.1109/INFVIS.2000.885091.

[19] Margaret-anne Storey, C Best, and J Michand. “Shrimp views: An interactive environment for exploring java programs”. In: *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on* (2001), pp. 1–4. DOI: http://doi.ieeecomputersociety.org/10.1109/WPC.2001.921719.

[20] GW Furnas. “Generalized fisheye views”. In: *Proceedings of ACM CHI’86* April (1986), pp. 16–23.

[21] M.-A.D. Storey and H.A. Muller. “Manipulating and documenting software structures using SHriMP views”. In: *Proceedings of International Conference on Software Maintenance* (1995). ISSN: 1063-6773. DOI: 10.1109/ICSM.1995.526549.

[22] M. A D Storey, K. Wong, and H. A. Müller. “How do program understanding tools affect how programmers understand programs?” In: *Science of Computer Programming* 36 (2000), pp. 183–207. ISSN: 01676423. DOI: 10.1016/S0167-6423(99)00036-2.

[23] Michael Balzer and Oliver Deussen. “Level-of-detail visualization of clustered graph layouts”. In: *Asia-Pacific Symposium on Visualisation 2007, APVIS 2007, Proceedings* (2007), pp. 133–140. DOI: 10.1109/APVIS.2007.329288.

[24] Michael Balzer and Oliver Deussen. “Hierarchy Based 3D Visualization of Large Software Structures”. In: *Proceeding VIS ’04 Proceedings of the conference on Visualization ’04*. 2004, p. 598.4. ISBN: 0-7803-8788-0. DOI: 10.1109/VISUAL.2004.39.

[25] Danny Holten. “Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006).

[26] Irving Biederman et al. “GEON THEORY AS AN ACCOUNT OF SHAPE RECOGNITION IN MIND , BRAIN , AND MACHINE”. In: *Bmvc* (1993), pp. 175–186. DOI: 10.5244/C.7.18.

[27] Ken Casey and Chris Exton. “A Java 3D implementation of a geon based visualisation tool for UML”. In: *PPPJ ’03 Proceedings of the 2nd international conference on Principles and practice of programming in Java* c (2003), pp. 63–65.

[28] J.Y. Gill and S. Kent. “Three dimensional software modelling”. In: *Proceedings of the 20th International Conference on Software Engineering* (1998). ISSN: 0270-5257. DOI: 10.1109/ICSE.1998.671107.

[29] Martin Gogolla, Oliver Radfelder, and Mark Richters. “Towards three-dimensional representation and animation of UML diagrams”. In: *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard.* 1999, pp. 489–502. ISBN: 3-540-66712-1.

[30] Oliver Radfelder and Martin Gogolla. “On better understanding UML diagrams through interactive three-dimensional visualization and animation”. In: *Proceedings of the working conference on Advanced visual interfaces - AVI ’00.* 2000, pp. 292–295. ISBN: 1581132522. DOI: 10.1145/345513.345358.

[31] Carsten Gutwenger et al. “A new approach for visualizing UML class diagrams”. In: *Proceedings of the 2003 ACM symposium on Software visualization.* 2003, pp. 179–188. ISBN: 1-58113-642-0. DOI: 10.1145/774833.774859.

[32] H.C. Purchase, J-a. Allder, and D. Carrington. “User preference of Graph Layout Aesthetics: a UML study”. In: (2000), pp. 5–18. DOI: 10.1007/3-540-44541-2_2.

[33] A. R. Teyseyre and M. R. Campo. “An overview of 3D software visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), pp. 87–105. ISSN: 10772626. DOI: 10.1109/TVCG.2008.86.

[34] T. Panas, R. Berrigan, and J. Grundy. “A 3D metaphor for software production visualization”. In: *Proceedings on Seventh International Conference on Information Visualization, 2003. IV 2003.* (2003). DOI: 10.1109/IV.2003.1217996.

[35] Thomas Panas et al. “Communicating Software Architecture using a Single-View Visualization”. In: *Babel* (2007), pp. 217–228. DOI: 10.1109/ICECCS.2007.20.

[36] Andreas Dieberger and Andrew U. Frank. “A City Metaphor to Support Navigation in Complex Information Spaces”. In: *Journal of Visual Languages & Computing* 9 (1998), pp. 597–622. ISSN: 1045926X. DOI: 10.1006/jvlc.1998.0100.

[37] Sazzadul Alam and Philippe Dugerdil. “EvoSpaces visualization tool: Exploring software architecture in 3D”. In: *Proceedings - Working Conference on Reverse Engineering, WCRE.* 2007, pp. 269–270. ISBN: 0769530346. DOI: 10.1109/WCRE.2007.26.

[38] Andreas Dieberger. “Navigation in textual virtual environments using a city metaphor”. PhD thesis. Vienna University of Technology, 1994.

[39] Hamish Graham, HY Yang, and Rebecca Berrigan. “A solar system metaphor for 3D visualisation of object oriented software metrics”. In: *APVis ’04 Proceedings of the 2004 Australasian symposium on Information Visualisation - Volume 35* (2004), pp. 53–59.

[40] H Yang and H Graham. “Software Metrics and Visualisation”. In: *Univ. of Auckland, Tech. Rep* (2003).

[41] Orla Greevy, Michele Lanza, and Christoph Wysseier. “Visualizing live software systems in 3D”. In: *SoftVis ’06 Proceedings of the 2006 ACM symposium on Software visualization.* 2006, pp. 47–56. ISBN: 1595934642. DOI: 10.1145/1148493.1148501.

[42] A. R. Teyseyre and M. R. Campo. “An overview of 3D software visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 15.1 (2009), pp. 87–105. ISSN: 10772626. DOI: 10.1109/TVCG.2008.86.

[43] A.J. Hanson, E.A. Wernert, and S.B. Hughes. “Constrained Navigation Environments”. In: *Scientific Visualization Conference (dagstuhl ’97)* (1997).

[44] M. Tory et al. “Visualization task performance with 2D, 3D, and combination displays”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.1 (2006), pp. 2–13. ISSN: 10772626. DOI: 10.1109/TVCG.2006.17.

[45] Maurice Termeer et al. “Visual exploration of combined architectural and metric information”. In: *Proceedings - VISSOFT 2005: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis.* 2005, pp. 21–26. ISBN: 0780395409. DOI: 10.1109/VISSOF.2005.1684298.

[46] Heorhiy Byelas and Alexandru Telea. “Visualizing metrics on areas of interest in software architecture diagrams”. In: *IEEE Pacific Visualization Symposium, PacificVis 2009 - Proceedings.* 2009, pp. 33–40. ISBN: 9781424444045. DOI: 10.1109/PACIFICVIS.2009.4906835.

[47] Richard Wettel and Michele Lanza. “Visually localizing design problems with disharmony maps”. In: *Proceedings of the 4th ACM symposium on Software visualization.* 2008, pp. 155–164. ISBN: 978-1-60558-112-5. DOI: 10.1145/1409720.1409745.

[48] Shyam R. Chidamber and Chris F. Kemerer. “Metrics suite for object oriented design”. In: *IEEE Transactions on Software Engineering* 20 (1994), pp. 476–493. ISSN: 00985589. DOI: 10.1109/32.295895.

[49] T.J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[50] L.C. Briand, J.W. Daly, and J. Wust. “A unified framework for cohesion measurement in object-oriented systems”. In: *Proceedings Fourth International Software Metrics Symposium* (1997), pp. 43–53. ISSN: 13823256. DOI: 10.1109/METRIC.1997.637164.

[51] James M. Bieman and Byung-Kyoo Kang. “Cohesion and reuse in an object-oriented system”. In: *ACM SIGSOFT Software Engineering Notes* 20.SI (1995), pp. 259–262. ISSN: 01635948. DOI: 10.1145/223427.211856.

[52] Alexander Egyed. “Semantic abstraction rules for class diagrams”. In: *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on* (2000), pp. 301–304. DOI: 10.1109/ASE.2000.873683.

[53] Alexander Egyed. “Automated abstraction of class diagrams”. In: *ACM Transactions on Software Engineering and Methodology* 11.4 (Oct. 2002), pp. 449–491. ISSN: 1049331X. DOI: 10.1145/606612.606616.

[54] FD Rácz and Kai Koskimies. “Tool-supported compression of UML class diagrams”. In: *«UML»'99 - The Unified Modeling Language* (1999), pp. 172–187.

[55] Karim Ali and O Lhoták. “Application-only call graph construction”. In: *ECOOP 2012 - Object-Oriented Programming* (2012), pp. 688–712.

[56] David Grove and Craig Chambers. “A framework for call graph construction algorithms”. In: *ACM Transactions on Programming Languages and Systems* 23.6 (Nov. 2001), pp. 685–746. ISSN: 01640925. DOI: 10.1145/506315.506316.

[57] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. “Incremental computation of dominator trees”. In: *ACM Transactions on Programming Languages and Systems* 19.2 (1997), pp. 239–252. ISSN: 01640925. DOI: 10.1145/244795.244799.

[58] G. Ramalingam and T. Reps. “An incremental algorithm for maintaining the dominator tree of a reducible flowgraph”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '94 (1994), pp. 287–296. ISSN: 07308566.

[59] Md Carroll and Bg Ryder. “Incremental data flow analysis via dominator and attribute update”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1988), pp. 274–284. DOI: 10.1145/73560.73584.

[60] Yifan Hu. “Efficient, High-Quality Force-Directed Graph Drawing”. In: *Mathematica Journal* 10 (2005), pp. 37–71. ISSN: 17482631. DOI: 10.3402/qhw.v6i2.5918.

[61] Thomas M. J. Fruchterman and Edward M. Reingold. “Graph Drawing by Force-directed Placement”. In: *Software-Practice and Experience* 21 (1991), pp. 1129–1164. ISSN: 1097-024X. DOI: 10.1002/spe.4380211102. arXiv: 91/111129âĂŞ36 [0038-0644].

[62] Josh Barnes and Piet Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: *Nature* 324 (1986), pp. 446–449. ISSN: 0028-0836. DOI: 10.1038/324446a0.

[63] Ron Davidson and David Harel. “Drawing graphs nicely using simulated annealing”. In: *ACM Transactions on Graphics* 15.4 (1996), pp. 301–331. ISSN: 07300301. DOI: 10.1145/234535.234538.

[64] Chris Walshaw. “A Multilevel Algorithm for Force-Directed Graph-Drawing”. In: *Journal of Graph Algorithms and Applications* 7 (2003), pp. 253–285. ISSN: 3540415548. DOI: 10.7155/jgaa.00070.

Web Links

[W1] *Wikipedia - Unified Modeling Language.* https://en.wikipedia.org/wiki/Unified_Modeling_Language.

[W2] *WALA - Watson Libraries For Analysis.* http://wala.sourceforge.net/.

[W3] *Eclipse.* https://eclipse.org/.

[W4] *Javassist.* http://www.javassist.org/.

[W5] *ASM.* http://asm.ow2.org/.

[W6] *BCEL.* http://commons.apache.org/bcel/.

[W7] *Git.* https://git-scm.com.

[W8] *GitLab.* https://gitlab.com.

[W9] *Redmine.* http://www.redmine.org/.

[W10] *Gephi.* http://gephi.github.io/.

[W11] *D3 - Data-Driven Documents.* http://d3js.org.

[W12] *JUNG - Java Universal Network/Graph Framework.* http://jung.sourceforge.net/.

[W13] *JavaDoc, Eclipse ASTParser.* http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html.