Quicksilver

A Framework for Hierarchical Data Analysis

Master Thesis
at the Faculty of Natural Sciences
University of Bern

submited by

Dennis Schenk

September 2014

Thesis Supervisors:
Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu
Institute of Computer Science and Applied Mathematics

Abstract

Software analysis is an advanced field of study in computer science. It deals with large and complex datasets and has found the abstractions, concepts and techniques to deal with the challenges such data sets present. We see the fields of reverse engineering, architecture recovery and related visualization techniques as an ideal starting point to enable exploration of a wider range of data sets, not just the ones pertaining to software. We motivate and derive a model for a generic kind of data: data that contains entities that are ordered in a hierarchical way and connected through arbitrary relationships. Based on this model we propose a framework which enables exploration and analysis of data that has the mentioned characteristics naturally or to which such characteristics can be introduced. We present a proof-of-concept implementation of such a framework called Quicksilver and validate our approach by presenting two case studies made possible by using the tool.

Contents

List of Figures

List of Tables

Acknowledgements

First of all I would like to thank Dr. Mircea Lungu for his endurance, encouragement and good spirit. Discussing with him was always inspirational and helpful. I would also like to thank Prof. Oscar Nierstraz for giving me the opportunity to work in his group and for his interesting lectures on software composition and analysis. Dr. Tudor Girba got me fascinated on the topic of software evolution with his captivating lectures, thanks for the inspiration and also giving me input about some technical aspects of the thesis. I also thank the members of the Software Composition Group for their valuable comments and suggestions. And last but not least, the people around me for always asking me about the status of my thesis and motivating me, I thank my family and my friends.

Thank you all.

1 Introduction

Software analysis is an advanced field of study in computer science. It deals with large and complex data sets and has found the means, abstractions and concepts to deal with the challenges such data sets present. The tools put forth by the field are, naturally, restricted to software. Often they are only considering source code, but a software system is only fully described by all its constituting and produced artifacts — such as configuration files, build scripts, contents of databases, run-time information, error reports, etc. Consequently there are efforts to create models and tools that try to encompass and use all of these possible sources of information.

We propose to go a step further, to learn from the field, take a little step upwards in the abstraction ladder and generalize to a generic kind of data: data that contains entities that are organized in a hierarchical way and connected through arbitrary relationships. We do not try to find a model for software, or data in general — We propose to find a model and build a framework around it that allows people with particular data sets, which are not confined to software systems, to find answers to specific questions they have. These data sets have the mentioned characteristics naturally, or the characteristics can be introduced manually or automatically.

By looking for similar entities in data sources and recursively grouping them together a hierarchy can be built that helps one to abstract complex and large data. By further looking for relationships between single entities we are able to build a hierarchical graph representing the data. If we then propagate attributes and relationship information from the lower levels of the

hierarchy up the graph, we are able to support data analysis with a divide and conquer technique, that allows one to drill down in abstraction levels from the aggregated to the explicit.

The questions users have about their data will ultimately need to be posed in a formal way, *i.e.* by using a programming language and an API of a framework built upon such a model. Such an approach would allow us to analyze a wide range of arbitrary data.

1.1 Thesis Statement

We argue that by finding a generic model of data, data that is hierarchical and intra-connected through relationships, by building a framework around such a model and by adapting and generalizing proven concepts and techniques from software analysis, we can enable exploration and analysis of a wide range of data while retaining the ability to analyze software.

1.2 Goals

Our goals for this thesis are thus the following:

1. Get an overview of software analysis tools, learn from them and adapt known concepts and take another step in the abstraction ladder towards generic hierarchical data exploration and analysis
2. Identify and motivate a supporting model and theory
3. Present a proof-of-concept implementation
4. Validate our model by providing case studies

1.3 Outline

The remainder of this thesis is structured as follows: In Chapter 2 we look at related approaches for software engineering and compare how our approach is similar to or different from other solutions. In Chapter 3 we present our derivation of a generic model that allows us to abstract hierarchical data and in Chapter 4 we introduce a framework built on the mentioned model to allow for generic hierarchical data analysis. In Chapter 5 we present a proof-of-concept implementation of the framework, explain its architecture and show some examples on how to use the tool. Two case studies will be presented in Chapter 6 and in Chapter 7 we will discuss our findings, conclude and look at future work.

2
Related Work

The work presented in this thesis is related to software and data visualization as well as large scale data analysis. Our starting point to a generic model of hierarchical data analysis and visualization is reverse engineering software architecture recovery and related visualizations. In this thesis many of the concepts of these more specific fields are re-used or adapted and applied to a more generic approach.

We will first give a broad overview over the fields of reverse engineering and architecture recovery, we will look at software visualization and finally at generic software analysis to see how they compare to our approach.

2.1 Reverse Engineering

M.G. Rekoff originally defined reverse engineering in 1985 as *the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system*[45]. Chikofsky and Cross defined reverse engineering based on Rekoff, but related to software in 1990 as *the process of analyzing a subject system to identify the systems components and their relationships, and to create representations of the system in another form or at a higher level of abstraction.*[14]. In 1996 Aiken looked into reverse engineering with focus on system or organizational data, which he called *data reverse engineering* and defined as *the use of structured*

techniques to reconstitute the data assets of an existing system.[3]. In the same year Hainhaut looked into database reverse engineering[22]. Consequently Müller *et al.* proposed code and data reverse engineering as additional focal points for research in computer science for the new millennium in their paper *Reverse Engineering: A Roadmap*[39].

2.2 Architecture Recovery

Architecture recovery is an element of reverse engineering. Understanding the architecture of large software systems is a prerequisite for their maintenance and evolution[30, 43]. Two problems make architecture understanding difficult: architecture is usually not explicitly represented in the code, and architecture is the subject of degradation, drift, and erosion[42]. When drift and erosion have brought the system architecture away from the initial state, the solution is to recover the architecture of the system from the code. Jazayeri *et al.* defined architecture recovery as *the techniques and processes used to uncover a system's architecture from available information*[26].

In large software systems, the architecture is specified through multiple architectural views. An architectural view is a representation of a whole system or of part of a system from the perspective of a related set of concerns. Each architectural view conforms to a viewpoint. A viewpoint is a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis[27].

Although different authors propose different viewpoints [10, 25, 31] there are two points of consensus. First, all the proponents agree that the architecture of a system is complex and multifaceted and that multiple viewpoints are needed to express it. Second, all the authors specify one viewpoint which presents the modules and the relationships between them. Bass and Clemens call this viewpoint the *module viewpoint*. Most of the work in architecture recovery is focused on recovering views that correspond to the module viewport.

2.3 Software Visualization

The first architectural visualization prototype was Rigi[40], a programmable reverse engineering environment which emphasizes visualization and interaction. Rigi can visualize software as hierarchical typed graphs. It used various abstraction mechanism related to software constituents. The reconstruction process is based on a bottom-up process of grouping the software elements into clusters by manually selecting the nodes and collapsing them, *i.e.* the user starts from the lowest-level facts and aggregates them as he climbs up in the abstraction hierarchy. This approach does not scale when analyzing very large systems because the number of low-level artifacts is too large.

Rigi was later enhanced with simple-hierarchical-multi-perspective-views, or shorter: SHriMPs [49]. The idea was to use fisheye techniques first, rather than bottom-up approaches, when presenting software systems. This is the approach we use in our solution as well. The SHriMP approach was later continued as Creole[34], an eclipse plug-in. SHriMP and Creole display architectural diagrams using nested graphs. Their user interface embeds source code inside the graph nodes and integrates a hypertext metaphor for following low-level dependencies with animated panning and zooming. The tools used to be rather slow because the graphs were computed on the fly, this is why our solution is based on a complete pre-computation of the whole hierarchical graph including automatically aggregating low-level relations.

Hy+ is a hygraph-based query and visualization system[15] for databases. It provides a user interface with support for visualizing structural or relational data as hygraphs. Hygraphs are a hybrid between Harel's higraphs and directed hypergraphs. The system supports query visualizations, the visualization of data that constitutes the input to the query, and the visual presentation of the result. In this sense Hy+ visualizes not only answers but also the questions users ask, allowing for a generic visual abstraction, but it is limited to a special domain.

VANISH[29] is a software tool which supports rapid prototyping of interactive information visualizations through a special-purpose language. While it is a generic tool to create visualizations, it lacks in performance and freedom to interactively adapt, and manipulate graphs and layouts.

CodeCrawler[32] is a programming language independent software visualization tool. It relies on the FAMIX meta-model[16], which models object-oriented languages and also procedural languages. FAMIX has been implemented in the Moose reverse engineering environment[17] which offers a wide range of functionalities like metrics, query engines, navigation, etc. The visualizations implemented in CodeCrawler are based on polymetric views described by Lanza[33].

Mondrian[37] is an information visualization framework. In contrast to CodeCrawler it does not use an internal meta-model, but works directly with the objects in the data model. It provides a user interface with which *programmers* can script visualizations in a declarative fashion. In our solution we adapted the idea of providing a tool set to programmers with which they can build meaningful visualizations but argue for the use of a meta-model of hierarchical data to enable information aggregation.

Another tool that uses the FAMIX meta-model is CodeCity[52]. It is a language-independent interactive 3D visualization tool for the analysis of large software systems. Using a city metaphor, it depicts classes as buildings and packages as districts of a *software city*. While the city metaphor works very well to get an overview of the structure of a system, it does not show relationships. By combining polymetric views with treemaps the visualization in our approach can convey the same type of information as CodeCity. However, the advantage of Quicksilver is that it also provides

information about relationships.

Softwarenaut[36] is a tool for architecture recovery — it enables the recovery of architectural views from a software system through interactive exploration of a hierarchical decomposition of the system. It supports the *"overview first, zoom and filter, and details on demand"* principle of information visualization, which we also adapted for our approach. Quicksilver does not offer all the capabilities of Softwarenaut, *e.g.* it does not show multiple simultaneous perspectives on a system and does not allow for operations on the history of a system out of the box. Softwarenaut was our starting point and inspiration when we began, but Quicksilver is more generic, it allows one to analyze any hierarchical data, not just software.

2.4 Generic Software Analysis

There are tools that are generic in their approach. One of them is the one described by Telea *et al.* in their paper *An Open Toolkit for Prototyping Reverse Engineering Visualizations*. It is a software system which allows prototyping of reverse engineering data exploration and visualization scenarios for a large range of domain models. Another example is the *Generic Software Exploration Environment* (GSEE)[18], which is similar in its holistic approach to Moose: it makes it possible to specify and create new exploration tools.

All of these tools or meta-tools are built for the domain of software. Our solution has its root in software architecture recovery and visualization but offers a more generic approach by abstracting and allowing to analyze any domain's hierarchical data.

3 Hierarchical Data

If we want to deal with heterogeneous data of a-priori unknown specifics we have to find generic concepts inherent in as many data sources as possible. Software architecture serves as our starting point when thinking about data and we extrapolate from there.

3.1 Definitions

According to Bass and Clemens software architecture is *the structure or structures of the system, which compromises software elements, the externally visible properties of those elements, and the relationships among them*[10]. This definition is similar to the one proposed by the IEEE 1471 standard[27], in that they both emphasize elements and the relationships between them. Adding the hierarchical aspect contained in software we propose accordingly that two basic characteristics of software are containment and relation. Containment deals with composition and hierarchy of elements while relations deal with links between elements or their properties.

We define a hierarchy informally as a set with (1) a particular order in which (2) no element is superior to itself, *i.e.* there are no circular relationships in a hierarchy and (3) one element, the root, is superior to all of the other elements. Formally that means that hierarchies, in their most general form, are partially ordered sets. A *nested hierarchy* is a hierarchical ordering of nested sets. Sets with the same parent are grouped together on the same *level* in the hierarchy. On each

level all objects can only have one parent. *Containment hierarchies* are a special sort of nested hierarchies where all of the ordered sets are nested, but every set must be strict, *i.e.* no two sets can be identical.

Finally, a *compositional containment hierarchy* is an ordering of the parts that make up a system, *i.e.* the system is *composed* of these parts.

If we add relationships in the hierarchy, we arrive at *compositional containment hierarchies with relations between components*.

In the next sections we will motivate and present a model for this definition.

3.2 Finding Generic Properties

When we talk about hierarchical data, we mean data, which inherently possesses the mentioned characteristics or data to which such hierarchies can be introduced, *e.g.* by using recursive clustering algorithms and finding other, arbitrary relations between elements. An example of such an approach can be found in the paper *Interactive Exploration of Semantic Clusters*[35].

We hypothesize that these characteristics are inherent or can be found in many data sources besides software and are fundamental enough to allow for generic analysis of hierarchical data.

In Table 3.1 we investigate this idea further by looking at various fields and examples of related data fitting in this model. The table is only listing some examples and is by no means complete but provides us with a certain confidence in our hypothesis.

Domain	Leaf Entities	Hierarchies, Components	Relationships
Object oriented code	methods	classes ▷ packages	invocation, inheritance, reference
Functional programming	functions	files ▷ folders	invocation, reference, etc.
Version control system	commits	branches, tags	changesets
Filesystems	files	folders	symbolic links
Database	tuple	table	relations, views
Networking	hosts	networks ▷ sub-networks	links, bus
The world wide web	page	site ▷ domain	hyperlink
Economy	people, places	political/geographical regions	trade
Politics	people	unions, parties, towns, countries	laws, relations, agreements, trade
Biology	cells	organs ▷ individuals	shared hereditary information, functional links
Sociology	people	families, unions, clubs	norms, shared culture, laws
Astronomy	celestial objects	solar systems ▷ galaxies	gravitational-, electromagnetic effects

Table 3.1: Examples data that posses characteristics of compositional containment hierarchies with relationships between components

3.3 Defining a Model

We start by trying to find a generic pattern which encompasses these characteristics. With information about containment we are looking at hierarchies akin to the composition pattern. The intent using this pattern is to *compose* entities into a tree structure to represent part-whole hierarchies[19]. This allows us to analyze and treat individual entities and compositions uniformly. In other words: By using a composition pattern on data source entities we arrive at a hierarchical structure, in which all entities, except the leaves, are representations of a group of entities. In this sense we will be able to use a divide and conquer technique—representations of groups can be asked about properties of their subordinates.

When relations between entities are added we expand the basic pattern (see Figure 3.1). We call this a *hierarchical graph pattern*. With this pattern we augment the hierarchy with

relationship information which leads us to a graph structure. By using the hierarchy of the composition and merging its value with relationships we arrive at a hierarchical graph. This is to say that after the initial relations have been added, we propagate relationship information up the hierarchy. This allows us to analyze and treat individual relationships and propagate relationships uniformly as well.

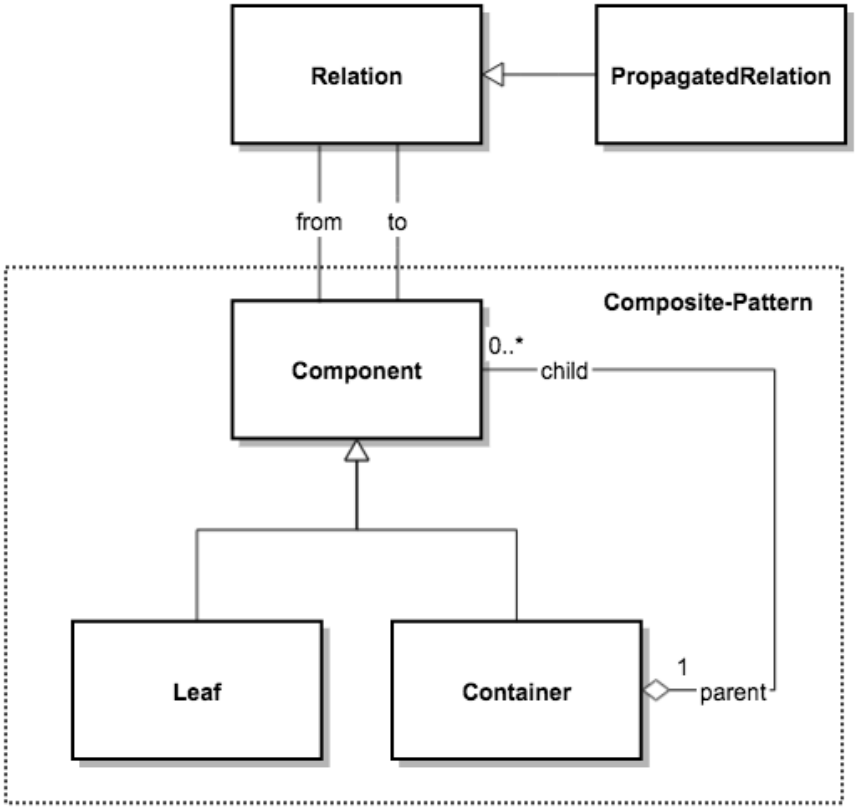

Figure 3.1: Proposed model: hierarchical graph pattern

Based on this model we build a framework which enables the analysis of software systems as well as arbitrary hierarchical data sets. The framework will use a hierarchical graph at its core and is meant to enable building supporting tools for persons who wish to find answers to questions they formulate about their data. It should be enhanced by a visualization engine and offer interactive capabilities to allow for exploratory approaches as well.

3.4 Higraphs vs. Hierarchical Graphs

In 1988 David Harel introduced higraphs[21, 23] — a visual formalism of topological nature. The goal was to combine the advantages of Euler circles, or Venn diagrams, with those of graphs. The first two provide means to partition regions on a plane in disjoint inside and outside regions, thus enabling the topological notion of *enclosure*, *exclusion* and *intersection*. Graphs on the other hand allow one to represent a set of elements together with some special relation(s), thus enabling the topological notion of *connectedness*.

The notion of a hierarchical graph as used in this thesis is similar to the concept of higraphs but it also has some differences. Its base lies not in visual formalism but on structural abstraction. The visualization of the structure and the usefulness of exploring it interactively and visually is added later.

There are small differences in the description of both concepts on the semantical level, *e.g.* what Harel calls blobs, we call nodes, and atomic blobs are leaf nodes.

Generally the hierarchy takes a more prominent role in our case, since, as mentioned, we firstly focus more on the structural than on the visual aspects. The hierarchical tree is our main structure to which relationships are added.

Another notable difference lies in the fact that our hierarchical graphs have no cartesian products or orthogonal states. In higraphs they were introduced to allow concurrent states, *e.g.* in state charts, for which we have no need. We also do not differentiate between enclosures and blobs, for they can both be represented by nodes: an enclosure is simply the parent node of its containing nodes.

4

Hierarchical Data Analysis Framework

The architecture of the proposed hierarchical data analysis (HDA) framework is divided into two main components: the hierarchical graph components and the visualization components. The framework can use any data a user might provide to build the graph, but it is the user's task to tell the framework how to build the hierarchy out of entities and how to establish relationships between entities. We will look at the various steps involved in building and using the framework and also provide some examples of the API of the framework which we realized in Quicksilver, as presented in Chapter 5.

4.1 Graph Building

There are three main steps involved when building a hierarchical graph: (1) building the hierarchy, (2) adding edges and (3) propagating edge information. The hierarchy we are building in the first step is akin to the composition pattern we mentioned earlier. When relations between the nodes are added we expand the basic pattern to a hierarchical graph pattern (see Figure 3.1). The added relations are stored as edges in the graph.

After edges have been added, we propagate relationship information up in the hierarchy. This means that we define that if two nodes in separate parts of the hierarchy share a relations, their parents do as well. We do this for all relations, as long their parental hierarchies do not meet in a

common ancestor or two separate root nodes. Thus we arrive at what we call a hierarchical graph.

With the relationship propagation we make sure that information about high-level dependencies is only computed once.

In the following sections we look at these three steps in details.

4.1.1 Creating the Hierarchy

To build the hierarchy we first add the base entities. They can either be leaf or root entities. All added entities get wrapped in nodes and are added to the collection of all nodes in the graph. The decision on whether to use a top-down or bottom-up approach is driven by the user. He can choose either the *bottomUp* or *topDown* method. By providing instructions on how to navigate the hierarchy of the data we start the building process. The nodes that are already in the graphs collection are now regarded as either root or leaf nodes, based on the method chosen.

If the user is building the graph from top to bottom we know that all nodes previously added are root nodes. This means we can reference the current nodes in the graph as nodes on level 0, the root level. In many cases there will only be one node, but multiple are possible. If the graph is built bottom-up, we first have to find a root node, that is, an entity for which no parent entity can be found, before we know a leaf's node level. Only then, once all nodes have been identified, can we set the level of all nodes. Knowing node levels is important for edge propagation, as we will see below.

In Listing 4.1 we see an example of how we can build a hierarchical graph with a bottom-up approach. Leaf entities (in this particular case: users) are added, then instructions on how to build the hierarchy are given. The instructions simply tell the building process how to get from one type of encountered entity to its parent in the hierarchy (the API will support additional ways of specifying such strategies).

```
1 hg ← HierarchicalGraph with: model users.
2 hg bottomUp: {
3     SoUser->[ :e | e country ].
4     SoPoliticalArea->[ :e | e parent ]
5 }.
```

Listing 4.1: Building a graph hierarchy

4.1.2 Adding Edges

After the hierarchy has been built up and all nodes are in place, edges are added between the nodes and lastly the edges are propagated up the hierarchy. Let's take a look at the first step in detail. To add edges a user has to provide a collection of entities that represent the edges to be

added (*e.g.* models describing inheritance in source code, models describing export in economical data) and two strategy patterns: how to get the source entity of the edge and how to get the destination entity. The source and destination entities have to already exist in the hierarchy, else the relationship will not be added.

A user can add multiple collections of relations to the graph. The only restriction is that the entities we find have to be in the graph already. For example in the case of analyzing a software system we can consequently add inheritance, access, reference, etc. relations.

An edge in the graph wraps the model it stands for and knows its source and destination nodes in the graph. Conversely every node has references to all its incoming and outgoing edges.

In Listing 4.2 we see an example on how edges can be added. In this particular example the relationships we add are answers to questions. All answers are added and we give instructions on how to reach the user who gave the answer and how to reach the user who posed the question.

```
1 hg addEdges: model answers
2     from: [ :a | a owner ]
3     to: [ :a | a question owner ].
```

Listing 4.2: Adding edges to the hierarchy

4.1.3 Edge Propagation

After the edges have been added, we propagate them in the hierarchy. To do this we traverse all node's outgoing edges, except those of the root nodes. By traversing all outgoing edges, we also traverse all incoming edges - they are always some node's outgoing edges. We don't look at edges of root nodes since we can not propagate them anywhere. We are also only looking at edges that have not already been propagated. To propagate edges of two nodes we first have to find either their common ancestor in the hierarchy or notice when their top ancestors are both root nodes. If we find a common ancestor we can create edges between all ancestors of the two nodes below the common ancestor. If there are two separate root nodes found we add propagated edges between all ancestors including the root nodes. We do this by creating two lists: one contains the node ancestry of the source node, including the source node itself, the other does the same for the destination node of the original edge. It is possible that edges are added that link two nodes that are not on the same level in the hierarchy. As a first step we thus add ancestor nodes to the source or destination list, until we are on the same level in the hierarchy. Now we can start to add the parents of the last entries in the lists until we arrive at a root node, or we find that we encountered a common ancestor. If one of these is the case we stop. Now we finally create propagated edges. We go through all nodes in the two lists and create propagated edges from every node in one list to every other node in the other and vice-versa. Propagated edges are created with the original

edges models but with new source and destination nodes.

4.2 Graph Traversal and Attribute Synthesization

Many information retrieval and analysis tasks can only be done after the graph is fully constructed. We only know how many hierarchy levels there are in the graph, or which nodes are leaf nodes, after full construction. If we want to find out maxima and minima of certain attributes of nodes or edges we can only do so with a full graph. The same is true if a certain node attribute is depended on node attributes of its children. To run such tasks we want to avoid two things: (1) traversing the graph more than once and (2) re-calculating values or re-running algorithms that have already been run.

To meet this goal we propose a system which allows arbitrary tasks to hook themselves into an only-running-once full graph traversal. The system uses the visitor pattern to run all given tasks on nodes or edges with a bottom-up approach.

A particular use case of the system we call attribute synthesization. As already mentioned, all nodes and edges can store arbitrary attributes. If we want to aggregate attribute values up the hierarchy, *e.g.* if we define that the weight of parent nodes is the sum of the weight of all their children, we only have to calculate the weight of leaf nodes and sum them up for all parents and add them as attribute values. This method of synthesization stands in contrast to storing a strategy pattern as weight attribute per node which would lead to unnecessary re-calculations.

4.3 Visualization

Visualizations are an indispensable way of communicating information that is hard to express in plain numbers or in prose. The basic building block we use to visualize hierarchical graphs are tree maps[28, 47]. They are able to visualize hierarchical data by subdividing a given space recursively based on some weighting mechanism. They allow us to see whole trees or sub-trees from a top-down perspective and thus provide a pseudo third dimension on two dimensional plains.

4.3.1 Visualizing Nodes

4.3.1.1 A Brief Historical Intermezzo

There are various algorithms for laying out tree maps. An ideal algorithm would produce rectangles with an aspect ratio close to one, *i.e.* squares, to allow for intuitive comparison of area sizes. Furthermore it should preserve a sense of order over the visualized nodes so that small

changes in the underlying data do not lead to totally different layouts which are not comparable. Unfortunately, these properties have an inverse relationship. As the aspect ratio is optimized, the order of placement becomes less predictable. As the order becomes more stable, the aspect ratio is degraded.

The default shape used in tree maps are rectangles. Other approaches exist as well *e.g.* circles, convex polygons[41], voronoi based polygons[8]. Often small margins between nested rectangles are used to denote containment and thus the hierarchy. Another approach is to use shading, *e.g.* cushion tree maps[50].

Tree maps were initially introduced in 1991 to work with the *"slice and dice"* algorithm[28]. The algorithm places child nodes in a rectangle by slicing it in only one direction, *e.g.* vertically, and allocating space for the children rectangles of various horizontal widths based on their weight. On the next level, the slicing direction is alternated. The algorithm preserves the order of nodes but often produces very high aspect ratios, meaning long and skinny rectangles are generated which are difficult to compare, select and label.

To counter-act this disadvantage alternative algorithms were proposed which try to optimize aspect-ratios by giving them a higher priority than order. The squarified tree map algorithm[13] is one such example. This algorithm uses a look-ahead technique to determine if the next placement of a node is increasing or decreasing the aspect-ratio. It places rectangles in one direction as long as the aspect ratio does not get worse. If it gets worse, it starts to fill alternative space. The order of nodes gets mangled in this process but all nodes are as square as possible.

Shneiderman proposed to get order back into tree map layouts in 2001 by improving on the *"slice and dice"* algorithm[48]. This algorithm works by using a pivot, a special node (pivot-by-size: node with the largest weight, pivot-by-middle: node with median weight), which divides the current list of nodes to be placed. It uses three lists: L_1, L_2 and L_3. L_1 contains all items with larger weights than the pivot node. L_2 and L_3 contain the other nodes, while the nodes in L_3 all have lower weights than those in L_2. the algorithm then creates four rectangles: one for the pivot node itself (its aspect ratio is kept as close to one as possible), and the other three corresponding to lists of nodes. The algorithm is then recursively applied to the created rectangles.

In 2002 Bederson and Shneiderman introduced the strip layout algorithm[11]. It is a simplification of the squarified algorithm and produces pretty good aspect ratios while retaining ordering. It works by processing input rectangles in order, and laying them out in horizontal (or vertical) strips of varying thicknesses. It is efficient in that it only looks at rectangles within the strip currently being processed.

In 2006 Vliegen and Wijk proposed to mix various layouting algorithms[51]. They introduced the idea to use different lay-outing algorithms per hierarchy level and also presented a number of variations of the squarified algorithm, the use of variable borders and the use of non-rectangular

shapes.

4.3.1.2 Our Approach

When we devised the tree map algorithm to use for hierarchical graphs, we favored low aspect ratios over steady ordering to make the tree map layout as readable and usable as possible. As we have seen other algorithms differentiate between the underlying ordering of the nodes, *e.g.* an ordering in the model itself, and the order of how the nodes are graphically laid-out. There are use cases where an initial ordering in a model should be kept, so as to facilitate comparison between tree maps when the models ordering changes. Our use case is different. By sorting the nodes via a weighting function right from the start, we impose an arbitrary and user selected sorting of the nodes. This order is then kept and visible in the graphical output. It is stable as long as weights do not change. Other algorithms harden the layout algorithm against change in the underlying model, but in our case, we want the tree map to change, precisely so these changes are visible. In this sense we do not differentiate between a model's order of nodes and their graphical order. It is the same. We also wanted to make sure the layout is easily comprehensible. We decided to create a simple slice and dice variant with weighted sorting and node-list subdivision. The algorithm needs three parameters as input: (1) a list of nodes to layout, (2) a rectangle to layout the nodes in, and (3) a weighting function to sort the nodes by.

The algorithm is recursive and can draw whole hierarchical graphs as one tree map with nested tree maps. Insets are used to denote nesting. The darker the insets become, the deeper in the hierarchy the drawn sub tree map is located.

We propose two additional measures to improve readability and performance of the generated tree maps:

1. **Minimum weighting relations**: The relation to split up the rectangle's dimensions are not allowed to go over (and thus under) a certain, configurable value. The default value is 0.99 (and 0.01). This means if the sum of weight of one half of a split up node collection is larger than 0.99 times the size of the other half, it gets capped there. This measure makes sure every node collection gets a minimal area to get displayed, even if the visual weighting is a bit off then.

2. **Minimum node area**: If the algorithm allocates a too small space (configurable) for nodes to be draw, they do not get drawn. This increases performance immensely when drawing nodes with many children.

Algorithm 1: Custom tree map algorithm

```
Data: list of nodes, rectangle to draw in, weighting function
Result: nodes are drawn as a tree map
if node list contains only one node then
    if node has no children then
        draw the node in the given rectangle and return;
    else
        create an inset in rectangle;
        recurse call myself with inner rectangle and children nodes;
else
    sort list of nodes by weighting function;
    split the node list into two equal sized halves;
    calculate the summed weights of both these halves;
    if rectangle is wider then its height then
        split rectangle vertically by the same ratio as the summed weights of node list
        halves;
    else
        split rectangle horizontally by the same ratio as the summed weights of node list
        halves;
    if there is enough minimum space for the first half of nodes then
        recurse call myself with the first half of nodes and the first rectangle;
    if there is enough minimum space for the second half of nodes then
        recurse call myself with the second half of nodes and the second rectangle;
```

4.3.2 Visualizing Relationships

A single tree map by itself can only show hierarchy and weighting. What is missing to fully visualize a hierarchical graph are relations between separate nodes.

The visualizer should allow a user to display any nodes of a hierarchical graph, *e.g.* a cross-cut on a certain level, or any other arbitrary selection. Nodes in such a collection are displayed as separate tree maps, each showing the nodes sub-graphs. Such a view corresponds to the *module viewpoint* mentioned in Chapter 2.

Before the nodes are displayed, their weights are calculated. From an average node size (configurable) the size of the rectangle each node has available for its tree map is calculated. The weighting algorithm is the same used inside the tree maps. We will call the currently shown

subset of nodes and relations in the visualizer the working set.

A relation always has a source node and a destination node and wraps a certain entity in the model. Since a single relation is always directed, relations between nodes are shown as lines with arrows at the end points. We thus either have none, one or two edges between any given nodes of the current working set. Every single edge represents at least one relation but often stands for multiple. If for example a user is looking at two nodes which represent packages or folders of source code, she or he might see two lines between the nodes, representing all function or method calls from one node to the other.

Edges have various weights denoted as line thickness. The visual weighting algorithm determines the min and max values of the edges weights in the current working set, and linearly adjusts the thickness of the edges. The range of thickness to use is configurable.

An additional measure to improve readability is that the algorithm allows one to define negative values which are translated to reduced opaqueness of a line with width of 1. We calculate the amount of discreet negative weight values, add one (so no weight value has zero opacity) and divide over one to get the amount of opacity reduction per step. If for example the minimum value is -3, then edges with a weight of 0 will be displayed as a line of width 1 with opacity of 80%. -1 the same with an opacity of 60% and -3 with an an opacity of 20%.

In Listing 4.3.2 we see an example configuration of a visualizer. A hierarchical graph is given to the visualizer and after some configuration parameters are set we show the all nodes on the first level (continents in the particular example) and the relationships between them.

```
1 vis ← QsVisualizer with: hg.
2 vis weight: #weight;
3     averageNodeSize: 250;
4     relativeNodeSizeExtrema: 0.05@1.9;
5     edgeThicknessRange: -3@9;
6     layout: QsContinentLayout new;
7     drawLabels: false;
8     showLevel: 1.
```

Listing 4.3: Configuring and using the visualizer

4.3.3 Layout of Nodes

Since any number of nodes can be displayed by the visualizer, we need to find sensible ways to arrange them. Users can move nodes around and arrange them interactively, but the HDA framework needs to have a default layout mechanism in place. The goal of the default layout is to give users a good overview of the nodes and the relations between them. We have two basic entities: nodes and relations, both are weighted. Placing the nodes can either be based on the relations between them or on the nodes themselves. We can either have a node-centric or a

relation-centric layout. The node's weight is the most obvious feature, so we propose to focus on relations as a first step in the layout process to add a balance of importance and as an attempt to combine both approaches. The layout looks for the three (the number is configurable) nodes which have the heaviest relations between them. We call these nodes and the relations between them the backbone. We lay these nodes out with a force-based approach. The other nodes are placed around the backbone in a circle. We call this default layout the backbone-layout. Users can of course also create and use their own layouts.

4.3.4 Filtering

The visualization engine can display any selection of nodes of the hierarchical graph. This allows for easy filtering of certain nodes. See also Section 4.3.6 on how filtering is available through interactivity.

4.3.5 Highlighters

The visualizer allows one to highlight any node or relation in the hierarchical graph in a user defined way. There are two ways to highlight or "colorize" nodes or edges: absolute and linear. Absolute colorizers add the given color to a node or edge, if a certain criteria is matched. Linear colorizers are based on a weight value. They color nodes or edges in the given color with added alpha transparency, depending on the node's or edge's metric value compared to the minimum and maximum metric values of all selected nodes or edges. Multiple absolute or linear colorizers can be used. If a node or edge fulfills multiple colorizers metrics, the resulting colors are mixed together.

4.3.6 Interaction

The HDA framework must provide visualizations which are interactive. The interactive part of the system is very important since it allows users to explore hierarchical graphs, to drill-down, move nodes around, remove aspects which are not important for the question at hand or add others. If the mouse is hovered over a node or edge, information pertaining to this object are shown. Users can also open a context-menu on selected objects to access further actions.

The context menu should allow users to inspect the hierarchical graph's node or edge objects or the underlying models. Nodes can be expanded, compressed or completely removed. When a node is expanded its direct children are added as new nodes to the working set and are drawn separately, meaning the direct sub-graph the node's visualization represents is split up into the sub-graphs of its direct children. Compressing a node is the inverse of this process: all nodes with the same direct parent are removed from the working set, and their parent is added.

5 Quicksilver

In this Chapter we are going to present our proof-of-concept implementation of the hierarchical data analysis framework proposed in Chapter 4: Quicksilver — a tool that provides support to explore and analyze hierarchical data. We will be looking into the implementation details and overall structure of the project. Section 5.1 will detail the architecture of Quicksilver. In Section 5.1.1, Section 5.1.2 and Section 5.1.3 we will look into details of the core algorithms used to build, manipulate and query hierarchical graphs. In Section 5.1.4 we will look into visualization and interaction and in Section 5.1.6 we present a package with various code examples on how to use Quicksilver. In Section 5.2 and Section 5.3 we will present examples of how we can use Quicksilver to answer questions about hierarchical datasets.

5.1 Architecture

The implementation of Quicksilver was written in Pharo[1], an open-source Smalltalk-environment released under the MIT license[2]. It uses a hierarchical graph as its base data structure and accompanying algorithms to offer its functionality. These have been contributed to Moose[17], an extensive platform for software and data analysis, also built on Pharo. For its graphical

[1] `http://www.pharo-project.org/`
[2] `http://opensource.org/licenses/MIT`

output Quicksilver uses the Roassal[3] visualization engine. A tree map layout and other small improvements have been written and contributed to the project.

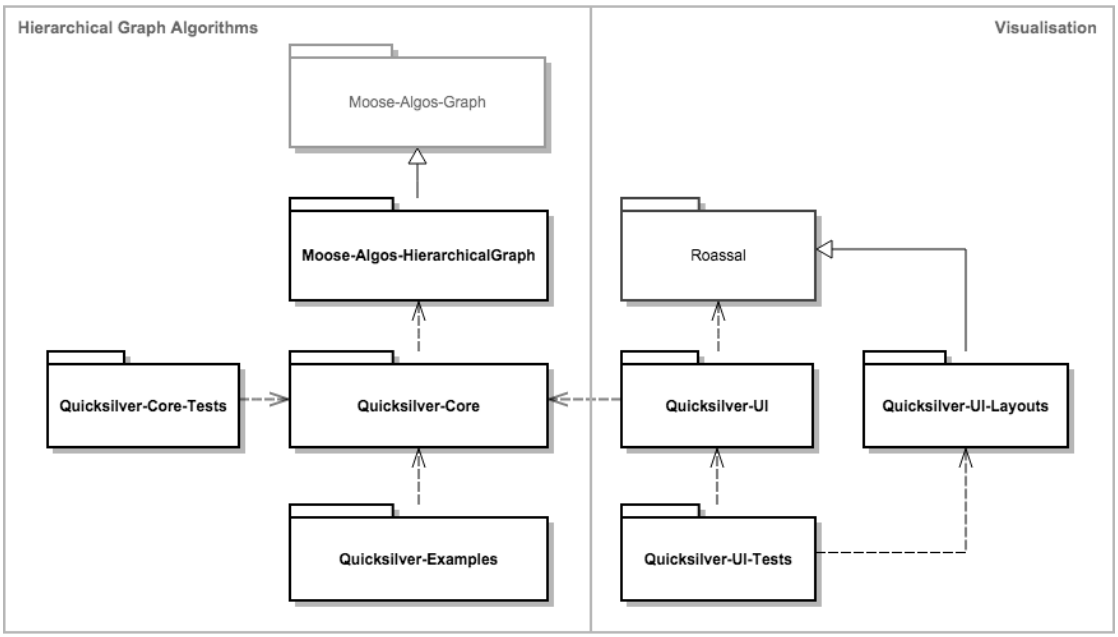

Figure 5.1: Overview of the Quicksilver architecture

Figure 5.1 represents an overview of the architecture of Quicksilver and how it is embedded in its ecosystem. The architecture is divided into two main components: the hierarchical graph algorithms and the visualization engine. The core of the first has been integrated into the Moose ecosystem and is part of the *Moose-Algos-HierarchicalGraph* package. *Moose-Algos* are generic libraries for various analysis algorithms. They are self-contained and can be reused out of the box. *Moose-Algos-Graph* is a stand-alone library containing both classical and prototypical graph algorithms. Quicksilver adds additional features and utilities in the *Quicksilver-Core* package to complete its functionality. All classes related to visualization can be found in the *Quicksilver-UI* package.

5.1.1 Graph Algorithms

As we have seen a hierarchical graph is composed of a set of nodes and two types of relationships: (1) containment relationships, which build a partial order on the nodes, and (2) direct,

[3] `http://objectprofile.com/Roassal.html`

node-to-node relationships which are aggregated across the hierarchy resulting from the containment relationships. By propagating relationship information up in the hierarchy we arrive at a hierarchical graph.

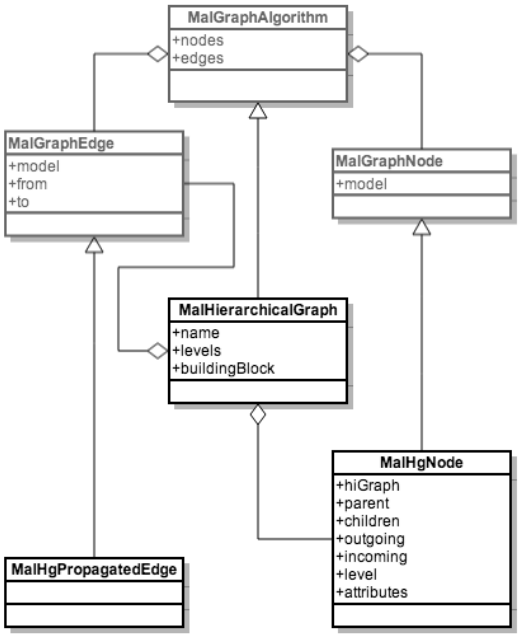

Figure 5.2: Class Diagram of Hierarchical Graph Classes

In Figure 5.2 we see a class diagram representing our hierarchical graph components. *MalHierarchicalGraph* inherits from *MalGraphAlgorithm*, it adds an instance variable called levels. Here we store additional references to node-collections per level, thus enabling us to have an abstraction or view over the hierarchy of the graph. *MalHgNode* inherits from *MalGraphNode* and mainly adds instance variables that help in identifying its place in the hierarchy and its relations. It has references to the hierarchical graph it belongs to, it knows its parent and its children (if there are any), its outgoing and incoming edges and on what level in the hierarchy it resides. Attributes is a general purpose field, a dictionary, which can be used to add arbitrary information about nodes, *e.g.* weights. References to parent and child nodes are a consequence of using the composition pattern, nodes being used as surrogates for their children. Looking in the other direction, we know that the parent is a delegate of the node we are looking at, it being one step higher in the hierarchy. We store the additional added incoming and outgoing edges but also the propagated ones, which allows us to use the same hierarchy propagation effects for edges. *MalGraphNode* and *MalGraphEdge* both store references to their models, that is, to the

underlying entity which they wrap and represent.

5.1.2 Graph Building

As we have seen in the HDA framework description in Chapter 4 we want to provide users with two possibilities to build the graph: with a bottom-up and a top-down approach. Knowing and setting the level property for nodes is an important but also costly factor when building the graph. For both methods of construction there are two ways to set the levels of the nodes. However we set the levels, the root node(s) should always be on level 0 with rising levels when going down the hierarchy. This is a convention of the API, and its user must be able to count on it.

Performance Analysis. We ran benchmarks to compare the performance for the different methods. We ran the profiler in Pharo while building a hierarchical graph with 11 levels and 3 children per node, resulting in a total of 265'719 nodes.

For the top-down approach we can either set the node's levels simultaneously with the construction of the graph or after the full graph has already been built. Not surprisingly setting the levels while building was faster than going through all nodes at the end again, although the difference was pretty small: Over 5 runs we get an average build time of 1'612 ms (on-going level-setting) versus 1'712 ms (a-posteriori level setting). We thus decided to use the on-going level setting mechanism for top down building.

When building the graph bottom-up we face the challenge that we never know on which level we currently are in the building process until we reach a root node. It is thus not possible to set the levels correctly in an on-going process. We saw three possibilities to set the levels for this method: (1) calculate levels continuously nevertheless with an arbitrary initial level for the leaf nodes, then correct the levels at the end, once we know the root node's level, or (2) start with the first method but do not correct levels at the end, merely store a level delta in the hierarchical graph and (3) set levels only after the whole hierarchy has been built. We implemented all three methods and found that method (1) was not fast at all. Starting with an arbitrary level number for the leaves and then counting down as new levels are encountered is no problem: the levels are correct relative to each other. However, when we want to correct the levels at the end, we find that we can't directly rewrite the keys of the level dictionary (which are the levels), since we would get collision problems while trying to access and write to the same levels, so we have to create a new dictionary and add the levels separately at the correct keys. This means we would temporarily need double the amount of space for the dictionaries for the levels, which tend to get pretty large with large hierarchical graphs. This method is thus not an option. Method (2) is an optimization of (1): by storing a delta from the found root node(s) level to 0, we do not

have to rewrite levels. It has the disadvantage that every time we want to get a level of nodes, or we want to ask a node's level, we have to subtract this delta from its found level. We ran the same benchmarks used for the top-down approach and found that setting levels at the end is much faster than trying to set them simultaneously. For 5 runs for he second method (using level-delta) we get an average building time of 27'836 ms versus 5'830 ms for the third method (a-posteriori level setting). The slowness of the second method can be attributed to the additional, on-going and costly manipulation of the values in the dictionary. Every time we find a parent we have to add it separately to the collection of nodes on that level. This boils down to many dictionary searches and value changes. We do not have the same problem in the top-down approach, since we can always add all the children at once to a level. This is also why the a-posteriori method is faster: Although we have to go through the whole graph again we can do it much faster, since all information is already there. Dictionary manipulations are optimized in the sense that we do as few as possible. We thus decided to use method (3) for the bottom-up building process.

Storing Nodes. Besides the collection of all nodes, the graph stores references to collections of nodes per level in a dictionary to allow for fast querying of the hierarchy. Additionally the nodes themselves also know on which level they are. To know for each node on which level it resides, and to know which nodes are on which levels is essential for the edge propagation functionality as described in Section 4.1.3.

Adding Edges. We thought about allowing new nodes to be found and consequently to be added to the graph (*i.e.* if edges lead to nodes which are not yet known) but decided against it. We think that from the standpoint of a user of the API it makes more sense to only search for relations inside the graph he or she has built in the first step.

If new nodes are added to the graph we face the challenge that we have to dynamically rebuild its structure, *e.g.* in the case that new root nodes are found, maybe even on a higher level than was known before, or new leaf nodes, on a lower level, etc. While technically not impossible, this can lead to confusion from the users perspective. Once the hierarchy has been build, as per the users specifications, it should not be tampered with.

Edge Propagation. We found that in some cases users of Quicksilver are only be interested in edge propagation information between certain entities in their data, *e.g.* they might only be interested in export and import of goods between continents and not between single countries. If the data available is only on the level of, say states, we have to aggregate this data, we can't just directly introduce relationships between continents.

Only adding propagation information on certain levels in the hierarchy leads to a significant boost in performance and reduction in build time of the graph. This is why it is possible to start the edge propagation process by giving it a list of entity classes that should be considered when creating propagated edges. Edges between nodes wrapping other entities will be ignored when propagating relationship information up in the hierarchy.

Pre-computation. While working on the graph building system, we revisited the thought of lazy-loading the graph while it is used. Calculating the full hierarchical graph up-front can take a long time. The advantage of a pre-built graph are obvious: no loading time while exploring the hierarchical data. Could there be any advantages of only building the graph as far down as it is actually explored? The first problem is already apparent in the statement before: it could only work in the case of a graph that is built in a top-down manner. A graph that is built bottom-up has to be fully built, because only then we know where the root nodes are.

Furthermore, metrics that are relative to a maximum or minimum over all data in the graph can only be used with a full graph. The only advantage of a lazy-loaded graph would be the case that if a user does not need to drill down to the bottom of the data, it would not need to be calculated (but again only if the graph was built top-down). This advantage is quickly lost by the fact that the user could in this case build the graph only to a certain level from the beginning.

5.1.3 Quicksilver-Core

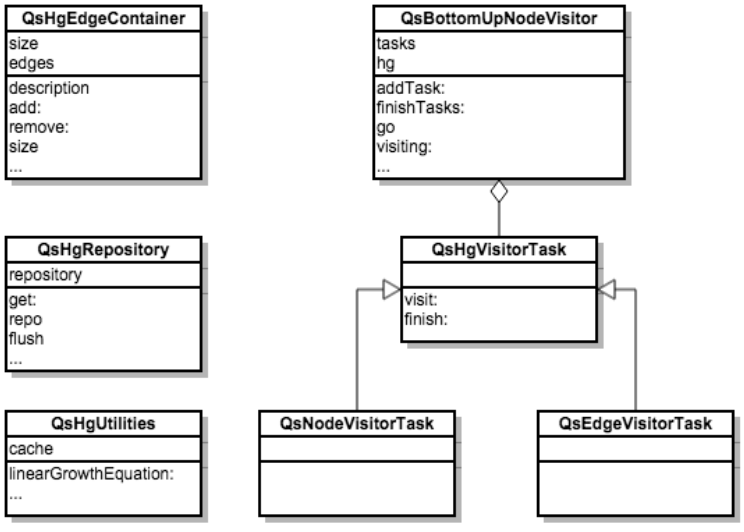

Figure 5.3: Class Diagram of Quicksilver-Core package

Since the main algorithms of Quicksilver have been integrated into *Moose-Algos-HierarchicalGraph* the *Quicksilver-Core* package merely holds some utility and supporting classes for the system. Figure 5.3 shows an overview of the package. The main functionality offered is a graph traversal system described in Section 4.2: a visitor pattern based system which allows one to run arbitrary tasks on edges and nodes in the graph.

Currently Quicksilver mostly has needs for node visiting — realized in *QsBottomUpNodeVisitor*. Instances of *QsHqNodeVisitorTasks* can be added and will then be run. *QsHgEdgeVisitorTask* is also available and allows one to run arbitrary task on all edges.

Other functionalities offered are the following: *QsHgRepository*, which allows one to cache graphs. *QsHgEdgeContainer*, which acts as a container for aggregated edges. And finally *QsHgUtilities*, which offers miscellaneous supporting functionality.

5.1.4 Quicksilver-UI

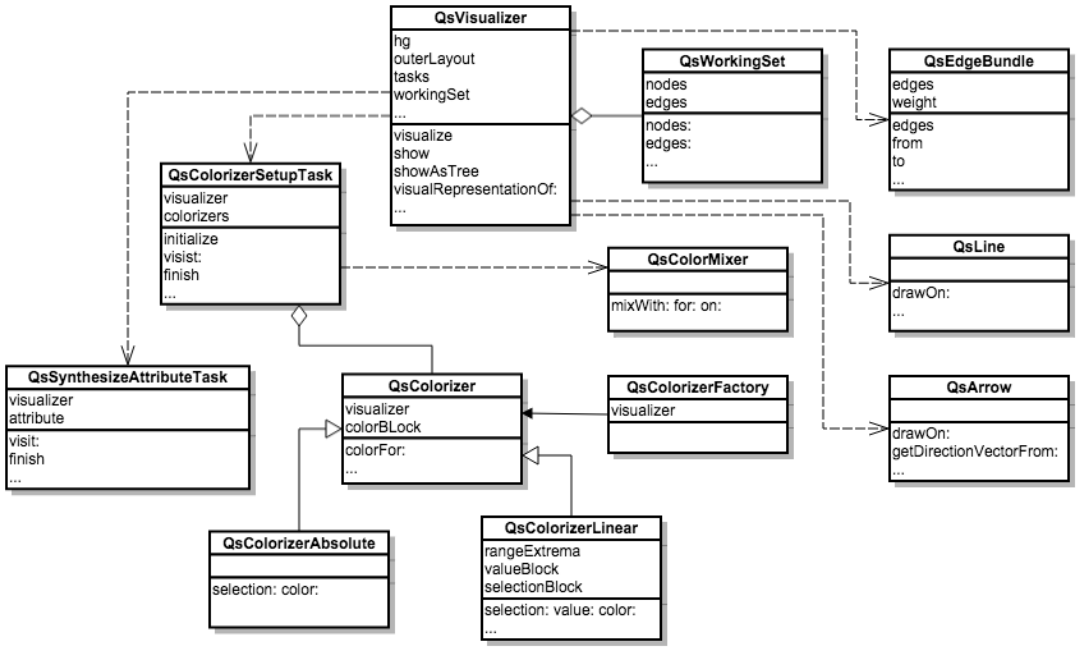

Figure 5.4: Class Diagram of Quicksilver-UI package

Figure 5.4 shows the most important classes of the package *Quicksilver-UI*. The visualizer is built on top of the Roassal visualization engine. It is used for all graphical output and interactivity in the user interface (UI). *Quicksilvers-UI* systems main class is *QsVisualizer*. It handles all UI-related tasks and functionality with the help of numerous other classes in the

same package. Attribute synthesization is realized in *QsSynthesizeAttributeTask* (a subclass of *QsHqNodeVisitorTask*) and is used to calculate node weights.

The tree map layout algorithm as described in Section 4.3.1 was implemented and contributed to Roassal as its standard tree map layout algorithm.

Highlighting. The visualizer allows one to highlight any node or relation in the hierarchical graph. There are two ways to colorize nodes or edges: absolute and linear. Absolute colorizers add the given color to a node or edge, if the selection block returns true for a node or an edge. The following code for example colorizes all nodes which wrap Continent entities in a certain color.

```
vis colorize absolute
  selection: Continent
  color: (Color fromHexString: #C4ECEC).
```

Linear colorizers are based on a weight value. They color nodes or edges in the given color with added alpha transparency, depending on the node or edges value compared to the minimum and maximum values of all selected nodes or edges. The following is an example for such a linear colorizer:

```
vis colorize linear
  selection: FAMIXFunction
  value: [ :f | f localVariables size ]
  color: Color red.
```

All nodes whose wrapped entity classes are *FAMIXFunctions* are colorized red, with an alpha transparency based on the number of local variables inside the function. The node representing the function with the maximum value is colorized in deep red while function with less local variables get less and less opaque. It is possible to specify two colors for linear colorizers: the first one will be mixed with the second one, based on the value. If only one color is provided as in the example above, the second color, or background color, is set to white. If for example one would specify the colorizer in the following way:

```
vis colorize linear
  selection: FAMIXFunction
  value: [ :f | f localVariables size ]
  color: { Color red, Color black }
```

Nodes with the maximum value would be deep red, as before, but nodes with fewer values would become darker and darker, and the nodes with the fewest number of local variables would be displayed in black.

When sending the message ‘colorizer’ to the visualizer, an instance of *QsColorizerFactory* is returned. This factory then creates instances of either a *QsColorizerAbsolute* or of *QsColorizer-Linear*, the given selection and colors are added, and the colorizer with its configuration is added back to the visualizer. Multiple colorizers can be added to a visualizer in this way. If one has defined two or more colorizers the resulting colors are mixed per node or edge.

The visualizer stores the list of colorizers and hooks a *QsColorizerSetupTask* into the visitor pattern task system that runs through the whole graph exactly once (see Section 4.2). While the graph is traversed, minimum and maximum values for linear colorizers are set and updated. All colorizers blocks are then mashed together in an instance of *QsColorMixer*. The class provides a method which adds all colorizer results for a single node together, by adding each result with an alpha channel strength of 1 over the total number of colors. The resulting generic block is set as the block to run on all nodes and edges while they are drawn.

5.1.5 Quicksilver-UI-Layouts

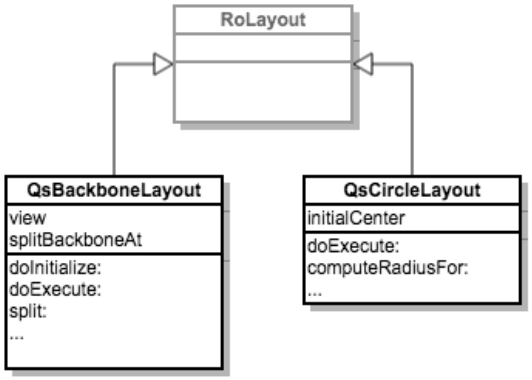

Figure 5.5: Class Diagram of Quicksilver-UI-Layout package

Figure 5.5 shows the package *Quicksilver-UI-Layouts*. The backbone layout algorithm as described in Section 4.3.3 was implemented as *QsBackboneLayout*. A circle layout was also implemented.

The two basic principles of how hierarchical graphs can be represented visually are edge- or node-centric. With the backbone layout we envisioned a layout that combined both viewpoints into one. Ideally there would be ways for users to switch between the main layout but also between layouts that are fully centered on one aspect of the two.

The visualization of node sizes are relative to the extremes of a certain metric of the current working set. Currently there is no way to directly compare different working sets based on this

metric.

5.1.6 Quicksilver-Examples

This package offers many examples on how to build and use hierarchical graphs with Quicksilver including all those mentioned in this thesis.

See also Appendix A for a short introduction on how to use Quicksilver and start playing with some of the examples.

5.2 Using Quicksilver

In this section we are going to show two examples of how to use Quicksilver to construct and use hierarchical graphs. The first builds a simple hierarchy based on a file system, and the second builds a graph based on a model of software including file system information.

Since different data models will have different representations, the API of Quicksilver supports hierarchical graph building with either top-down or bottom-up approaches. The following example code builds a graph out of a directory tree where the containment relationship is specified in a top-down manner to match the way the data structure is defined.

```
1 directory ← '/systems/ArgoUML/' asFileReference.
2 hg ← MalHierarchicalGraph new.
3 hg
4     root: directory;
5     topDown: [:e | e isDirectory ifTrue: [ e children ]].
```

Listing 5.1: ArgoUML filesystem - top-down graph construction

The user of the API needs only to provide two pieces of information: (1) what is the root and (2) how can the hierarchy be constructed in a top-down manner.

The graph building process starts from the root entity: the directory. Internally it gets wrapped in a root node and is set as initial node in the graph. After the top-down block has been specified, the hierarchy is built. We add the root node to the first level in the hierarchy and get all children of the root entity, wrap them in nodes as well and add them on the second level in the hierarchy. And so on. We do this, until no more children are returned from the top-down block, *i.e.* we are dealing with file entities, and thus leaf nodes.

A hierarchical graph can be similarly constructed with a bottom up approach. In the following code example, we use a Moose model of argoUml 0.34, an open source UML tool written in Java. We build the hierarchy by starting with all methods in the model. From the methods we get the next level: classes, from classes we get packages. If an entity is of another type (i.e. a kind of another class) as listed, nil is returned, meaning we are at the top of the hierarchy. In this way, all

found packages are added as root nodes. As we see from this example, our hierarchical graph allows for multiple root nodes.

```
system ← self argoUmlModel.
hg ← MalHierarchicalGraph new.
hg
    leaves: system allMethods;
    bottomUp: {
        FAMIXMethod->#belongsTo.
        FAMIXClass->#parentPackage
    }.
```

Listing 5.2: ArgoUML Moose model - bottom-down graph construction

In the example we see an alternative way to specify the hierarchy construction block: the class based way. The algorithm is provided with an array of class to method name associations. From this information a build-block is created internally, which evaluates the given method name for entities which are of given class. It is also possible to provide class to block associations.

When exploring the hierarchical graph of argoUml we encounter 130 packages as root nodes 2'711 classes on the second level and 21'693 methods on the third.

The next step in creating a useful graph is to specify how to get the relationships between entities. For the second example, one might be interested in all invocations between methods. For example: a relationship between node wrapping method A and node wrapping method B is added if if method A is calling method B.

We can specify these relationships in the following way:

```
hg edges: system allSureInvocations
    from: #sender
    to: [ :e | e to first ].
```

Listing 5.3: ArgoUML graph - adding relationships

To define the edges between the nodes in the graph one has to provide the relationship entities and instructions on how to get the source and destination entities. In the example, we provided all invocations from the Moose model as edges, a group of *FAMIXInvocations*. The method *#sender* and the given block will return the source and destination methods of a single invocation. We define a short block for the destination part which returns the first entity in a collection. We do this because the *#to* method will return a list of candidate-invocations. Because we are in a dynamically typed environment, not all invocations are clearly attributable to their destinations. The method *#allSureInvocations* gives us only one candidate, but it stills returns a list. So we can just use the first element.

We now have a hierarchical representation of a certain system with invocation relationships between leaf nodes, namely between all methods. What if we are interested in the relationships

between packages of this system, *i.e.* we could ask: how many methods of package A have a relationship with methods in package B? In this case we want to aggregate all invocations from the method level up the hierarchy to the package level.

This final step in the graph building process has already happened behind the scenes in the code example above: the edges are propagated up in the hierarchical graph. This means if entity A and entity B have a relationship, their respective parents in the hierarchy now have one too. We do this with all edges in the graph. This means that all node tuples on a level will have edges representing all edges between their respective sub-graphs directly between them. When looking at the example given above, we are now able to directly access all invocations between two packages.

The API allows both for a declarative and imperative way of building the hierarchical graph. We have seen the declarative way in the examples above. If, for example, we would like to specify multiple kinds of relationships in the graph and imperatively tell the graph when to propagate the edges, we are able to do so in the following way:

```
1 hg addEdges: system allInheritanceDefinitions
2     from: #from
3     to: #to.
4 hg addEdges: system allAccesses
5     from: #from
6     to: #to.
7 hg propagateEdges.
```

Listing 5.4: ArgoUML graph - adding multiple relationships

So far we have seen how to build a hierarchical graph. In the next section we are going to look at how we can visualize hierarchical graphs and how we can use Quicksilver to answer specific questions about a system.

5.3 Advanced Example

In the last section we have seen examples of how to build a hierarchical graph based on a Moose model and based on a file system. To show the benefits of the HDA framework, which is general enough to model both software entities and anything else, we are going to combine the two approaches. We are going to look at the Apache httpd server source code[4] to illustrate this approach, some advanced features of the API and the visualization engine.

C code knows no real namespaces or packages, it is organized in folders by developers. This structure is not reflected in the source code itself. Using Quicksilver we are going to create a hierarchical graph that contains this additional information. We are going to use a Moose model

[4] `https://www.apache.org/dist/httpd/`

of the https source code to extract all functions as leaf nodes and use them as the base of our hierarchy. On the next level we will use the files they are contained within. Then we add folders and parent folders up to the projects root folder. We are going to construct the hierarchical graph in the following way:

```
model ← MooseModel root entityStorage at: 'httpd-2.4.6'.
functions ← model allFunctions asOrderedCollection.
rootDirectory ← functions anyOne sourceAnchor rootFolder.

hg ← MalHierarchicalGraph with: functions.

hg bottomUp: {
  FAMIXFunction->[ :e | e sourceAnchor fileReference]. "function -> file"
  FileReference->[ :e | e ~= rootDirectory ifTrue: [ e parent] ]. "file/dir
    -> dir"
}.

hg edges: model allSureInvocations
  from: #from
  to: [ :e | e to first ].
```

Listing 5.5: httpd graph construction

The Moose model was created by using InFusion[5] by Intooitus. With inFusion it is possible to create a FAMIX model out of the httpd source code and export it as an MSE file. We can then import the FAMIX model into Moose and use it in Quicksilver. Moose is reporting that the httpd server consists of 6098 functions. We are going to use these as the base and build the hierarchical graph with a bottom up approach. For every *FAMIXFunction* we get a *FileReference*, which tells us where it is contained in. For every *FileReference* in turn we get its parent Folder, except if it is the root directory. This gives us the hierarchical structure of the graph. Next we add all function invocations as edges, which will then be propagated as described above in Section 5.2. This yields us 6'402 nodes and 380'514 edges in the graph.

A question we might ask about the httpd source code is whether the folder structure created by the developers is sound in relation to the invocations of functions from one folder to the other *i.e.* if there is a good modularity in place. If we look at the folder structure itself, we see a folder called modules, indicating that modularity seems to be an important topic to the developers. Let us try to visualize this modularity and see if it holds when looking at function invocations.

We can use *QsVisualizer* by simply instantiating it, associating it with our hierarchical graph and sending it the message show.

```
vis ← QsVisualizer with: hg.
vis show.
```

Listing 5.6: httpd graph visualization

[5] http://www.intooitus.com/products/infusion

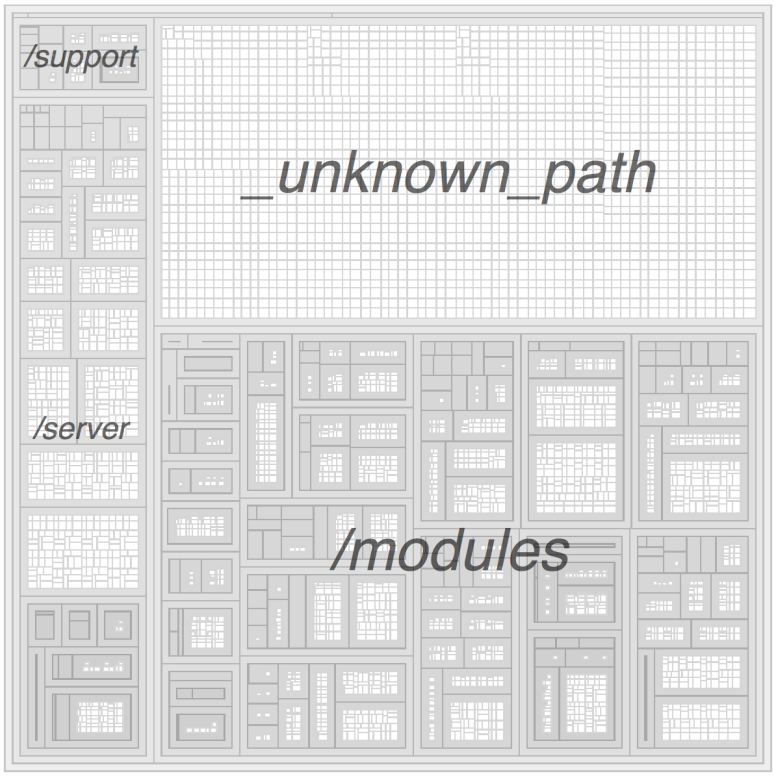

Figure 5.6: basic httpd treemap, each function has a weight of 1

In the default configuration the visualizer will draw a tree map of the hierarchical graph from the root node as can be seen in Figure 5.6. The rendered tree map gives us an overview of the *httpd* folder and file structure and shows us in which folders most functions reside (annotated in the figure). In the default configuration a weight of 1 per entity is used. This means in our case, the larger an area the more functions are contained within. In the created visualization we see seven areas in the system. These areas correspond to seven folders of the root directory which contain code. The node */modules* with 2861 functions has the largest area. Next is a node with around the same amount of direct children called */_unknown_path*. The folder */server* contains 1023 functions and */support* contains 127 while */os*, */test* and */include* all contain less than 25 functions and are barely visible. Upon further investigation we see that */_unknown_path* is a placeholder node used in the FAMIX model, it contains all functions (mostly invocations of external sources) that could not be resolved within the model. The visualization offers various interactive parts that cannot be seen in the figure. If the mouse pointer is hovered over nodes or edges they are highlighted. A context menu allows us to get more information about each node or edges or the entities they wrap. We can explore and inspect these objects. It is also possible to expand and compress nodes, to show their children or their parents respectively. The node(s) can

also be dragged and moved around to obtain custom layouts.

While this basic view gives us a certain understanding about the systems structure, it does not help us answer our question. Let us start setting up the visualizer so we can see the systems modularity at a glance. First let us use another weighting mechanism. The visualizer can be configured to use any block or value available in the model as weight. Rather than knowing where most functions are, we are more interested in where most of the code resides. We will use lines of code (LOC) per function as new weight:

```
1 vis ← QsVisualizer with: hg.
2 vis weightSynthesized: #numberOfLinesOfCode.
3 vis show.
```

Listing 5.7: httpd graph visualization - weighted

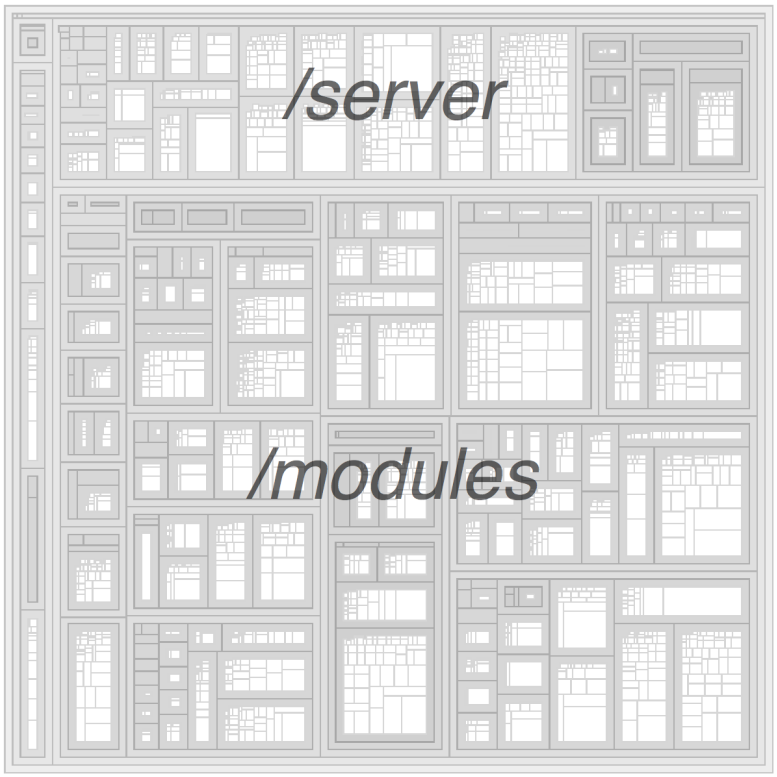

Figure 5.7: basic httpd treemap, weighted by LOC

We now see a more relevant picture in Figure 5.7: the */_unknown_path* node is not visible anymore since it contains no code in the limits of this model. The folder */modules* takes up most of the space, almost ¾ while */server* takes up around ¼. The other folders share the minimal space that is left. This breakdown confirms our initial hunch that modules seem to play an important

part in the system's structure.

Let us now look at all the nodes on the first level in the graph and at the dependencies between them. The graph's root node has level 0, level 1 would thus be all its children or in other words all the mentioned folders. In Listing 5.8 the visualizer is asked to show all nodes on level 1. We also tweak some of the default options: we want the theoretical average node to have a size of 100, with the smallest node not being smaller than 5% and the largest not bigger than 300% of that value. Edges between nodes, denoting number of invocations, should have a thickness range of 1 to 10. Negative numbers can be used to tweak opacity of the edges with the least amount of invocations. In this particular case the smallest edges will be displayed with a thickness of one but only with an opacity of 33%. We also added a block telling the visualizer to cut node labels to a more readable size, only showing the relevant portion.

```
1 vis ← QsVisualizer with: hg.
2 vis weightSynthesized: #numberOfLinesOfCode.
3     averageNodeSize: 100;
4     relativeNodeSizeExtrema: 0.05@3;
5     edgeThicknessRange: -3@10;
6     nodeLabelBlock: [ :n | n model name copyFrom: 50 to: (n model name size)
      ];
7     showLevel: 1.
```

Listing 5.8: httpd graph visualization - configuration

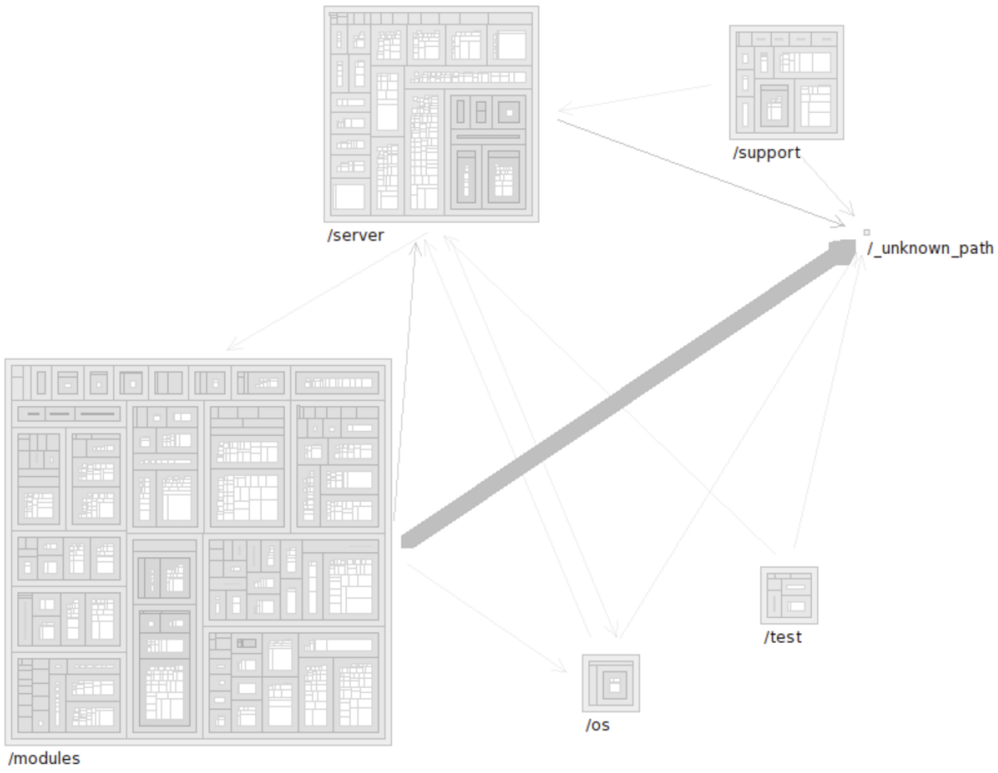

Figure 5.8: httpd nodes on first hierarchical level

In this configuration (Figure 5.8) we see one thick arrow and many small ones. The thick arrow is showing us function invocations from the large */modules* folder to the very small */_unknown_path* node. This means we have a proportionally great number of function calls to external sources, which stands in our way to investigate the intra-model relationships. Since this information does not help us answer our initial question, which is only related to the confines of the model, let us get rid of this node. In the next step we will create a selection of nodes and instruct the visualizer to show only the corresponding sub-graph.

```
1 vis ← QsVisualizer with: hg.
2 vis weightSynthesized: #numberOfLinesOfCode;
3     averageNodeSize: 100;
4     relativeNodeSizeExtrema: 0.05@3;
5     edgeThicknessRange: -3@10;
6     nodeLabelBlock: [ :n | n model name copyFrom: 50 to: (n model name size)
      ].
7 nodesToShow ← OrderedCollection withAll: (hg level: 1).
8 nodesToShow ← nodesToShow reject: [ :n | '_unknown_path' match: ((n model
      name subStrings: '/') last) ].
9 vis show: nodesToShow.
```

Listing 5.9: httpd graph visualization - showing sub-graph

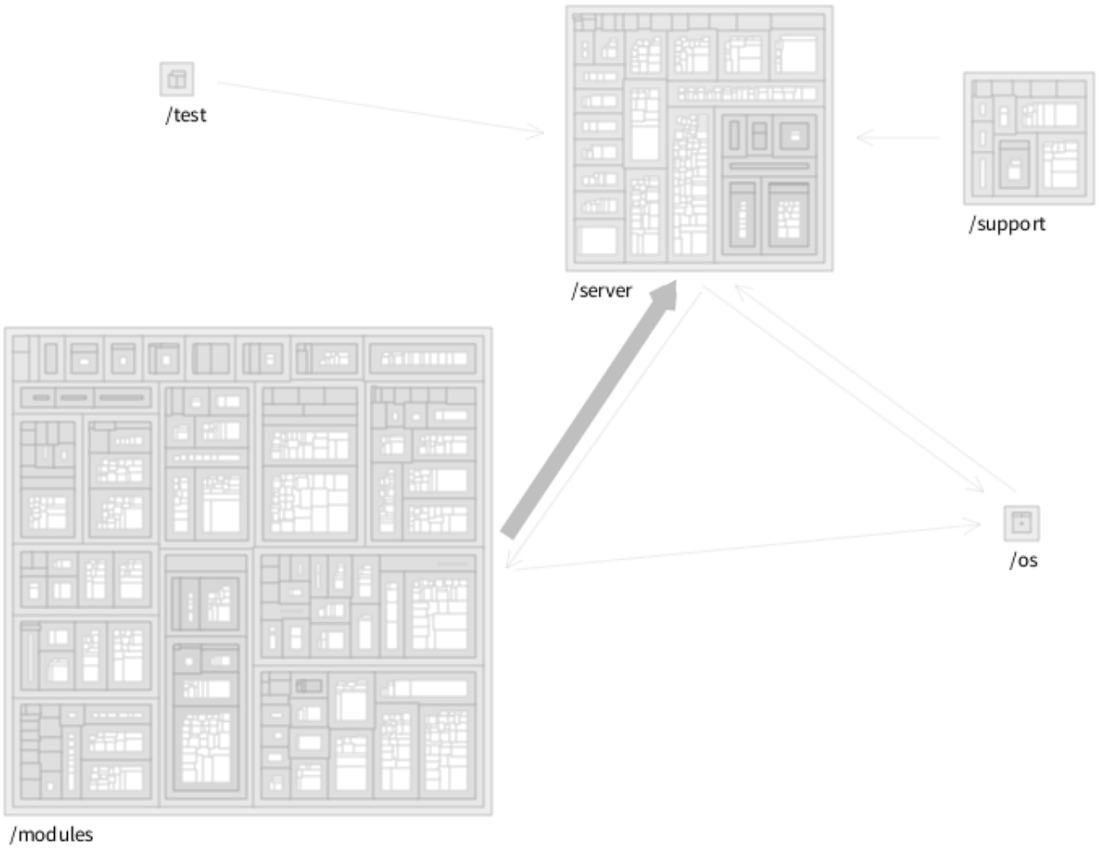

Figure 5.9: httpd nodes on first hierarchical level without unknown */_unknown_path* package

In this view (Figure 5.9) we also see one thick edge and many small ones but this time the thick edge is shown from */modules* to */server*. This seems to speak for a very modular architecture with */server* as core of httpd and */modules* as main consumer of core functions. We also see some function calls from */server* to */modules* though. This could be signs that the modular concept is not fully adhered to in the sense of a layered architecture. To see the modularity in more detail, we split up the */modules* node into its children and display these. To achieve this we give the visualizer another set of nodes to display. The visualizer can display any cross-cut of the graph in such a way. It is also possible to interactively select, expand, compress or remove nodes from the current working set, as we will see later.

```
1 vis ← QsVisualizer with: hg.
2 vis weightSynthesized: #numberOfLinesOfCode;
3     averageNodeSize: 100;
4     relativeNodeSizeExtrema: 0.05@3;
5     edgeThicknessRange: -3@10;
6     nodeLabelBlock: [:n | n model name copyFrom: 50 to: (n model name size)].
```

```
7  nodesToShow ← OrderedCollection withAll: (hg level: 1).
8  nodesToShow ← nodesToShow reject: [ :n |  '_unknown_path' match: ((n model
       name subStrings: '/') last) ].
9  nodesToShow addAll: ((nodesToShow select: [ :n | 'modules' match: ((n model
       name subStrings: '/') last) ]) last children).
10 nodesToShow ← nodesToShow reject: [ :n | 'modules' match: ((n model name
       subStrings: '/') last) ].
11 vis show: nodesToShow.
```

Listing 5.10: httpd graph visualization - refining sub-graph selection

Figure 5.10: httpd graph on level one without references to model externals

In Figure 5.10 we now see a more complete picture of the httpd architecture. The functionality in */server* seems to be in the center of all invocation calls, we can assume it is the core of the software. Around it we have many modules that enhance the core and are mostly independent of each other.

- Modules *proxy*, *filter* and *aaa* (authentication and authorization) are invoking the most functions in */server*.
- Module *http* has the most incoming function invocations, it seems that it is also a very basic module on which others depend, which is no surprise, since we are looking at an http server.
- There are some other minor dependencies between certain modules though, it would be interesting to investigate whether these dependencies are made explicit somewhere and permitted in an architectural sense or if some are errors in the design.

To highlight certain metrics we can use colorizers. If for example we want to check out which functions invoke the most other function, we can colorize them in a certain manner. We want to color them in a linear way, meaning we color those that have the highest number of outgoing invocations in a strong color and make them lighter the less outgoing invocations they have. We can do this in the following way:

```
1  vis ← QsVisualizer with: hg.
2  vis weightSynthesized: #numberOfLinesOfCode;
3      averageNodeSize: 100;
4      relativeNodeSizeExtrema: 0.05@3;
5      edgeThicknessRange: -3@10;
6      nodeLabelBlock: [ :n | n model name copyFrom: 50 to: (n model name size)
       ].
7
8  vis colorize linear
9    selection: FAMIXFunction
10   value: [ :f | f outgoingInvocations size ]
11   color: Color cyan.
12
13 nodesToShow ← OrderedCollection withAll: (hg level: 1).
14 nodesToShow ← nodesToShow reject: [ :n |  '_unknown_path' match: ((n model
       name subStrings: '/') last) ].
15 nodesToShow addAll: ((nodesToShow select: [ :n | 'modules' match: ((n model
       name subStrings: '/') last) ]) last children).
16 nodesToShow ← nodesToShow reject: [ :n | 'modules' match: ((n model name
       subStrings: '/') last) ].
17 vis show: nodesToShow.
```

Listing 5.11: httpd graph visualization - highlighters

In Figure 5.11 this linear colorization gives us some additional pointers on where further investigations could yield interesting results. For example we see one large method in the *proxy*

module which invokes a lot of other functions, and there is also one in the *lua* (scripting language) module which stands out.

It is also possible to define multiple colorizers and mix them. We could add another linear colorizer which would highlight incoming function invocations in red for example. Methods which would then have both a high outgoing and incoming function invocation count would be rendered in dark violet, a mix of cyan and red.

Depending on the question that is asked, different visualizations and configurations can be used to find answers. In this section we have used Quicksilver to find answers about a software system. We already have seen some of the generic power of the tool by combining a model of software with its file system structure, a feature not available in Moose itself nor in the Softwarenaut or any of the other related work we discussed. In the next chapter we are going to look at two case studies to further validate our approach.

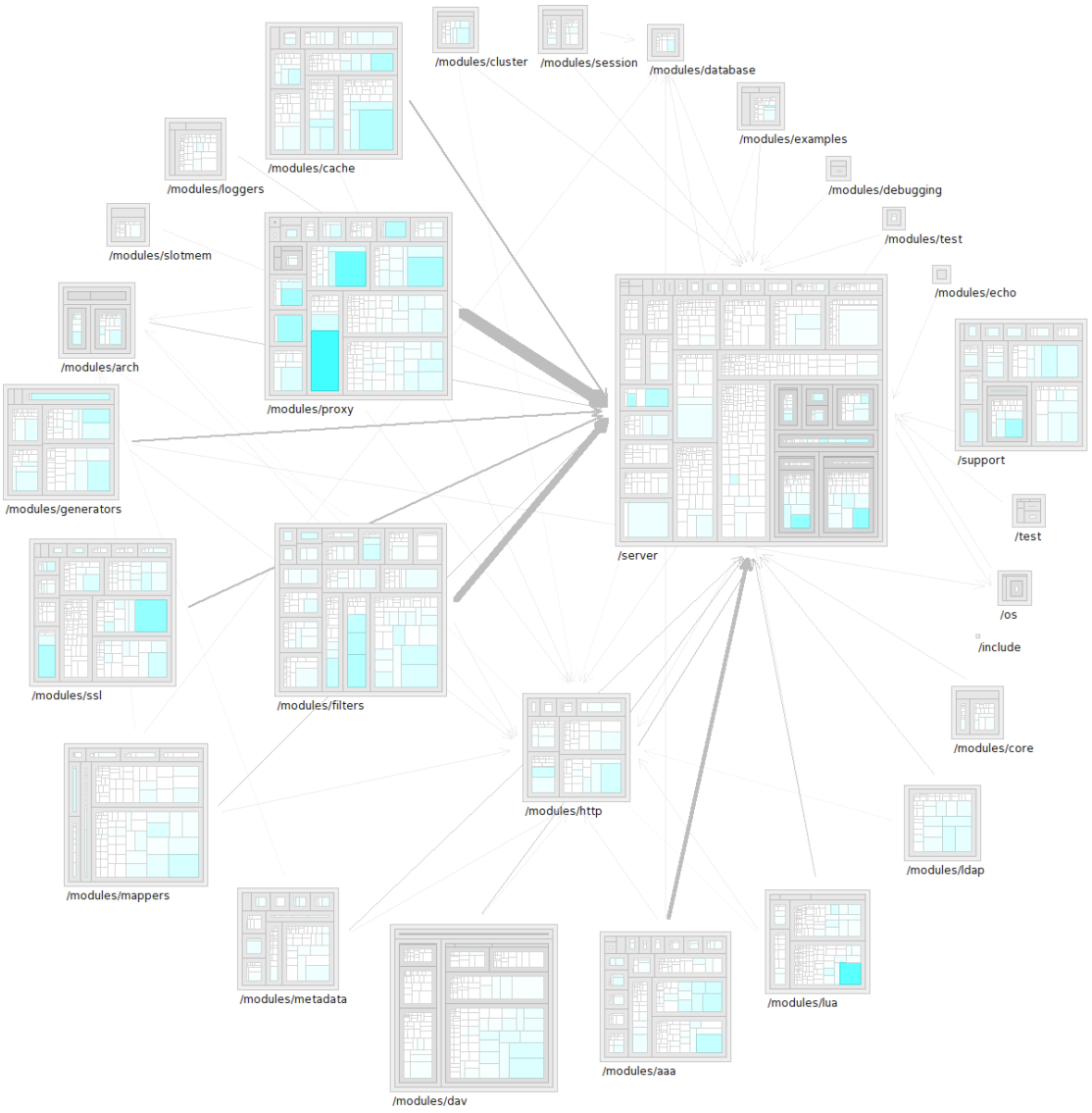

Figure 5.11: httpd graph with outgoing invocations highlighted

6 Case Studies

6.1 Stack Overflow

6.1.1 Introduction

In this section we use Quicksilver to analyze a dataset related to software but not software itself. We want to show how we can use a hierarchical graph to answer multiple questions about Stack Overflow. This section is based on a paper we published in the proceedings of the 2013 International Workshop on Social Software Engineering (SSE'13)[46].

Stack Overflow (SO) is a Q&A web site for programming knowledge, which can be seen as an open market for information in which the buyers are looking for knowledge (*i.e.* answers to their questions) and the sellers are recompensed with score points and badges contributing towards their reputation and power[1] on the market. SO can be viewed as a collective effort to solve common problems of those who participate [44] with motivational drivers very similar to those to be found in scientific and OSS communities [24]. Badges used on SO have also been shown to increase and steer users behavior[5, 20].

SO provides large part of its dataset publicly[2]. As a result, recent years have seen many researchers using it as a case study for a variety of goals: large graph visualization [4], IDE

[1] Different actions on the site require minimum reputation levels

[2] `https://archive.org/details/stackexchange`

improvements based on the information retrieved from SO [7], studying the kind of topics discussed on the website [9], studying the contribution of participants of different ages [38], etc.

In this case study we look at the state of the SO market from a geographical perspective. We are interested in getting a big picture representing SO, without trying to interpret and conclude too much from such a specific snapshot, which would go beyond the scope of this case study. To achieve such an overview we aggregate information from the individual user level to higher geographical levels. To be able to do such an analysis we need to properly geo-locate the users of the site and then assemble the information to the country and continent levels. This approach is similar to the one of Bird and Nagappan who localize the contributions of the participants to Eclipse and Mozzila projects [12].

The structure of this case study is the following: in Section 6.1.2 we present the details of the dataset we use and the geo-location approach and in sections 6.1.3 and 6.1.4 we present some of the observations that follow from our analysis. In section 6.1.5 we discuss some of the threats to the validity of the analysis.

6.1.2 The Data

To perform our analysis we started from the curated dataset provided by Bacchelli [6] which contains all data starting with the inception of the platform in the middle of 2008.

SO allows users to create a profile and optionally specify a location. However, since this information is optional we expect that not all the users will provide it. Many users might not provide their geographical information due to privacy reasons. On top of that, there are many questions and answers to which anonymous or deleted users are attached. Our first questions are thus: *Are those users that provide location information a significant percent of the total user population? How much do they contribute to the total SO knowledge economy?*

By analyzing the data we discover that less than 20% of users (250K users) provide their location information. However, many users create accounts that they never use so maybe many of these users also do not bother to provide detailed information in their profile.

In our analysis we consider the *wealth of an entity* in the market as being their SO reputation. This wealth can be aggregated from the individual to the geo-political entity, as we will see later. We now look at how much of the wealth is accumulated by the users that are geo-located. We sum up the reputation points for the geo-located users and we discover that 75% of the reputation points are distributed to this minority. For the remainder of this case study we only consider this subset of users. We also only look at question and answer relationships that both have geo-locatable owners.

Observation 1. *The minority of 20% of the users who provide their geo-location information in*

SO collect 75% of the wealth in the market.

We used the Yahoo Geolocation API and enriched the original dataset with the additional geographical information per user. One of the most interesting observations is that we have in this way discovered users from 198 countries, which is more or less the accepted number of countries in the world.

6.1.3 The Big Picture

Figure 6.1 presents a visualization obtained with Quicksilver on the information flows between the countries in the world through SO. Quicksilver is interactive and allows the user to drill down in the graph and search for certain nodes. To mitigate the lack of interaction in the medium of this thesis we have annotated the figure with relevant information where needed. We summarize the construction principles of the visualization in Quicksilver:

- The leaf elements of the hierarchical graph are users who have provided valid locations in their profiles.
- The relationships between users represent answers, starting from the provider and ending at the receiver. The receiver is the one who has posted the corresponding question.
- The users are aggregated up into geo-political units; this aggregation propagates the score points given for the answers of the users to the level of countries, and continents.
- The area of the geo-political units is proportional to the aggregated answer score points; the larger an area, the more score points it represents.
- The arrows represent information flow, answers, aggregated form the user level; the thicker and darker an arrow, the more answers it represents.
- The darker the color of a surface, the higher the average received answer scores of the users in that region.

The first thing that strikes the eye is that the SO conversation is global although most of the information exchange happens between Europe and North America, and most of the score points are also cumulated in these two continents. Both continents have similar scores and exchange similar amounts of information.

The United States and Canada are the only countries visible in North America. Although the countries of Central America are also included in the graph their aggregated answer score is barely visible.

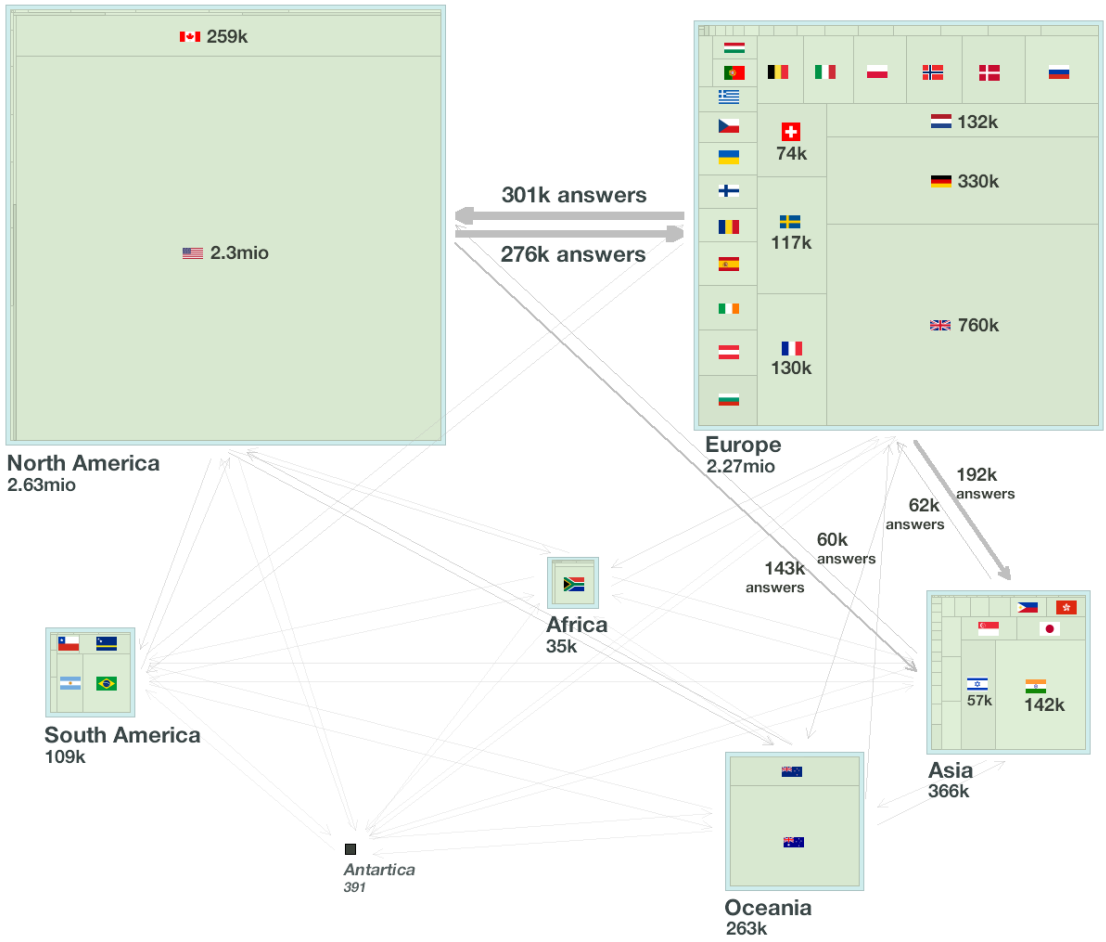

Figure 6.1: Information flow in SO aggregated to the continent level. Size and numbers for continents and countries represent the accumulated scores received by their users for answers given.

In Europe, although in principle all Union countries are visible and contribute, the contributions for Germany and UK add up to about as much as all the other European countries combined. However, the reason for the UK contributing more to the information exchange on the website might be related to the fact that the language of the website is English.

Observation 2. *The information exchange in SO is global. However North America and Europe are the main contributors to the knowledge base on the website.*

India has the most answer score accumulated in Asia, followed by Israel and Japan. This particular ranking might reflect the affinity of these countries to the English language although this would remain to be verified. Overall however, Asia is a disproportionately large importer of

knowledge with respect to its exports. One interesting observation in Asia is the higher average reputation of the users in Israel.

Observation 3. *Asia, Oceania and South America contribute much less than Europe and America to the global discussion happening in SO.*

We were curious about the participation of Antarctica. The idea of people coding in the loneliness of the continent and using SO intensely (they have the highest questions and answers numbers per participant on average) was enticing. After investigating the nine users who had put Antarctica as their location we concluded that most had probably provided misleading location information[3]

6.1.4 Evolution of the market

One aspect that the previous analysis does not answer is the evolution of the SO market over the years. How did it spread, was it a global phenomenon from the beginning? Figure 6.2 visualizes the evolution of the user activity from the beginning of the website.

The first image on top represents the interval between July to September 2008, the time at which the site was in a private beta stage. Even at this stage the global distribution of participants and flow of information as we have seen in the big picture is already, with some variance, established and it will not change much over the years.

Observation 4. *The overall distribution and of flow of information in the market stayed with little variance the same since SO went public, it grew and spread evenly.*

When the site went public it had about 5k geo-located participants. The number of participants almost tripled until the end of 2008. At the end of 2010 there were already 80k participants active and at end of August 2012 there were 250k.

Oceania has a total accumulated answer score of 130k at the end 2009 while Asia had 97k. An interesting development is how Asia grew bigger and surpassed Oceania in the following years. While Oceania had more than double the number of answers per participants than Asia at that time, Asia had more questions per participant. This trend then continued until the end of 2012: Although Asia had only a fourth of the answer scores per participant compared to Oceania, it had more than five times the number of participants than Oceania and thus surpassed it in total answer score (366k vs. 263k).

[3] We tried to contact all the users and two wrote back to us: One user is located in the USA, the other in Asia.

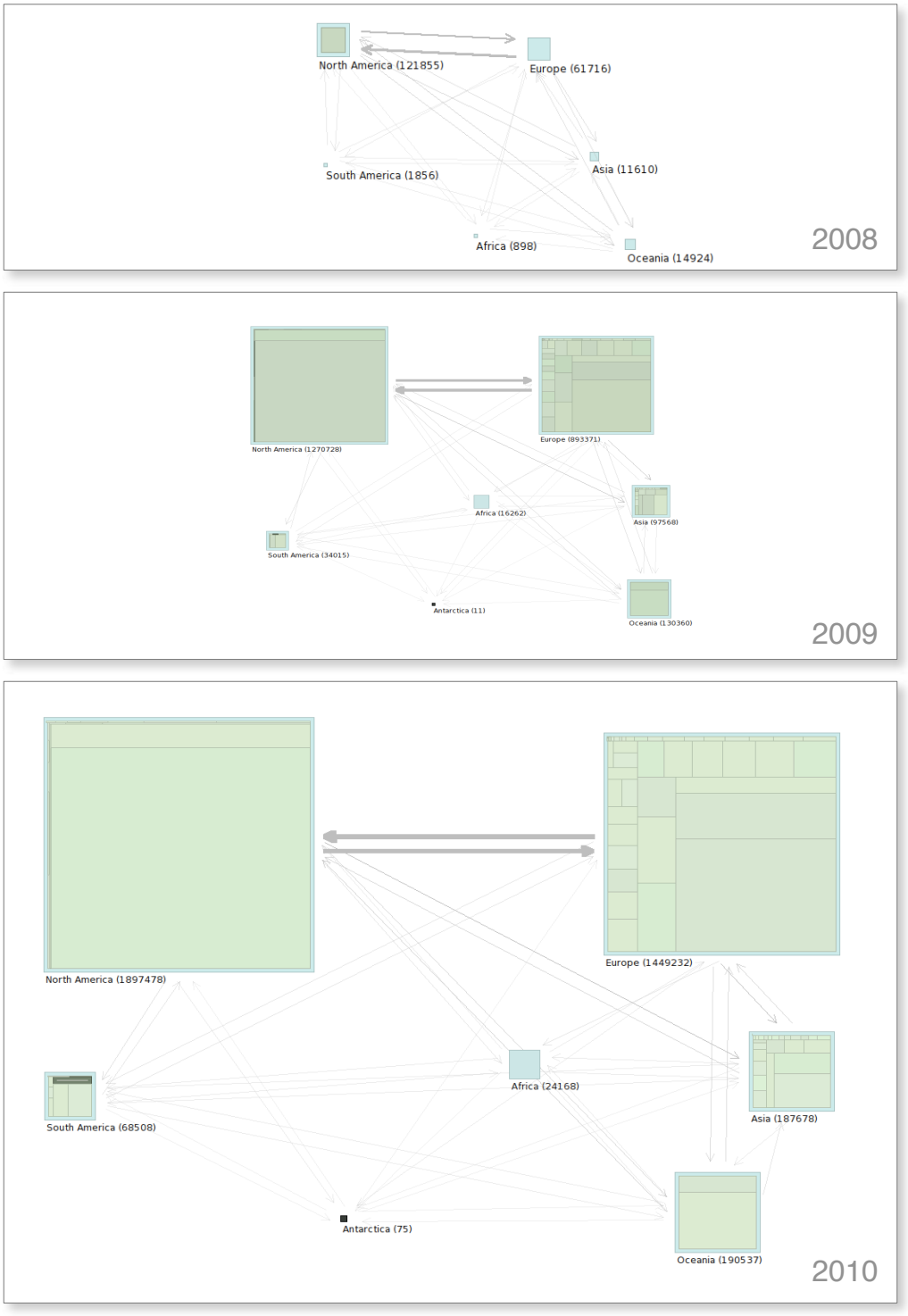

Figure 6.2: Three moments in the history of SO: (top) at the end of the private beta in mid September 2008; (middle) at the end of 2009, and (bottom) at the end of 2010. One sees that the overall configuration of the distribution of reputation and flow of information in SO shows little variance over the years.

6.1.5 Discussion

For simplicity in this case study we often referred to a country importing or exporting information. However, one must not forget the context in which these terms are used. All the assertions in this case study must be understood to hold only in the context of the SO dataset, and particularly only that part of the data that involves users that have provided a location in their profile.

User Location Although these users represent only a fraction, a little less than 20% of the total SO user population, this threat is alleviated by the fact, that they accumulate around 75% of all answer scores in the SO knowledge economy.

The user location is the one specified at the moment when the dataset was created, so if the user has moved recently, the previous answers that he gave, while he was living somewhere else, are wrongly associated with his latest location.

We have no way of detecting users that have declared false locations so we assume that not too many of them will provide a false information, especially since publishing the location of a user is not a privacy threat.

Locations can be ambiguous, *e.g.* there is a Kingston in Jamaica, Canada and the USA. We decided to just use the first one returned by the API.

User Language SO is a platform where knowledge is traded in English, so the countries that have English as their native language might have an advantage. Although software development is an activity that happens naturally in English, observing Germany which has half the reputation of UK although a slightly higher population seems to support the concerns regarding this threat. It would be interesting to study whether users from countries with less affinity to English use alternative platforms although this is unlikely[4].

Traffic Data We correlate the results presented in this case study with information about the websites traffic. Indeed, many users that do not ask or answer questions are nevertheless using the website to find information. In a post from 2011, Joel Spolsky, one of the founders of SO, reports on the demographics of SO visits normalized to the number of users in a country[5]. The top seven countries at the time of his analysis were: Sweeden, Singapore, Finland, Denmark, Israel, Switzerland, and the Netherlands. Note that this top is normalized to the population, so what it tells us is that a higher percentage of those countries populations might be programmers.

[4] In a discussion on a German forum (`http://goo.gl/DMLcR`) a user searching for a German alternative to SO is directed to a small localized SO clone while advised to stick to English in programming matters

[5] `http://blog.stackoverflow.com/2011/04/stack-overflow-around-the-world/`

6.1.6 Conclusion

In this case study we have appraised the state of the knowledge economy in SO by aggregating the individual knowledge transactions to the geographical level. We have discovered that the users that contribute the large majority of the knowledge and collect the large majority of reputation care to provide their own location information. We have observed that Europe and the United States are the strongest contributors in a discourse that involves all the countries in the world. We have observed that Asia is a strong importer of information. Finally by analyzing the evolution in time of the geo-located answers we learned that SO started as a global phenomenon right from the beginning.

6.2 Analyzing PL/1 Ecosystems with Quicksilver

Quicksilver has been used by Aeschlimann to analyze a large legacy software ecosystem[1, 2].

The case study's ecosystem has grown over 40 years to support the business needs of a large banking company. It consists of hundreds of applications written in several programming languages. The focus of the analysis was only the PL/1 part though, but this part alone consists of 30 million lines of code (LOC) before preprocessing and 130 million LOC after.

The tool created to analyze the ecosystem is called St1-PL/1, which parses the code for association data and computes structural metrics. It is based on Quicksilver and its hierarchical graph algorithm and uses its visualization engine.

Before the tool was build, interviews with various stakeholders (domain architects, solution architects, requirement engineers, software engineers and software testers) of the ecosystem were conducted. These free form interviews led to the following initial requirements:

1. Produce high-level views of the entire code base
2. Provide code quality information
3. Support ecosystem restructuring

Consequently the goal of the analysis was to find the means to satisfy these requirements. The first step was to find the right questions and how to answer them.

For the first requirement, interviewees provided various questions: Some architects asked for a way to discuss high-level overviews of multiple domains, some engineers asked for high-level diagrams for documentation purposes and some testers asked for overviews of which parts of the systems need intensified testing,

These questions can be answered by a custom hierarchical graph and its visualization. The ecosystem is already inherently hierarchical. The four first levels of hierarchy are: domains, subdomains, applications and programs.

6.2.1 Ecosystem Overview

A high-level overview over the whole ecosystem can be seen in Figure 6.3. The figure helps in getting a feeling for the ecosystem at first glance. The domains vary in size significantly from the smallest (*N25*), which has 17k LOC, to the largest (*N16*), encompassing 4.6 million LOC. Inside the domains, the applications are visible as clusters of programs. Some domains contain only one large application (*e.g. N24*) while others contain a large number of applications (*e.g. N16*).

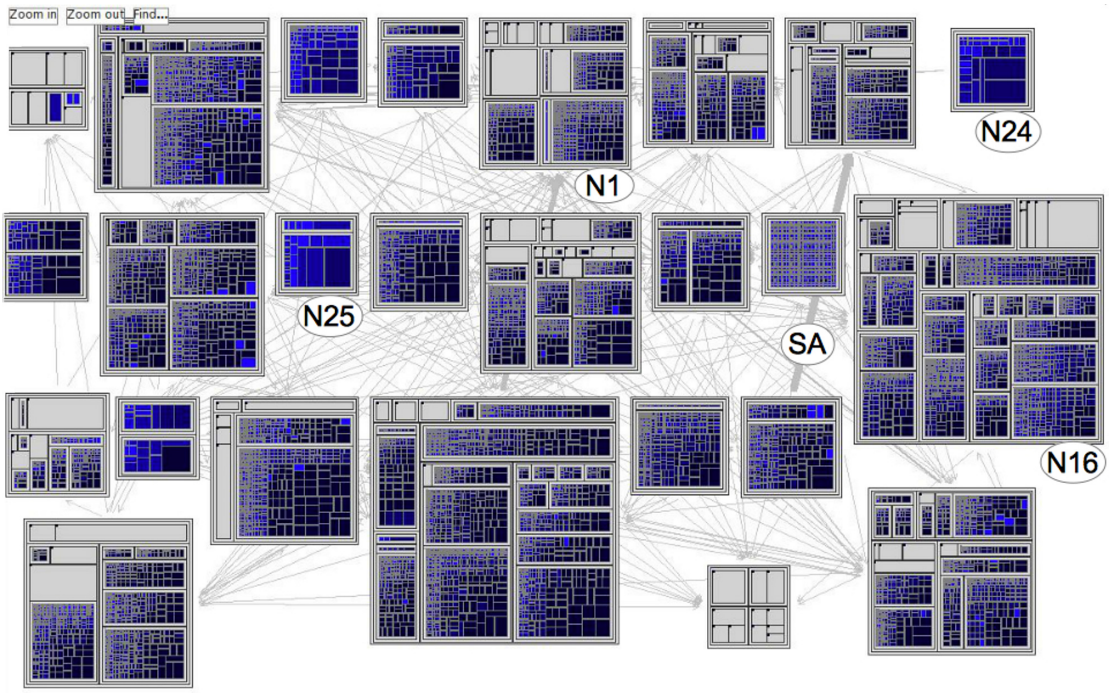

Figure 6.3: Visualization of the whole PL/1 ecosystem showing call relationships between domains and highlighting function points

6.2.2 Ecosystem Coupling

The second requirement is about helping stakeholders make decisions about refactoring efforts. One aspect that is important for such decisions is coupling. Figure 6.3 shows a large number of dependencies between the domains. The number of dependencies per domain varies widely, from

domain *N24* which is completely isolated to domain *N1* which is strongly connected to the rest of the ecosystem. The tools allows one to inspect single dependencies one might be interested in, e.g. dependency *SA*, which is the largest one. By inspecting one can learn the reason for its existence: does it exist because of many calls to a narrow API or due to a large palette of exposed functions?

6.2.3 Ecosystem Quality

The third requirement is about code quality. An example question the stakeholders asked was to detect where *GOTO* statements were used in the code. These statements are allowed in PL/1 itself, but highly discouraged by guidelines. In Figure 6.4 we see how Quicksilver allows one to answer this question. In the figure we see the domain *N1* with its children (sub domains) expanded. A simple highlighter can be used: color every program red, that uses a *GOTO* statement. The domain marked *G* in the top-right corner seems to be a candidate for further inspection.

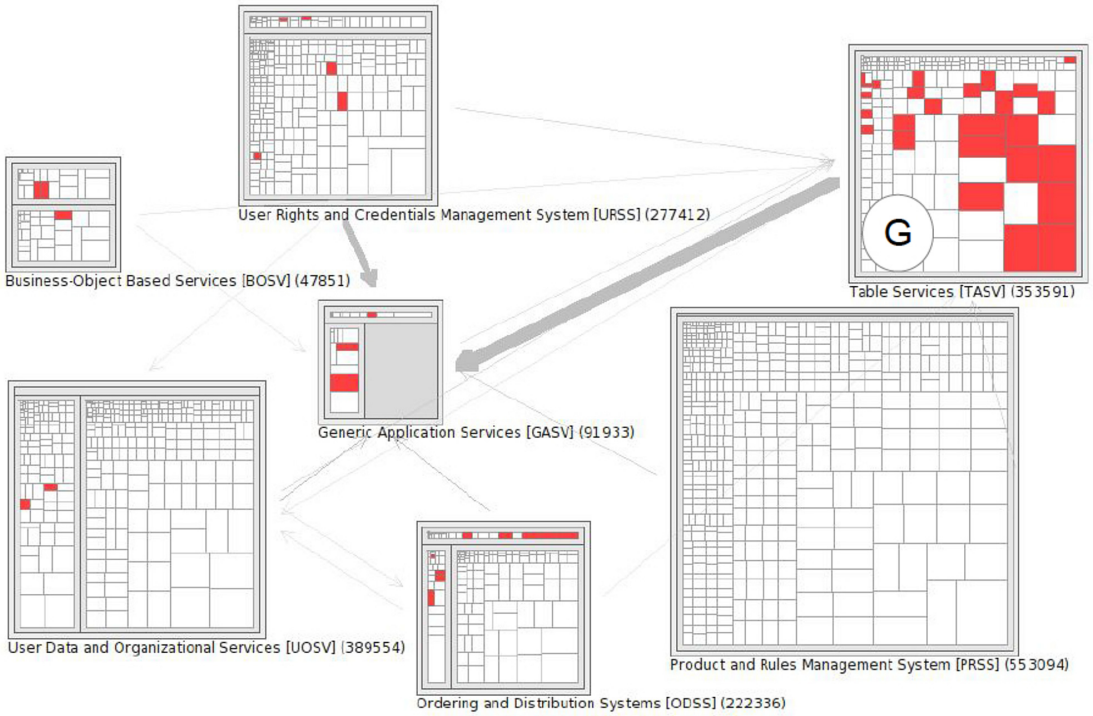

Figure 6.4: Visualizing the usage of GOTO inside a domain

6.2.4 Discussion

The full results of the analysis were presented to the stakeholders in another round of interviews. They reported seeing benefits in starting with a global view of the domains and the possibility of zooming in to show parts of the ecosystem at the sub domain and application level. The tool enabled them to find answer to their questions. Seeing the tool in action also led to new insights into what is possible and desirable and new requirements and questions were posed.

The study showed that it is possible to build custom tools using Quicksilvers infrastructure for an arbitrary domain. The underlying hierarchical data model and the tool support on top of it allows one to explore questions, provide answers and leads to further discussion.

7 Conclusions

In this thesis we have presented a theory and model for arbitrary hierarchical data. We abstracted and identified containment and relationships to be fundamental properties that can be leveraged. We used proven concepts and techniques from software analysis and adapted and generalized them to facilitate exploration and analysis of such hierarchical data.

7.1 Contributions

The main contributions of this thesis are as follows:

1. We gave an overview of software analysis tools and the concepts and techniques used in recovering and visualizing architecture of software systems
2. We provided and motivated a supporting theory and model for generic hierarchical data analysis
3. We presented the HDA framework based on this model
4. We introduced a proof-of-concept implementation based on this framework called Quicksilver

5. We validated our model and the HDA framework by providing two case studies and some additional examples

Besides the two case studies presented in Chapter 6, the work in this thesis has been the foundation for two scientific papers:

- D. Schenk and M. Lungu. Geo-locating the knowledge transfer in stackoverflow. In *Proceedings of the 2013 International Workshop on Social Software Engineering*, SSE 2013, pages 2124, New York, NY, USA, 2013. ACM.
- E.Aeschlimann, M.Lungu, O.Nierstrasz, and C.Worms. Analyzing PL/1 legacy ecosystems: An experience report. In *Reverse Engineering (WCRE), 2013 20th Working Conference on Reverse Engineering*, pages 441448, Oct 2013

The implementation of Quicksilver is available at `http://smalltalkhub.com/#!/~Quicksilver/Quicksilver/`. Appendix A provides more information on how to install and get started with Quicksilver while Appendix B provides a detailed API documentation.

7.2 Future Work

Further research could improve Quicksilver in a number of ways:

Scalability for big data

It would be interesting to further investigate the scalability of the HDA framework, *e.g.* looking at how very large hierarchical graphs could be built and used in a distributed way.

Complete User Interface

One of the main challenges lies in how to present information to the user, how to guide and help him find what is important for him to answer the questions he has. The user interface of Quicksilver is not yet fully-fledged. This is due to the fact that the main focus of the proof-of-concept implementation was the API that allows users to leverage the hierarchical graph. There should be multiple views to help in navigating the hierarchical graph and its visualization, *e.g.* a supporting view of the hierarchy as a tree, with the current working set of nodes highlighted. Such a view would help users orient themselves. More contextual information on screen should be provided *e.g.* sizes of nodes and other visual cues should be linked to metric numbers, so users know how to correlate the visualizations better.

Backbone Layout

There are various areas in which the current implementation of the backbone layout could

be improved. This is the layout which is currently used to display an arbitrary set of nodes of a hierarchical graph. E.g. the current heuristic for initial placement of nodes could be refined.

Layering

The GUI component of the tool should be further improved. To remove visual complexity we propose to add a layering technique to the layouts. Each layer would contain a certain set of node and/or edges, determined manually or by a certain metric, and each layer could be shown or hidden, displayed with a certain opacity or highlighted in some way. This would allow for another level of cross section of the currently displayed set of nodes.

Semantic Zooming

When zooming in and out of the visualization, artifacts shown should not just be resized, but certain elements should be shown in more or less detail, *e.g.* with more or fewer children elements.

Hierarchical Graph Repository

Like the Global View Repository (GVR) of Softwarenaut[1], a repository of hierarchical graphs, for example based on the Pangea Corpus[2], could be implemented. It would allow one to compare hierarchical graphs of data with each other or look at different historical versions of the same datasets.

Roassal 2

The visualization component is currently based on Roassal version 1. Roassal 2 is in development and will be available soon. The visualization component should be switched to use the new version. It would improve performance and its other improvements could be leveraged.

[1] `http://scg.unibe.ch/softwarenaut/book/collaboration/global-view-repository`
[2] http://scg.unibe.ch/research/pangea

Bibliography

[1] E. Aeschlimann. St1-PL/1: Extracting quality information from PL/1 legacy ecosystems. Masters thesis, University of Bern, Dec. 2013.

[2] E. Aeschlimann, M. Lungu, O. Nierstrasz, and C. Worms. Analyzing PL/1 legacy ecosystems: An experience report. In *Reverse Engineering (WCRE), 2013 20th Working Conference on Reverse Engineering*, pages 441–448, Oct 2013.

[3] P. Aiken. *Data reverse engineering: Slaying the legacy dragon*. McGraw-Hill New York, 1996.

[4] L. Akoglu, D. H. Chau, U. Kang, D. Koutra, and C. Faloutsos. Opavion: mining and visualization in large graphs. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 717–720. ACM, 2012.

[5] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec. Steering user behavior with badges. In *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pages 95–106, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[6] A. Bacchelli. Mining challenge 2013: Stack overflow. In *The 10th Working Conference on Mining Software Repositories*, page to appear, 2013.

[7] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing Stack Overflow for the IDE. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 26–30. IEEE, June 2012.

[8] M. Balzer and O. Deussen. Voronoi treemaps. In *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, INFOVIS '05, pages 7–, Washington, DC, USA, 2005. IEEE Computer Society.

[9] A. Barua, S. W. Thomas, and A. E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, page To appear, 2012.

[10] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[11] B. B. Bederson, B. Shneiderman, M. Wattenberg, and D. J. S. Com. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Transactions on Graphics (TOG*, pages 833–854, 2002.

[12] C. Bird and N. Nagappan. Who? Where? What? Examining distributed development in two large open source projects. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 237–246, 2012.

[13] M. Bruls, K. Huizing, and J. van Wijk. Squarified treemaps. In *In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Press, 1999.

[14] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[15] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[16] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 - the FAMOOS information exchange model, 2001.

[17] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. *Moose: a Collaborative and Extensible Reengineering Environment*, pages 55 – 71. RCOST / Software Technology Series. Franco Angeli, 2005.

[18] J.-M. Favre. g see: A generic software exploration environment. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 233–244. IEEE, 2001.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] S. Grant and B. Betts. Encouraging user behaviour with achievements: an empirical study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 65–68, Piscataway, NJ, USA, 2013. IEEE Press.

[21] O. Grossman and D. Harel. On the algorithmics of higraphs. Technical report, Citeseer, 1997.

[22] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to care tools. In *Reverse engineering*, pages 9–45. Springer, 1996.

[23] D. Harel. On visual formalisms. *CACM*, 31(5):514–530, May 1988.

[24] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy*, 32(7):1159 – 1177, 2003. Open Source Software Development.

[25] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley Professional, 2000.

[26] M. Jazayeri. *On architectural stability and evolution*. Springer, 2002.

[27] L.-R. Jen and Y.-J. Lee. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Std 1471-2000*, pages i–23, 2000.

[28] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[29] R. Kazman and J. Carriere. Rapid prototyping of information visualizations using VANISH. In *Proceedings of the 1996 IEEE Symposium on Information Visualization (INFOVIS '96)*, INFOVIS '96, pages 21–, Washington, DC, USA, 1996. IEEE Computer Society.

[30] R. L. Krikhaar. *Software architecture reconstruction*. 1999.

[31] P. B. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[32] M. Lanza. CodeCrawler — a lightweight software visualization tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE CS Press, 2003.

[33] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *Software Engineering, IEEE Transactions on*, 29(9):782–795, 2003.

[34] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: Experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 47–ff, New York, NY, USA, 2003. ACM.

[35] M. Lungu, A. Kuhn, T. Girba, and M. Lanza. Interactive exploration of semantic clusters. *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 0:26, 2005.

[36] M. Lungu, M. Lanza, and O. Nierstrasz. Evolutionary and collaborative software architecture recovery with softwarenaut. *Sci. Comput. Program.*, 79:204–223, Jan. 2014.

[37] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: an agile information visualization framework. In *Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144. ACM, 2006.

[38] P. Morrison and E. Murphy-Hill. Is programming knowledge related to age? an exploration of stack overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 69–72, Piscataway, NJ, USA, 2013. IEEE Press.

[39] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 47–60, New York, NY, USA, 2000. ACM.

[40] H. A. Müller and K. Klashinsky. Rigi — a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[41] K. Onak and A. Sidiropoulos. Circular partitions with applications to visualization and embeddings. In *Proceedings of the Twenty-fourth Annual Symposium on Computational Geometry*, SCG '08, pages 28–37, New York, NY, USA, 2008. ACM.

[42] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[43] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 137–148. IEEE, 2007.

[44] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[45] M. Rekoff. On reverse engineering. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-15(2):244–252, March 1985.

[46] D. Schenk and M. Lungu. Geo-locating the knowledge transfer in StackOverflow. In *Proceedings of the 2013 International Workshop on Social Software Engineering*, SSE 2013, pages 21–24, New York, NY, USA, 2013. ACM.

[47] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, Jan. 1992.

[48] B. Shneiderman, M. Wattenberg, and D. J. S. Com. Ordered treemap layouts. In *In Infovis01*, pages 73–78, 2001.

[49] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[50] J. J. Van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, INFOVIS '99, pages 73–, Washington, DC, USA, 1999. IEEE Computer Society.

[51] R. Vliegen, J. J. van Wijk, and E.-J. van der Linden. Visualizing business data with generalized treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):789–796, Sept. 2006.

[52] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *In ICSE Companion 08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.

Appendices

A

Quick Start Guide

Quicksilver is implemented in Pharo Smalltalk. To use Quicksilver download the newest version of Pharo from http://www.pharo-project.org and run it. Once it is open, create a new workspace (right-click somewhere in the Pharo window and select “Workspace”). In the workspace type the following:

```
Gofer new
  smalltalkhubUser: 'Quicksilver' project: 'Quicksilver';
  package: 'ConfigurationOfQuicksilver';
  load.
```

Now run the command: right click in the workspace window and select “Do it”.

Then, in the same manner, run the following to load Quicksilver and all its necessary dependencies.

```
ConfigurationOfQuicksilver loadDefault.
```

The installation process will take some minutes. While you wait, why not get a tea, relax, look out the window, ponder the mysteries of life...

After all packages have been installed, a good starting point to get to know Quicksilver is the package *Quicksilver-Examples*. Open up a the System Browser (right-click somewhere in the Pharo window) and search for Quicksilver. Various examples, including all code presented in this

thesis can be found there. The tests located in *Quicksilver-UI-Tests* and *Quicksilver-Core-Tests* are also a good starting point for finding out how the Quicksilver works.

Since the hierarchical graph algorithms have been integrated into Moose, the main code and tests can be found in the packages *Moose-Algos-HierarchicalGraph* and *Moose-Tests-Algos-HierarchicalGraph* respectively.

To display a visualization of a certain system in Pharo we can use *QsSystemViewer*. This class automatically creates a system's overview with some default configuration settings applied to get started. If we would like to visualize Zinc for example, the HTTP foundation framework used in Pharo, we can run the following code:

```
1 QsSystemViewer for: 'Zinc'.
```

QsSystemViewer will first create a *MooseModel* of all packages in the system whose names start with 'Zinc', then create a hierarchical graph, using invocations as edges and finally a visualization that gets displayed. The visualization uses lines of code as weight and two linear highlighters. All methods that have a relatively high amount of comments are shown in blue and all methods that have a relatively high amount of invoked methods are shown in red. The code for this example can be found in *QsSystemViewer #view: levelToShow*.

At first you will see a single treemap of the whole hierarchical graph. This does not yet tell us much about Zinc. Right click on the outer most rectangle and select expand. This will show us the graph split on the first level, meaning showing us all packages of the system and the invocations between them. You might have to move the packages around a bit to get a good overview.

B

API Reference

In this chapter we are going to list and shortly describe the methods used to create, query and visualize hierarchical graphs with Quicksilver. The commonly used methods and classes are mentioned here, for a full overview of the Quicksilver system please consult the code itself. All methods and classes are commented and should be self-explanatory.

B.1 Creating and Querying Hierarchical Graphs

A *MalHierarchicalGraph* is composed of a collection of nodes situated in a hierarchy and two types of edges: (1) edges between the base nodes and (2) edges that are propagated up in the hierarchy.

B.1.1 Creation

Instantiation of a hierarchical graph and adding base entities to it. Base entities should either be leaf nodes of a hierarchy or a root node.

MalHierarchicalGraph	
#with:	Creates an instance of MalHierarchicalGraph and set one or multiple base entities. Parameter with can be a single object or a collection of objects.

Table B.1: API: Creating a hierarchical graph (class based method)

B.1.2 Setting Base Nodes

MalHierarchicalGraph	
#nodes:	Adds given nodes as base entities. Nodes is expected to be a collection.
#node:	Adds given node as base entity. Node can be any object.
#leaves	The same as nodes. Adds given collection as base entities.
#root	The same as node. Adds given object as base entity.

Table B.2: API: Setting base nodes (root or leafs) for a hierarchical graph

B.1.3 Building the Hierarchy

Building the hierarchy starting from the given base entities. Can either be build bottom-up or top-down.

MalHierarchicalGraph	
#bottomUp:	Builds the hierarchy bottom-up starting from the given base entities (the leafs). Parameter can either be a block or an array of class to block or class to method-name associations. If a block or method returns nil, it is assumed that the top, the root entity, has been reached and the hierarchy is complete.
#aggregation:	The same as **#bottomUp**.

Table B.3: API: Building the hierarchy bottom-up

Block based:

```
hg bottomUp: [ :e | e isFile ifTrue: [ e parent ] ].
```

With array of associations with both mentioned possibilities:

```
hg bottomUp: {
    SoUser->[ :e | self countries at: (e location countryCode)] .
    PoliticalArea->#parent
}.
```

MalHierarchicalGraph	
#topDown:	Builds the hierarchy top-down starting from the given base entity (the root). Parameter can either be a block or an array of class to block or class to method-name associations. If a block or method returns nil, it is assumed that the bottom, a leaf entity, has been reached and the hierarchy is complete.
#containment:	The same as **#topDown**.

Table B.4: API: Building the hierarchy top-down

Block based:

```
hg topDown: [ :e | e isDirectory ifTrue: [ e children ] ].
```

With array of associations:

```
hg topDown: {
    SoCountry->#inhabitants.
    SoUser->nil
}.
```

B.1.4 Adding Edges and Edge Propagation

MalHierarchicalGraph	
#edges: from: to:	Declarative method to add and propagate edges in one go.
#addEdges: from: to:	Imperative method to add edges. Can be called multiple times, before **#propagateEdges** is called (must be done manually).
#addEdge: from: to:	Imperative method to add a single edge to the graph.
#propagateEdges	Propagates relationships up in the hiGraph from the leaf nodes.
#propagateEdges:	Propagates relationships up in the hiGraph from the leaf nodes. If a collection with classes is provided, only relationships between given classes are propagated.

Table B.5: API: Adding edges & edge propagation

B.1.5 Querying the Hierarchical Graph

MalHierarchicalGraph	
#nodes	Returns an Array of all nodes in the hierarchical graph.
#rootNodes	Returns a collection of all root nodes (*i.e.* all nodes on level 0) in the hierarchical graph.
#leafNodes:	Returns a collection of all leaf nodes in the hierarchical graph. That is, all nodes on the deepest level of the hierarchy.
#nodesUpFromLevel:	Returns a collection of all nodes on given and on higher levels (higher means nearer to the root node.
#nodesOnLevel:	Returns a collection of all nodes on a certain level (a cross cut of the graph)
#nodesDownFromLevel:	Returns a collection of all nodes on given and on deeper levels (deeper meaning farer away from the root nodes).
#findNode:	Returns the node wrapping the given model or nil if such a node does not exists in the graph.
#nodeWrapping:	The same as **#findNode:**
#findNode: ifAbsent:	Returns the node wrapping the given model, runs given block if such a node does not exists in the graph.
#nodeWrapping: ifAbsent:	The same as **#findNode: ifAbsent:**
#nodesFor:	Returns the nodes wrapping the given models or an empty collection if no such nodes could be found in the graph.
#nodesWrapping:	The same as **#nodesFor:**

Table B.6: API: Querying and enumerating nodes of a hierarchical graph

MalHierarchicalGraph	
#edges	Returns a collection of all edges in the hierarchical graph.
#edgesPropagated	Returns a collection of all edges which were propagated.
#edgesNotPropagated	Returns a collection of all edges which were not propagated, *i.e.* the originally added edges.

Table B.7: API: Enumerating edges of a hierarchical graph

MalHierarchicalGraph	
#nodeModels	Returns a collection of all wrapped models in the nodes of the hierarchical graph.
#nodesDict	Returns a dictionary with all wrapped models as keys all nodes as values.
#searchForModelsNamed:	Returns a collection of nodes whose model names match the given RegEx String
#searchForModelsNamed: onLevel:	Returns a collection of nodes on given level whose model names match the given RegEx string

Table B.8: API: Enumerating the models wrapped in a hierarchical graph

MalHierarchicalGraph	
#level:	Returns a collection of all nodes on given level.
#deepestLevel	Returns the deepest level in the hierarchy, *i.e.* the level the leaf nodes reside on.

Table B.9: API: Querying the levels of a hierarchical graph

B.2 Visualizing Hierarchical Graphs

QsVisualizer is a visualization engine to display, explore and interact with MalHierarchicalGraphs. It is built upon Roassal. The visualizer uses tree maps to render single nodes and the hierarchy below them. A weighting block is used to determine how to draw these tree maps.

B.2.1 Creation

QsVisualizer	
#with:	Creates an instance of QsVisualizer with given HiGraph.
#with: weightBlock:	Creates an instance of QsVisualizer with given HiGraph and weighting block.
#show:	Visualizes given hierarchical graph with default weighting block and default visualization configurations.
#show: withWeightBlock:	Visualizes given hierarchical graph with given weighting block and default visualization configurations.

Table B.10: API: Instantiate a hierarchical graph visualizer

B.2.2 Configuration

Before the visualizer renders the hierarchical graph, we can configure it in various ways. We can tell it how to render nodes, which part of the graph to render, with which layout nodes should be arranged, how to colorize certain nodes based on certain attributes, custom interactive elements can be added, etc.

QsVisualizer	
#hiGraph:	Set the hierarchical graph to visualize.
#layout:	Sets the layout that will be used to arrange nodes.
#weight:	Block which instructs the visualizer how to get a nodes weight.
#weightSynthesized:	Allows one to synthesize node weights, meaning if a nodes weight is based on the weight of its children, its weight will automatically be summed from its children weight. E.g. if we would define the synthesized node weight block to be: #numberOfLinesOfCode, we only have to calculate this value for all leaf nodes, then sum these value up in the hierarchy. This prevents us from having to write a weight block that works for all nodes in the hierarchy, *i.e.* calculating #numberOfLinesOfCode for parent nodes by querying their children, thus calculating the same thing over and over.
#averageNodeSize:	Sets the average node size. This value is used as a base line from which to calculate a nodes size, based on the weighting block.
#relativeNodeSizeExtrema:	Defines how much nodes can vary in size based on their weight. Value given is a point: x value is the smallest size factor for the node with the least weight, y value the same for the heaviest node. Factors are relative to the defined node average size. E.g. a nodeSizeExtrema of 0.5@10 and an average node size of 100 would mean the most light nodes would have a size of 50 while the heaviest nodes would have a size of 1000.
#edgeThicknessRange:	Range of edge thickness given as a point. Thickness weight is based on how many edges one graphical edge represents. Value given is a point: x value is the smallest, y value the largest possible thickness. Zero and negative values for the minimum get a thickness of 1 with lower opacity. E.g. -2@10 displays edges from thickness 1 with opacity 0.33 to thickness of 10 and opacity 1. Nodes with weight -1 would be displayed with thickness 1 and opacity 0.66, nodes with weight 0 with a thickness of 1 and an opacity of 1.
#leafBlock:	Block to run on all leaf nodes.
#nodeBlock:	Block to run on all nodes.
#drawLabels:	Whether to draw node labels in the visualization or not.
#nodeLabelBlock:	Block that returns labels for nodes.

Table B.11: API: Configuring and using a hierarchical graph visualizer

B.2.3 Highlighting & Colorizers

The visualizer allows one to highlight certain nodes in certain colors. It also offers the ability to colorize nodes linearly, meaning the higher their value compared to all selected nodes maximum and minimum values are, the more opaque the color used.

Several absolute and/or linear colorizer can be added to a visualizer. Their resulting colors will get combined and mixed on nodes that are affected by multiple colorizers.

QsVisualizer	
#colorize	Returns an instance of a colorizer factory which can be supplied with instructions on how to colorize what in which fashion. See the following examples, which are shortcuts that use this method.
#highlight: color:	Highlights all nodes that get selected with given block with given color. Instead of a block, a class can also be supplied, thus highlighting all nodes which wrap a model of given class.
#absolute selection: color:	Returns an absolute colorizer which colorizer the given selection (block or class based) with the given color. The method above is a shortcut for this one.
#linear selection: value: color:	Returns a linear colorizer which colorizes the given selection (block or class based) linearly with the given color. The colorizer first finds the minimal and maximal values for all selected nodes, then colorizes them more or less opaque in the given color based on the given value.

Table B.12: API: Highlighting and colorizing

B.2.4 Showing Visualizations

QsVisualizer	
#show	Opens a visual representation of the whole graph on level 0, thus showing all root nodes.
#show:	Opens a visual representation of a hierarchal graph but only of the given node collection. This allows one to create visualization of any sub-part of the graph.
#showLevel:	Opens a view of the higraph split at specified level: showing all nodes on that levels.
#showAsTree	Shows the graphs hierarchy as a tree.

Table B.13: API: Showing visualizations

B.3 Hierarchical Graph Nodes

A MalHgNode is a node inside a hierarchical graph. It knows the graph it belongs to, its children and its parent, on which level in the graph it resides and its outgoing ind incoming edges. One can also store arbitrary information as attributes to a node.

B.3.1 Creation

MalHgNode	
#new	Creates a new instance of a MalHgNode.
#with:	Creates a new instance of a MalHgNode with given object as model.
#in: with:	Creates a new instance of a MalHgNode for given hiGraph, with given object as model.
#in: with: level:	Creates a new instance of a MalHgNode for given hiGraph, with given object as model, on given level.

Table B.14: API: Instantiation of hierarchical graph nodes.

B.3.2 Querying

MalHgNode	
#parent	Returns the parent of this node. If node returns nil, it is a root node.
#parentOnLevel:	Returns the node that is on given level in this nodes ancestry. Returns nil if this node is on the same or on a deeper level than given one or if given level does not exist in the graph (*i.e.* greater level than leaf nodes).
#recursiveChildren	Returns the whole family tree below this node (excluding itself).
#recursiveChildren:	Returns the whole family tree below this node. Will adds itself to the returning collection if given parameter is true.

Table B.15: API: Enumerating kinship

MalHgNode	
#incomingEdges:	Returns a collection of of all incoming edges that are on given level. Returns an empty collection if there are no incoming edges. Returns all incoming edges if given level is nil
#incomingFrom:	Returns all edges that are incoming from given node, nil if no such edge was found.
#incomingFromThese:	Returns all edges that are incoming from given node collection, empty collection if no such edge was found.
#incomingNotPropagated	Returns all incoming edges that were not propagated.
#incomingPropagated	Returns all incoming edges that were propagated.
#incomingSources	Returns a set of all source nodes of all incoming edges to this node. Returns an empty set if there are no incoming edges.
#incomingSources:	Returns a collection of all source nodes of all incoming edges that are on given level. Returns an empty collection if there are no incoming edges. Returns all incoming sources if given level is nil

Table B.16: API: Enumerating incoming edges

MalHgNode	
#outgoingDestinations	Returns a set of all destination nodes of all outgoing edge target nodes. Returns an empty set if there are no such nodes.
#outgoingDestinations:	Returns a collection of all destination nodes of all outgoing relationship that are on given level. Returns an empty collection if there are no such nodes. Returns all outgoing destinations if given level is nil.
#outgoingEdges:	Returns a collection of all of all outgoing relationship that are on given level. Returns an empty collection if there are no such relationships. Returns all outgoing relationships if given level is nil.
#outgoingNotPropagated	Returns all outgoing relationships that were not propagated.
#outgoingPropagated	Returns all outgoing relationships that were propagated.
#outgoingTo:	Returns all relationships that are outgoing to given node, empty collection if no such relationship was found.
#outgoingToThese:	Returns all relationships that are outgoing to given one or more of given node collection, empty collection if no such relationship was found.

Table B.17: API: Enumerating outgoing edges