

Diplomarbeit

October 2, 2006

KiviatNavigator

Navigation of Source Code Data
using Kiviat-Graphs

Roman Flückiger

of Olten, Switzerland (01-703-578)

supervised by

Prof. Dr. Harald Gall

Dr. Martin Pinzger



University of Zurich
Department of Informatics



Diplomarbeit

KiviatNavigator

Navigation of Source Code Data
using Kiviat-Graphs

Roman Flückiger



University of Zurich
Department of Informatics



Diplomarbeit

Author: Roman Flückiger, r.fluckiger@access.unizh.ch

Project period: 3. April 2006 - 3. Oktober 2006

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

First and foremost, I would like to thank Martin Pinzger for his imperturbable calm and his reassurances during the dark hours of scientific research.

Next, where would I be without the tenacious companionship of Michael Würsch and Andreas Jetter through all these days. Your good spirits and helpfulness is unheard of. Thanks, you two.

Special thanks to Beat Fluri for his both gracious and generous support in my hour of need.

Further thanks to Patrick Knab for postgresSQL support and constantly taking away the Eclipse book — I have my own copy now.

Additional thanks to my parents who, to my constant surprise, never seem to loose faith in me...

And finally, thanks to my sister, Simone, for proof-reading my work.

Abstract

Source code data of large software systems tend to be very complex. To visualize and navigate these data pools, in a manner to reveal specific software traits, remains a challenge to date.

In this thesis we present an exploration strategy for navigating such source code data. We generate graphical views that expose specific design aspects, such as bad smells, and hotspots in general. The approach uses sequences of such views to incrementally gather knowledge about the code in scope. This finally allows us to identify entities of questionable design.

Our approach uses the *measurement mapping* principle combined with kivi diagrams to visualize system entities. We further present a prototype implementation as an Eclipse plug-in and evaluate it in a case study, analyzing parts of the Mozilla source code.

Zusammenfassung

Programmcodes von grossen Software Systemen tendiert dazu sehr komplex zu werden. Diese Daten zu visualisieren und navigieren, so dass spezifische Charakterzüge des Programmcodes hervorgehoben werden, ist nach wie vor eine Herausforderung.

In dieser Arbeit werden wir eine Strategie zur Erforschung von Programmcodedaten präsentieren. Wir werden graphische Ansichten generieren, die spezifische Designschwächen, wie zum Beispiel "Bad Smells", sowie allgemein verdächtige Strukturen entlarven sollen. Unser Ansatz verwendet Sequenzen solcher Ansichten um inkrementell Wissen über den Programmcodes zu sammeln. Dies erlaubt uns schliesslich Entitäten mit fraglicher Struktur zu identifizieren.

Unser Ansatz verwendet das *Measurement Mapping*-Prinzip, kombiniert mit Kiviat-Diagrammen als Visualisierung von Software-Entitäten. Des weiteren werden wir einen Prototypen als Eclipse-Plugin implementieren und evaluieren. Letzteres mit Hilfe einer Fallstudie, in der wir einen Teil des Mozilla Programmcodes analysieren werden.

Contents

1	Introduction	1
1.1	Contribution	1
1.2	Structure of the Thesis	1
2	Related Work	3
2.1	Polymetric Views	3
2.1.1	CodeCrawler	4
2.1.2	ArchView	4
2.2	Simple Hierarchical Multi-Perspective (SHriMP)	5
3	Approach	7
3.1	Exploring Large Graphs	7
3.2	Incremental Exploration and Navigation	8
3.3	Polymetric Views	9
3.4	Preset Views Concept	10
3.5	Preset View Catalog	11
3.5.1	Provider/Consumer View	11
3.5.2	Roots/Leaves View	13
3.6	Concluding Thoughts	14
4	Implementation	17
4.1	Integration	17
4.2	Requirements	18
4.3	System Setup	18
4.4	Kiviat Navigator Overview	19
4.5	Kiviat Navigator Architecture	20
4.5.1	KiviatContainer and KiviatContainerGenerator	21
4.5.2	IKiviatContainerNormalizer and KiviatMaxNormalizer	21
4.5.3	KiviatNodeRealizer	22
5	Mozilla Case Study	23
5.1	Approach	23
5.2	Investigation	23
5.3	Summary	27
6	Conclusions	29
6.1	Contribution	30
6.2	Outlook	30

A Contents of CD-ROM**31**

List of Figures

2.1	Up to five metric values can be mapped on a CodeCrawler node.	4
2.2	A polymetric view in CodeCrawler (source: [LD03]).	4
2.3	A view from the ArchView approach (source: [Pin05])	5
2.4	On the left, a SHriMP visualization using the fisheye distortion algorithm (source: [SWFM97]). In the center a newer implementation of SHriMP and on the right as view from Creole, as well using SHriMP (source: [Chi06])	6
3.1	Exploration paths as sequences of single viewpoints	8
3.2	A sample kiviati diagram, and how the three dimensions are mapped onto it	9
3.3	The left diagram shows a typical data consumer, the right one a data storage object.	12
3.4	The left diagram shows a typical consumer of functionality. The right graphic depicts a provider.	12
3.5	The left graphic shows a consumer of data and functionality, the right one a provider.	13
3.6	The left graphic shows most probably dead code, the right diagram an entity that is both provider and consumer at the same time.	13
3.7	Leaves of the inheritance tree	14
3.8	Roots of the inheritance tree	14
3.9	Extreme members of the view.	15
4.1	This chart shows how the Kiviati Navigator is embedded into the Eclipse environment and where data comes from. Components in faded grey are not currently used by our implementation, but show the envisioned goal.	17
4.2	The HierarchyView shows all entitites available for visualization	19
4.3	The MetricView allows selection of metrics	19
4.4	This is a simplified graph of the components involved in the Kiviati Navigator. The modules with thicker black borders contain multiple classes, which mostly are members of model-view-control patterns.	20
5.1	A simplified System Hotspot View [Pin05][LD03]. The metrics are the following: (0) imagix_CIMemVar, (1) imagix_CIMemTyp, (2) imagix_CIMemFnc, (3), imagix_CIMemCl, (4) length.	24
5.2	Provider/Consumer View. The metrics are the following: (0) in_invokesnrRelsDirect, (1) in_accessesnrRelsDirect, (2) out_invokesnrRelsDirect, (3), out_accessesnrRelsDirect.	25
5.3	Roots/Leaves View. The metrics are the following: (0) in_overridesnrRelsDirect, (1) in_inheritsnrRelsDirect, (2) out_overridesnrRelsDirect, (3), out_inheritsnrRelsDirect.	27

List of Tables

3.1	Table of Metrics	11
5.1	Table of Findings	28

List of Listings

4.1	IKiviatiContainerGenerator	21
5.1	NameSpaceDecl	25

5.2	Two method examples from nsXMLProcessingInstruction	26
-----	---	----

Chapter 1

Introduction

Software systems tend to get large and complex during their lifetime. They are subject to constant change and extensions in numerous ways. In addition, people leave projects, new developers join the team, documentation gets sloppy, or is never done at all. Sooner or later, there comes a time when the people working on a software system cease to know anything about the tricks and traps hidden within this behemoth, being their work.

This is where reverse engineering comes into play. How can we recover from the raw source code data what we have lost along the way? While there exist concepts how to systematically recover the architecture and structure of a software, our set of mind is a bit more optimistic, since we assume that unknown software systems are not entirely bad. We will focus on how to specifically expose unfortunate structures or bad smells [FBB⁺99].

We are going to do this using the concept of *measurement mapping* as the base of our approach. This will lead to a graphical representation of source code data with so-called kiviats diagrams. How we will navigate the complexity of software systems with the help of the mentioned visualization is the core subject of this thesis.

1.1 Contribution

Our goal is to propose a simple and useful way of navigating source code data using kiviats graphs as presented in the ArchView approach [Pin05]. Thus simplifying the analysis of large and unknown source code data and maybe even make a first step to standardization of such analyses. We will use kiviats graphs to devise views that highlight specific source code aspects, such as bad smells. We will further present a strategy called *incremental exploration* that uses sequences of such views to gather knowledge about a target system. During the thesis a prototype Eclipse plug-in implementing these concepts will be developed. Finally, we evaluate our tool in a case study, analyzing parts of the Mozilla source code, and discuss the results.

1.2 Structure of the Thesis

Chapter 2 presents different concepts of investigating and navigating source code data, considered as related work. In chapter 3 our approach to the problem is presented. Starting with our own thoughts of inspecting large graphs, leading to the principles of *incremental exploration* and finally examples of useful view configurations and their analysis. The subsequent chapter will describe the implementation of our approach, the Kiviats Navigator. Our tool will finally be put to the test in the case study, in chapter 5, where we will apply our methods to a subset of the Mozilla

web browser source code. In the final chapter we will conclude the thesis and give an outlook for future work.

Related Work

In this chapter we review concepts and tools that focus on layout techniques and navigation of source code data, which is a part of information visualization. Before we focus on a few closely related approaches, we will do a fast sweep of to vast field of other interesting work.

First of all, there are visualization techniques that make use of the third dimension. One way of using 3D would be to add the dimension of time to 2D views, a concept presented by [SDB98]. Another approach is given by the source viewer 3D (sv3D) [MMF03], that uses an extension of SeeSoft [ESS92] to represent software systems.

Then there are concept that focus on visualizing runtime information of programs, such as the program explorer [LN95].

Finally, coming back to the frame where our thesis best fits in, there are static visualizations. A taxonomy of software visualization of this kind is given by Price *et al.* [PBS93]. In this class belong approaches like Rigi [MK88], SeeSoft [ESS92], SHriMP [SWFM97] and CodeCrawler [LD03]. We will describe two of these concepts in more detail in the next sections.

We start with an introduction to *polymetric views*, the visualization method that is our primary focus. Next, two software tools, that use *polymetric views* to inspect source code data will be briefly presented, CodeCrawler [LD03] and ArchView [Pin05]. Finally, the SHriMP approach [SWFM97] is presented, and along with it a number of graph navigation techniques, that are being used by this approach.

2.1 Polymetric Views

Polymetric views are the starting point of this thesis. The basic concept our visualization techniques will use to layout source code data and enrich it with information. It is solely intended for object-oriented source code data. A polymetric view is a two dimensional visualization in which nodes represent software entities and edges represent relationships between entities. Furthermore a number of metric measurements are mapped onto nodes (and edges). This methodology is called *measurement mapping*. It should fulfill the representation condition: "if a number a is bigger than a number b , the graphical representation of a and b must preserve this fact" [LD03].

This approach essentially uses metric visualizations to show symptoms of the underlying source code data. The following two tools are both based on this concept.

2.1.1 CodeCrawler

The CodeCrawler is presented by Lanza *et al.* in [LD03]. It uses rectangles as nodes and maps up to five metrics onto a single node. Figure 2.1 shows how this is done. Both width and height represent a measurement, as well as the color of the node body. In some configurations the location of the node in the view is used to map two additional metric values.

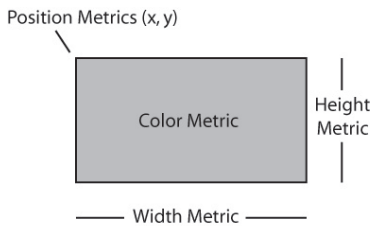


Figure 2.1: Up to five metric values can be mapped on a CodeCrawler node.

The CodeCrawler needs three basic ingredients to generate a polymetric view. A choice of entities and metrics and a third one: a layout. The layout determines how nodes are arranged in a view, for instance if they should be sorted in a specific manner. The list of layout strategies used by the CodeCrawler contains layouts such as tree structures and scatterplots. Figure 2.2 shows a so-called checker distribution of nodes. The entities are sorted according to a specific metric. In this particular case the target entities are attributes. The width and height of the nodes render the number of local accesses and the number of nonlocal accesses, respectively. The color indicates the total number of accesses. In this way, attributes that are never accessed at all, and can therefore be removed, line up in the top row. Attributes, which are heavily accessed are found at the bottom. In addition, attributes that get predominantly nonlocal accesses stand out as very tall nodes — candidates for accessor methods.

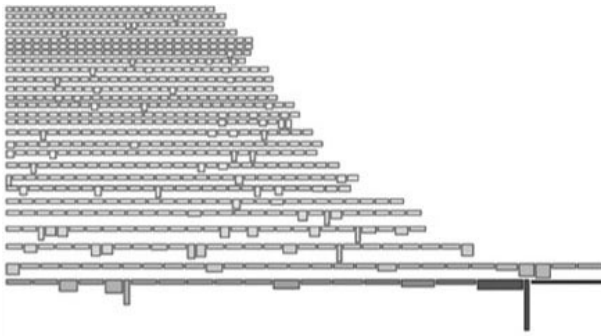


Figure 2.2: A polymetric view in CodeCrawler (source: [LD03]).

2.1.2 ArchView

The ArchView approach by Pinzger [Pin05] is also based on the concept of *polymetric views*. The crucial difference to CodeCrawler is the graphical representation of nodes. Instead of rectangles ArchView uses kivi diagrams (refer to Figure 2.3 for examples of kivi diagrams). The advantage of such a representation is the possibility to map considerably more than five metrics onto a single node simultaneously. A benefit, that is actually seldom used. There are cases where metrics

are all of the same “kind” and kiviati diagrams prove their worth. But we will see, that most of the time around four metrics produce views that have stronger interpretations.

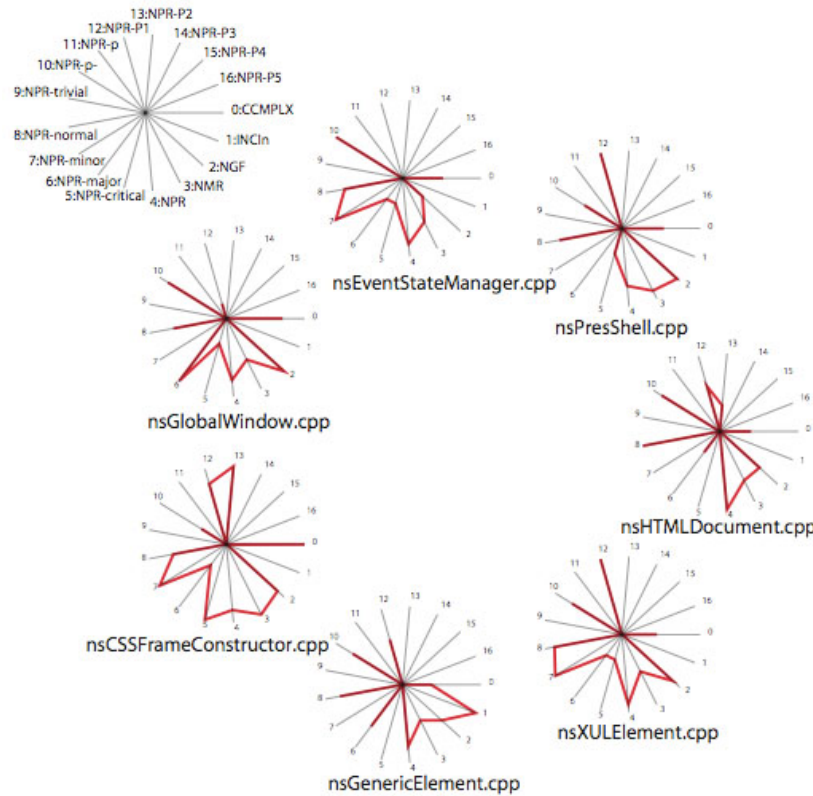


Figure 2.3: A view from the ArchView approach (source: [Pin05])

Figure 2.3 shows a detailed modification hotspots view generated by ArchView. Nodes represent files in this view. The rendered metrics are concerned with problem reports, their priority and severity. This reveals how many problems a file was affected by and how severe these problems were.

ArchView introduces a number of strong changes to *polymetric views* as CodeCrawler uses them. Firstly, there is just one layout in ArchView. The location of the nodes are subject to the user’s wishes and not bound to any metric. Moreover the representation condition of *measurement mapping* is no longer fulfilled by the complete visualization, since metrics are no longer normalized within a node. This means values can no longer be related within a single node, but still between all the diagrams. We will later on see what consequences this fact has for our visualization.

On the other hand, ArchView adds code releases as a new dimension to the concept.

2.2 Simple Hierarchical Multi-Perspective (SHriMP)

An entirely different approach to reverse engineer large source code data is the SHriMP approach [SWFM97]. When graphs reach a certain size and complexity, navigation techniques become necessary to find your way around. Software systems usually are large and complex and therefore

also yield complex graphs. SHriMP has its focus not on symptoms but on traversing hierarchies and relationships of source code. It uses different navigation strategies to accomplish this goal.

To prevent information overflow by displaying a whole system at once, SHriMP uses *semantic zooming* [HMM00] to hide and reveal information of nodes. This zoom is not applied to the whole view at the same time. You can choose which node to enlarge and show details, with the context of the entity is still visible. This is called fisheye distortion. SHriMP even allows you to focus on different parts of the graph at the same time, to inspect disjoint source code entities.

Figure 2.4 shows different implementations of the SHriMP approach.

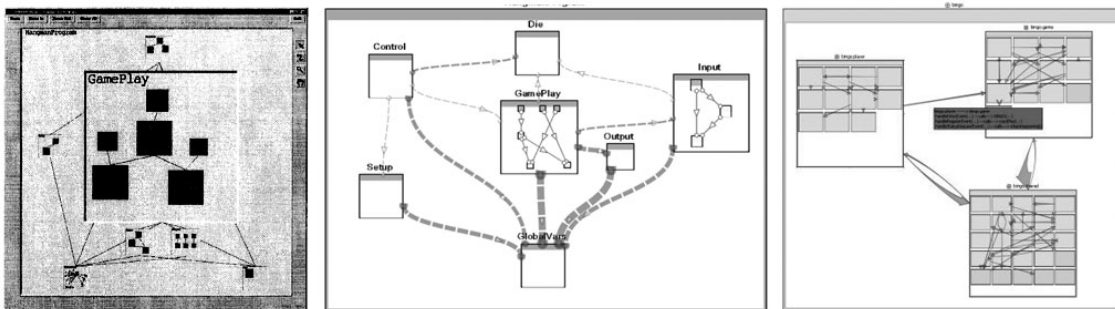


Figure 2.4: On the left, a SHriMP visualization using the fisheye distortion algorithm (source: [SWFM97]). In the center a newer implementation of SHriMP and on the right as view from Creole, as well using SHriMP (source: [Chi06])

Chapter 3

Approach

The previous chapter gave a brief insight into some already existing source code visualization techniques. From the beginning our goal was to take the ArchView [Pin05] approach as our starting point and enhance it while integrating it into the Eclipse platform. The ArchView approach has its focus on visualizing source code metrics, as opposed to SHriMP [SWFM97], that basically visualizes whole tree structures and uses zooming, panning and disjoint context focussing to navigate. The ArchView stand-alone implementation is more of a static visualizer. What is left is our contribution, to find out where navigation fits into this concept.

The question we asked ourselves was, how to navigate such large graphs, like software systems usually are? We found most of the answers in the work of Lanza *et al.* [LD03] and Herman *et al.* [HMM00].

3.1 Exploring Large Graphs

Software systems are usually very large and a single mind has most of the time difficulties to understand it entirely. That is certainly the case, when you look at the whole system at once. Even other researchers, who committed quite some time in solving this problem have not come up with a satisfactory solution. Storey *et al.*, for instance have presented the SHriMP approach [SWFM97] with the incentive to let the user see the whole system all the time, since this increases the understanding of the whole system. They use fisheye distortion methods to let the user zoom in on specific parts of the system without losing the connections to the rest. But their approach is not safe from information overflow in the visualization. Some intelligent filtering will be necessary to make this concept useful.

So here is a thought about the exploration of large spaces. Since their vastness is the biggest obstacle in understanding, why let the user see everything and burden him with the difficult task to mentally (or graphically supported) fade out irrelevant information? Being able to see the whole picture all the time certainly helps building a mental map, but since the entire space is not understood at this time, most mental associations will be useless.

There are other strategies to explore such large networks, by continuously fading relevant information in, instead of fading irrelevant facts out. Let us take the World Wide Web for example [HMM00]. It is impossible to behold the whole network at once, but day by day by searching for specific information a user may explore this vast network and start to build a mental map from it, which is solely built from pieces of relevant information. There is no use to know the entire system if all you want to know can be found by a couple of mouse-clicks, just two or three hyperlinks away.

So let us imagine our target software system like a large three dimensional cloud (refer to Figure 3.1). It is impossible to get a grasp of it all from just one viewpoint. Something will always be hidden from view or covered by some other information. A layout alone can not overcome the challenge of revealing the structure of large spaces [HMM00]. We need some kind of navigation to travel around.

The key is to look at the target system from multiple perspectives or viewpoints and hereby gain information step by step. This approach is called *incremental exploration and navigation* and is also presented in the survey by Herman *et al.* [HMM00].

3.2 Incremental Exploration and Navigation

By gaging a system through a sequence of different viewpoints (look at Figure 3.1 to get a general idea), which is essentially the process of *incremental exploration*, the main question is who chooses the next step in the path and how?

There are two possibilities for the *who*. It is either the user itself, who has all the time complete control over the next step in line, or the computer, that suggests the next view configuration based upon some heuristics, for example. Of course the idea, that there exist specific scenarios of view sequences that lead to the detection of certain code issues is very appealing. It has to be shown if such scenarios emerge during the case study.

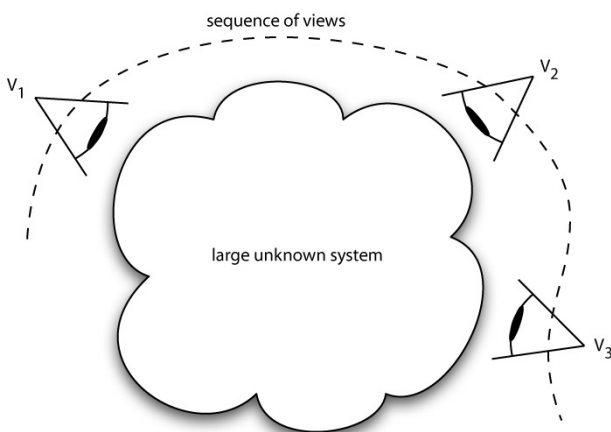


Figure 3.1: Exploration paths as sequences of single viewpoints

This leads us directly to the *how*. The sequence of views is based on constant decisions. From every viewpoint on the system the user gains a bit of information about the graph. At this point the user must make a decision, where to go next. This decision is at the moment solely based upon the scientific knowledge of the user. He must ponder the information at hand and decide what viewpoint on the system would probably add to his pool of current information. The choices are obviously quite numerous and the decisions all but trivial.

What we learn from this is, that although the exploration paths through the graph are to some degree unforeseeable up front, we can help the user by delivering him viewpoints with a purpose. Views, that are designed with the thought in mind, to shed light on specific aspects of software systems. These views can then be chosen from a catalog whenever needed.

Fortunately, albeit not that much of a surprise, the concept of *polymetric views* [LD03] plays nicely with this approach.

3.3 Polymetric Views

In accordance with the approach of *incremental exploration* we need a layout strategy for the realization of our views. We already gained some insight into the concept of *polymetric views* in the respective related works section. Our approach will make use of a polymetric visualization similar to the one presented in ArchView [Pin05]. The space we are exploring has basically three dimensions: entities, metrics, and releases. On top of that is a fourth one, which are relationships. How all these four characteristics are mapped onto a kiviatic diagram is shown in the following.

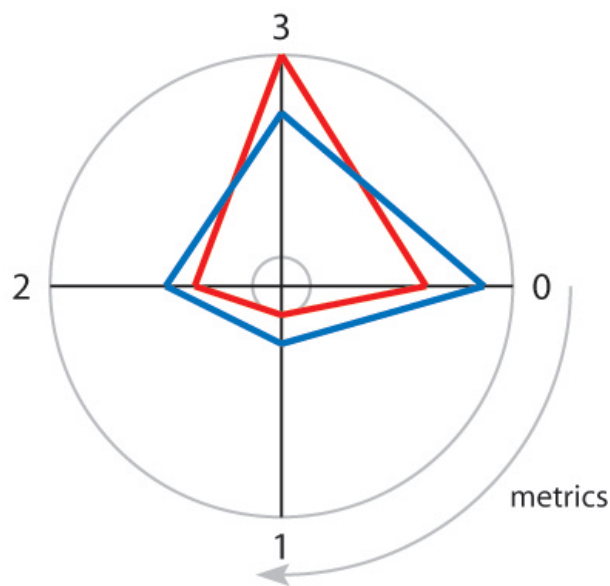


Figure 3.2: A sample kiviatic diagram, and how the three dimensions are mapped onto it

Figure 3.2 shows how a kiviatic diagram might look. One diagram represents an entity, e.g. a class. The spokes — there can be more than four — all show a metric value. Values from different releases are drawn in different colors. Relations between entities are visualized as arcs between diagrams.

Entities. Entities, such as classes are the actual subject of interest of our visualization. All metrics are bound to an entity or relate them in a way. Every view must focus on one or more entities. Of course a view may even show all available entities at the same time.

Metrics. Metrics describe entities. They are used to gain information about their subject through visual filtering. In other words: highlight interesting entities. All kiviatic diagrams shown at the same time are drawn with identical metric configuration. Moreover values are normalized for each metric over all entities. So keep in mind that a maximum value on two different views must not necessarily be the same absolute value. Deductions must be made primarily from visual features and are restricted to one view at a time. Additionally, a minimum value prevents the diagram from cluttering in the center. Values on the smaller circle in the center are therefore to be interpreted as zero values.

Releases. For every metric and entity there might be one or more releases present. Multiple release values can be drawn on the same metric axis in different colors. This enables users to get

a grasp at the change over time an entity has made.

Relationships. A further, in a way different kind of dimension are relations. Exploration can also be achieved by navigating entities through their associations to other subjects. By displaying neighbor entities, small compounds of closely related or tightly coupled entities can be shown and further investigated.

Depending on the data at hand there can be a huge number of metrics available. It is the selection of specific configurations that yield interesting results. It is all about adjusting the parameters of a view in a way to visually highlight entities with interesting qualities. The parameters are, like listed above, the choice of entities, the selection and order of metrics and releases. With this in mind, there is a strong incentive to create or rather design view configurations that have a specific purpose, e.g. revealing especially large entities or unusually strong coupled ones. Such predefined views or presets can then be used to ease navigation of an unknown system.

This can be done in one of two different ways. A user may start with a preset view, a hotspot for instance, search for extremely large classes and then adjust the view settings, adding other metrics or removing some — in thereby generating new views — to find out more about this class and its neighborhood. On the other hand, preset views can be used as part of a utility belt, purposeful instruments that can perform specific tasks when needed. Tasks such as revealing provider or consumer entities.

The upcoming sections goal is just that. Presenting a set of preset views, maybe the first few in a larger catalog to come.

3.4 Preset Views Concept

Preset views help to gain insight into an known or unknown software project in a short time. It is a way of navigation through the code by applying specific preset views to the project to find hotspots. These can then be investigated from another angle (using other views, or by manipulating metrics) or analyzed on source code level.

Part of this thesis is to compose a set of such views, describe their intended usage, and later test them during the case study. It was our intention to reuse some of the views already discussed by Pinzger [Pin05] and Lanza *et al.* [LD03]. However, the pool of metrics available to us is not the same as the one used in those references. Obviously standardized views can only exist with standardized metrics at hand. Therefore the preset views described here are adaptations or new views from scratch.

You will notice that most of the preset views use only four metrics at a time. You might wonder why that is, since kivi diagrams were primarily chosen due to their ability to show more than three or five metrics (the maximum number of metrics used by Lanza's approach). There are specific tasks, where a number greater than five metrics is very useful. But we gathered, that four metrics at a time yield more concise views that enable stronger interpretations.

3.5 Preset View Catalog

The following catalog of preset views follows a specific layout. Every preset starts with a description of its purpose, the source code aspects it tries to reveal. This is followed by the configuration of the view, namely the chosen metrics and their ordering. Note, that the number in brackets represents the index of the metric in the diagram pictures. Finally, characteristic diagram patterns, that help the user identify interesting entities, are described and interpreted.

Note: Keep in mind, when you try to deduce information from a view, that you cannot compare two metrics within a single kivi diagram. You can only compare the same metric over all displayed diagrams. Within one kivi diagram a relative value of 0.5 on a metric axis can be any kind of absolute value, depending on other values of the same metric in other kivi diagrams. It is a common mistake to look at a kivi diagram and make a judgement like: "There are significantly more attributes than methods in this class". Just because, there is a peak in this diagram in the attributes metric, this must not be true. It can even be the other way round, so beware.

This list provides descriptions to all class metrics used in the following preset views.

Table 3.1: Table of Metrics

<i>Metric</i>	<i>Description</i>
in_invokesnrRelsDirect	Number of method invocations to this entity from all other entities
in_accessesnrRelsDirect	Number of field accesses to this entity from all other entities
in_overridesnrRelsDirect	Number of methods of this entity that are overridden by any children
in_inheritsnrRelsDirect	Number of children of this entity
out_invokesnrRelsDirect	Number of methods this entity invokes in any other entity
out_accessesnrRelsDirect	Number of fields this entity accesses in any other entity
out_overridesnrRelsDirect	Number of methods overridden by this entity from any parent
out_inheritsnrRelsDirect	Number of parents of this entity
imagix.CIMemCl	Number of class members (inner classes) of this entity
imagix.CIMemFnc	Number of functions of this entity
imagix.CIMemTyp	Number of type definitions of this entity
imagix.CIMemVar	Number of variables of this entity
length	Total lines of code of this entity

3.5.1 Provider/Consumer View

Aspect. This view distinguishes between entities that predominantly provide functionality or consume it. Thus, library classes or typical data storage objects can be revealed.

Configuration.

- (0) in_invokesnrRelsDirect
- (1) in_accessesnrRelsDirect
- (2) out_invokesnrRelsDirect
- (3) out_accessesnrRelsDirect

Interpretation. In general, classes with neither incoming accesses nor invocations are typical library classes or any kind of class that is close to the top of the system and is called from outside,

meaning a higher situated framework that is not inside the scope of the data.

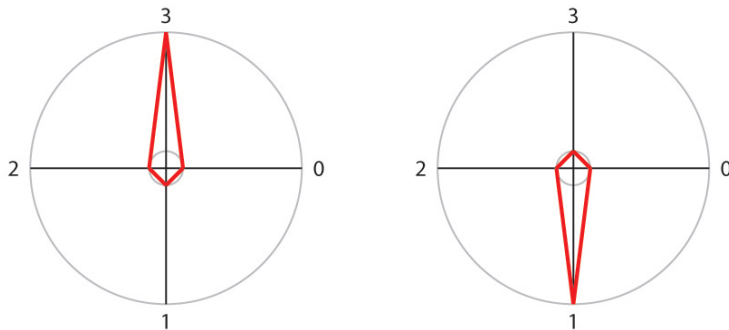


Figure 3.3: The left diagram shows a typical data consumer, the right one a data storage object.

Figure 3.3 reveals direct field accesses. A practice that is not exactly a bad smell, but rises at least considerable suspicion [FBB⁺99]. The graph on the left shows an entity that accesses variables directly, a data consumer. The diagram on the right shows a typical data storage object. An object with non-encapsulated fields that are accessed directly. An object that is definitely worth a closer look.

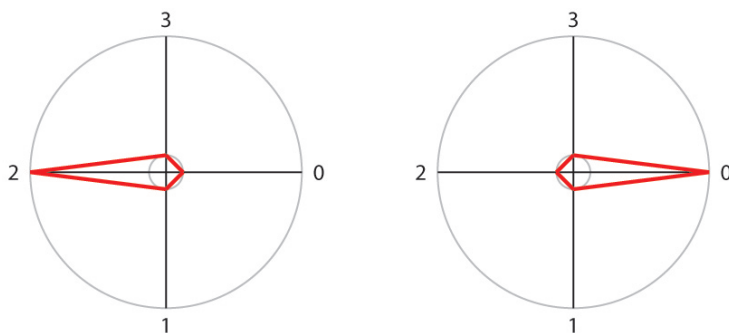


Figure 3.4: The left diagram shows a typical consumer of functionality. The right graphic depicts a provider.

The left diagram in Figure 3.4 shows a consumer of functionality. At this point we do not know what kind of services it consumes or from which other objects. Functions, procedures or data via a getter method. On the right is the equivalent provider. We do not know what kind of methods are invoked from this entity. Showing the neighborhood of invocations could yield additional information at this point.

In Figure 3.5 we see heavy consumers and providers, that peak in both aspects, accesses and invocations. The left diagram is once again a consumer, the right one a provider. Again both entities have peaks in the access-metric, what makes them somewhat smelly, due to non-encapsulated fields.

Finally, Figure 3.6 shows some extreme members of this preset view. The entity on the left is probably dead code, since there are neither incoming nor outgoing accesses or invocations. Although, there is a slight chance, that it is a class implementing an interface (and solely has implemented methods). Such invocations can only be determined at runtime, therefore only measurements of static relationships make it into the database. The pure opposite can be seen in the center. Probably a rare member in good designed software, once again due to the flaws coming from the access peaks. Without those, as shown by the diagram on the very right, an entity

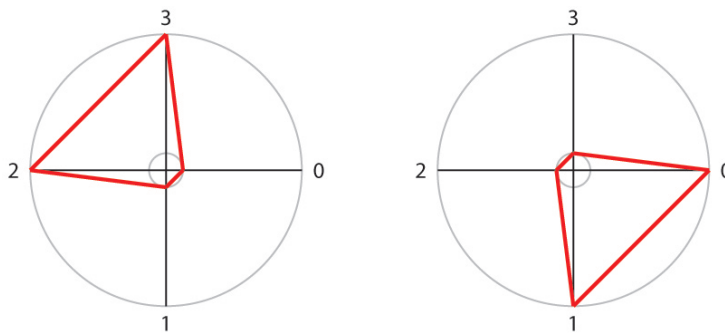


Figure 3.5: The left graphic shows a consumer of data and functionality, the right one a provider.

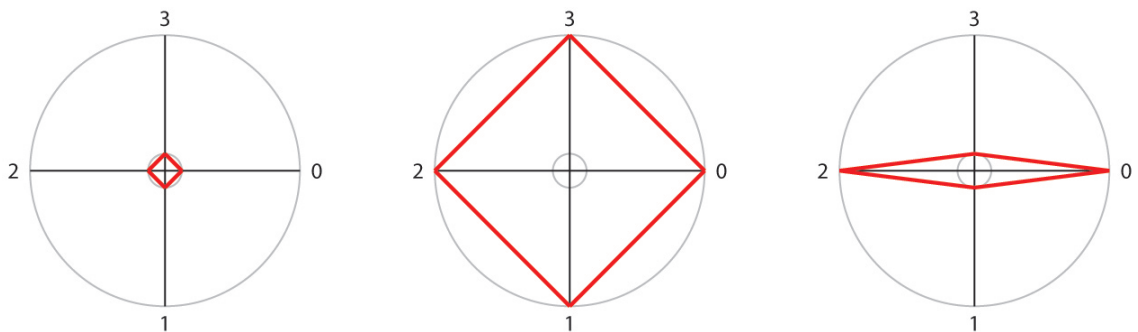


Figure 3.6: The left graphic shows most probably dead code, the right diagram an entity that is both provider and consumer at the same time.

with heavy incoming and outgoing invocations is probably a controller. A clear sign of method delegation.

3.5.2 Roots/Leaves View

Aspect. This view has its focus on inheritance. You can identify entities that are part of parent-child relationships and on which side of this relation they stand.

Configuration.

- (0) in_overridesnrRelsDirect
- (1) in_inheritsnrRelsDirect
- (2) out_overridesnrRelsDirect
- (3) out_inheritsnrRelsDirect

Interpretation. Since not every programming language supports multiple inheritance, that is inheritance from more than one parent, it can be quite natural to have many entities having full peaks in out_inheritsRelsDirect. For programming languages, like Java, that know only single inheritance, this metric is binary.

Figure 3.7 shows entities that are leaves in the inheritance hierarchy of the system in scope, since they have one or more parents but no children. The entity on the left has an additional peak in overridden methods from its parents. Entities that show an especially massive amount of overriding might be indicators for bad designs.

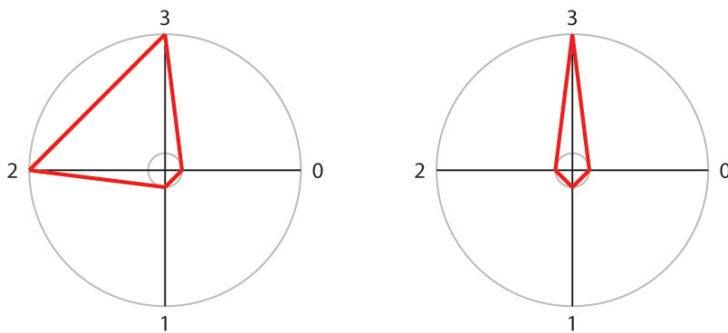


Figure 3.7: Leaves of the inheritance tree

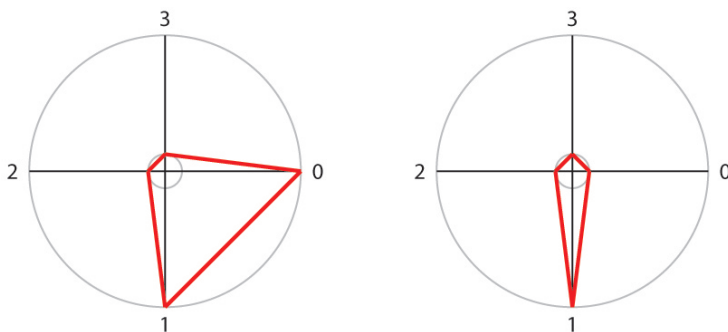


Figure 3.8: Roots of the inheritance tree

Figure 3.8 on the other hand shows roots in the hierarchy. They have no parents but inherit to children. Once again the left graph has an additional peak. It indicates how many methods are being overridden by children of this entity. Peaks in overriding can again be signs for inappropriate design structures.

Figure 3.9 shows two very different members of this view. The first on the left is definitely an entity that simply has inheritance relationships of any kind. The entity on the right however seems to be right in the middle of inheritance. It has parents as well as children. Entities like that are always worth a closer look, since such massive inheritance and overriding makes for a very complex subject and complex goes most of the time hand in hand with vulnerability to errors.

3.6 Concluding Thoughts

While playing around with the metrics at hand and fathom possible useful preset views, we felt the urge to relate metrics within single kivi diagrams. The relation between two metrics, for instance how long is the average method in this entity, cannot be displayed, when you only got metric values for the number of methods and the length of the file. Basically it is a lack of metrics. Although these metric values could be calculated and added to the pool without much difficulties, there might be an easier solution. One that may also open up lots of new possibilities.

The current normalizing algorithm compares every single metric over all displayed kivi diagram. For further implementations we should consider the addition of a second normalizer, that just compares values within single kivi diagrams. This would allow the direct comparison of metrics within a single entity and allow for additional aspects to be visualized. For instance,

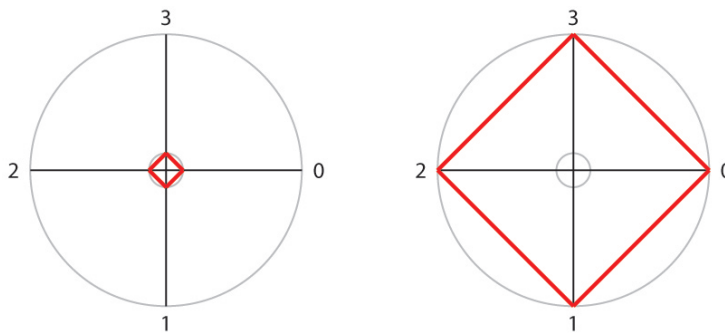


Figure 3.9: Extreme members of the view.

show the relation between the number of attributes and methods. Entities with twice as much methods as attributes might be data storage objects, that have a getter and a setter method for every attribute. Such object could easily be identified by visualizing this relation. Until now, peaks are pretty much everything you can look for. But given these changes, rifts between metrics would be able to be interpreted.

In fact, we are back to the representation condition, that we already mentioned earlier in the related work section. This simple rule does not apply in its entirety to our approach, only to the same metric values from each kiviati diagram. Within a single node, relations between metrics cannot be interpreted. Changing this principle by normalizing every metric *and* diagram gives new possibilities, as stated above, but also creates room for a new kind of misinterpretation. Not every two or more metrics can be sensibly related. There could arise a strong temptation for users to compare apples and oranges.

Since the “playing around” with metric configurations logically happened after the implementation was done, this proposition did not make it into the final release. But might be considered as future addition.

Implementation

An integral part of this thesis is the implementation of our approach as an Eclipse plug-in. We call it the Kiviat Navigator.

4.1 Integration

It is part of the thesis description to integrate our tool into the Eclipse environment¹ as a plug-in. And this with good reason. The Eclipse environment has an extensive plug-in infrastructure and is widely used as a integrated development environment for software projects. It makes sense, that users have their tools right where they are working everyday.

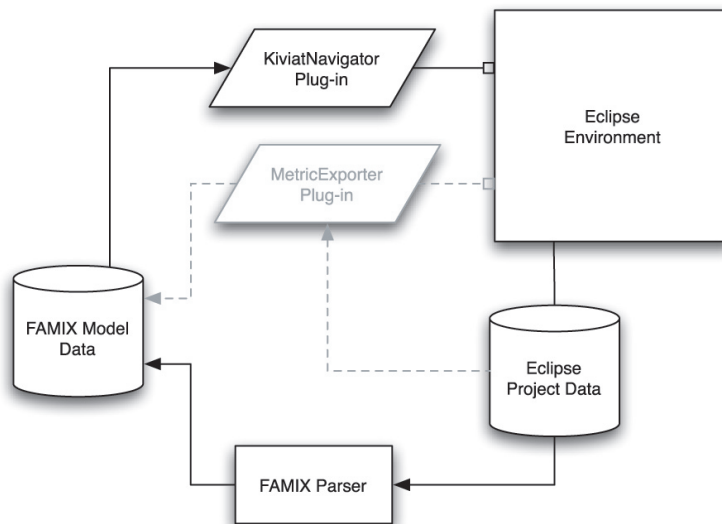


Figure 4.1: This chart shows how the Kiviat Navigator is embedded into the Eclipse environment and where data comes from. Components in faded grey are not currently used by our implementation, but show the envisioned goal.

The Kiviat Navigator relies on data, that differs according to the project to be analyzed and changes constantly while the project is under development. Before metrical values find their

¹<http://www.eclipse.org>

way to the final database they are parsed from the project repository and converted into FAMIX [SDT99], a meta model for representation of object-oriented source code data. Figure 4.1 gives you an overview how the neighborhood of the Kiviat Navigator looks like.

This task was intended to be accomplished by the Metric Exporter by Hanimann [Han06]. Another Eclipse plug-in, that would have enabled us to create the necessary model data dynamically. Unfortunately the Metric Exporter did not reach the final state of implementation until the end of our project time. Hence, the current version of the Kiviat Navigator can only visualize specifically prepared data pools.

4.2 Requirements

The description of the *incremental exploration* strategy in the previous chapter leads us to the following requirements of our plug-in.

We basically need two things. A layout strategy, that allows the generation of flexible views and a navigation strategy. For the layout we will make use of the ArchView approach. This means basically a migration of the KiviatVisualizer [Pin05] to the Eclipse platform. In accordance with our thoughts in the last chapter, the choice of exploration paths will reside with the user for now. Although there is a strong potential for the tool to support specific exploration scenarios to assist the user, the plug-in will not suggest such paths at the moment. However, there must be the possibility to choose from a pool of preset views.

Additionally, we need a way of choosing one or more entities as our subjects in focus. And we need facilities for selection and configuration of metrics and releases.

4.3 System Setup

The following tools and libraries have been used to develop the Kiviat Navigator.

Eclipse Platform for Plug-in Development. Eclipse SDK comes with integrated plug-in development support and uses the Standard Widget Toolkit (SWT). The usage of yFiles (see description below), which uses the Java Swing Framework forces us to use the SWT_AWT bridge class. Since this feature is currently not supported on macintosh platforms, keep in mind that you are restricted to development on linux or windows.

PostgreSQL Database. An open source relational database system².

Hibernate. In accordance with other projects in our research team, hibernate³ is used as data access layer and query engine.

FAMIX. Data is mapped from the database to FAMIX entities. A meta model for source code data [SDT99].

yFiles. yFiles⁴ is a Java graphics library, with extensive features in graph visualizations.

²<http://www.postgresql.org>

³<http://www.hibernate.org>

⁴http://www.yworks.com/en/products_yfiles_about.htm

4.4 Kiviat Navigator Overview

The Kiviat Navigator adds a new perspective to the Eclipse environment, consisting of two new views and an editor. The editor handles the visualization of kiviat diagrams. While the two views, the HierarchyView and MetricsView, allow configuration of said visualization.

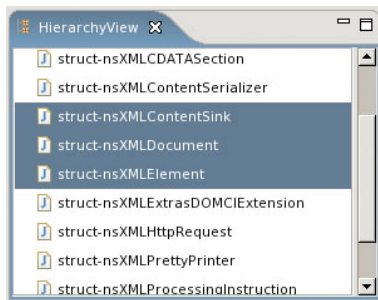


Figure 4.2: The HierarchyView shows all entites available for visualization

The HierarchyView, as shown in Figure 4.2, presents the user with all available entities. Currently a TreeViewer handles the layout and entities, such as packages and classes are distinguished. This view is obviously still very raw, it just gives basic functionality to use the tool. For future enhancement of the tool, this view should be switched with a project explorer native to the project language.

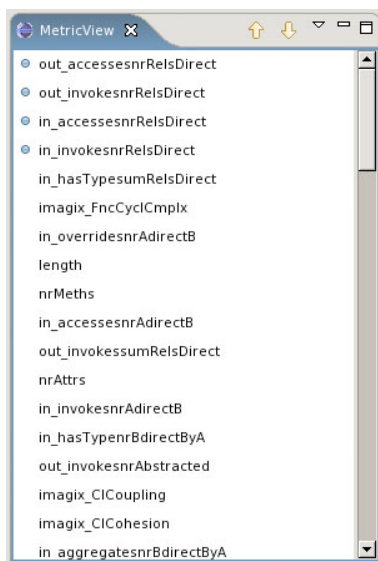


Figure 4.3: The MetricView allows selection of metrics

The MetricView, presented in Figure 4.3, allows the configuration of metrics. Select or deselect any metric to add or remove it from the active kiviat visualization. To change the order of displayed metrics use the arrow buttons (refer to Figure 3.2 as to how metrics are drawn in a kiviat diagram).

The afore mentioned preset views can also be chosen here. Open the drop down menu to access all available presets.

Note: In similar design, the `ReleaseView` would be found here, as a second view frame in the folder. Most modules already implement functionality for the visualization of multiple release data. Unfortunately, during the final implementation no data pools with multiple release data was available to us, so this functionality was never tested and therefore did not make it into the final release.

Finally, the kivi diagrams are visualized inside the editor space. The editor is dynamically updated whenever the user changes something. Adding or removing entities, adding or removing metrics or changing their order will cause the editor to be updated.

Performance Issues: During the whole implementation we worked with a relatively small amount of data. So performance was never really an issue. All hibernate queries and data handling was implemented straight forward without giving a second thought to performance, which is of course in perfect agreement with the credo of first do it right and tune it later. However, when we migrated to the considerably larger Mozilla database (refer to the case study for more information) performance hit us like a hammer. We have not yet determined where exactly most speed is lost. It remains future work to be done.

4.5 Kivi Navigator Architecture

Let us take a brief look beneath the surface of the Kivi Navigator. The whole implementation stays relatively close to the Eclipse framework. There are no extraordinary design patterns involved that might let the software engineers heart beat faster. Nevertheless we will pick a few aspects from the design and describe them in more detail. At least the guy, who will continue the implementation of the Kivi Navigator might be happy to get some insight on how things work inside, before he writes, hopefully, a better version of it.

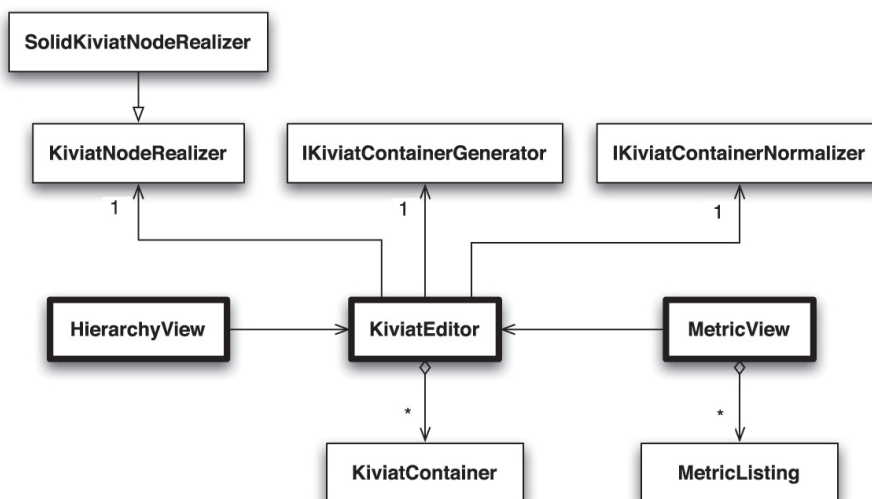


Figure 4.4: This is a simplified graph of the components involved in the Kivi Navigator. The modules with thicker black borders contain multiple classes, which mostly are members of model-view-control patterns.

Figure 4.4 shows the basic relations between the components of the Kivi Navigator. The com-

ponents with the thick borders are conglomerates of multiple classes, but all direct extensions or implementations of Eclipse framework classes. `MetricView` and `HierarchyView` are both Eclipse views and `KiviatEditor` is a `MultiPageEditor`. All three are strongly coupled with the framework. If you need closer information about their inner workings we recommend [CR06].

4.5.1 KiviatContainer and KiviatContainerGenerator

`KiviatContainers` are the most central data objects in this implementation. They contain every information necessary for a single kiviat diagram to be drawn. This is basically a hashtable with metric names as keys and an array of doubles for values (an array, since there might be more than one release). Additionally a name is remembered as label.

Depending on the programming language we inspect, these `KiviatContainers` must be created from different kinds of entities. To preempt this problem we delegate the creation of our object to a factory. Whenever the creation of `KiviatContainers` is necessary the call will be routed via an implementation of `IKiviatContainerGenerator`. The implementation depends on the source code language of our data.

This interface contains only one simple method:

```
public interface IKiviatContainerGenerator {
    public KiviatContainer createKiviatContainer(
        Object kiviatObject,
        String[] attributes, //list of attributes to be loaded
        String[] releases); //list of releases to be loaded
}
```

Listing 4.1: `IKiviatContainerGenerator`

You will notice that this method asks for a `java.lang.Object` instance as input. In a perfect world we would have used customized subclasses of hibernate objects. In this way, we could have stored every bit of information needed for our visualization inside our data object at the time of its creation, during the hibernate query. Unfortunately a procedure like this, would involve changing the famix model entities. We decided to leave these classes with their pristine existence and chose the way generating a fresh object for every database entity. The performance issue here is no singularity. The same wrapping mechanism is also used in other parts of the system. The `MetricListing` classes, for instance, are generated to suit the needs of the `MetricView` and are created for all the metrics available. Concluding one might say, there is definitely room for performance improvement in this part.

4.5.2 `IKiviatContainerNormalizer` and `KiviatMaxNormalizer`

Before `KiviatContainers` can be drawn on screen, they need to be normalized. We want them all to have equal size and furthermore the metric values for each and every metric must be normalized in a way, to allow comparison.

The `KiviatMaxNormalizer` computes the maxima for each metric from all `KiviatContainers` and normalizes the respective values accordingly. This computation generates a set of kiviat diagrams that can be compared by their metrics.

A striking performance issue cannot be overseen at this point. Whenever we change the smallest part of the visualization; add or remove a metric or an entity. We will be forced to compute a large part of the visualization anew, if not everything from scratch. For instance, adding a metric to the existing configuration will just cause the computation of that single metric for all `KiviatContainers`, but nonetheless every container must be refreshed, since adding an axis to the diagram

will change the angles. Now this was the happy case. Should the user decide to add an entity to the visualization, every metric value of every KiviatContainer will need to be computed and drawn again.

4.5.3 KiviatNodeRealizer

This is where we actually touch the yFiles libraries. The KiviatFrame harbors an instance of Graph2DView, which is the core of our yFiles visualization. The Graph2DView needs a NodeRealizer that knows how nodes should be displayed. This is where we extend the framework. The KiviatNodeRealizer extends the yFiles NodeRealizer and implements drawing of the kiviatic graphs. On a closer look, the KiviatNodeRealizer only provides drawing routines for the kiviatic diagram grid. No actual drawing happens there. There are further distinguished child classes that handle the actual drawing. The only working subclass of KiviatNodeRealizer at this point of time is the SolidKiviatNodeRealizer.

Chapter 5

Mozilla Case Study

In the Mozilla case study we focus on analyzing the classes that implement the content and layout handling of the Mozilla¹ web browser 1.7. In total, we get 2786 classes. It is an unfortunate fact, that the current version of the Kiviat Navigator is unable to process such a massive amount of entities at once in a reasonable timeframe. And since we cannot visualize the whole lot of classes in smaller groups — this would distort our results, as metric measurements will be normalized with different maxima — we decided to apply our approach to a subset of all entities.

5.1 Approach

Given the subset of entities, we apply preset views as tools to reveal bad smells or characteristic classes. Whenever we find such interesting entities, that resemble a pattern from our catalog or otherwise show interesting features, we mark them. After analyzing one view, we make a decision what view to apply next, based on the findings of the last one. We also have the possibility to extend views to gain additional information and hereby get a grasp at the true identity of the entities in scope. We remove markings when we think we identified a class' true face and consider it "harmless". All classes that show disturbing qualities will be added to a list for closer inspection. Along the way we make statements concerning the usability of our approach and the views we used.

5.2 Investigation

We focus on a small set of entities, chosen by the mere similarity of having the string "nsXML" in their class name. This selection yields a pool of fourteen classes, that implement parts of the browsers XML functionality. All of them unknown to us at this point, only their name might give us an indication of their role in the system.

Hotspot View.

For starters we decide to generate a view that has its focus on pure size metrics. Lines of code (length), number of methods (imagix.CIMemFnc), number of attributes (imagix.CIMemVar), number of type definitions (imagix.CIMemTyp), and number of internal classes (imagix.CIMemCl) will do for now. This first view is actually a very simple hotspot view, adapted from [Pin05] and [LD03].

¹Mozilla source code is written in C++.

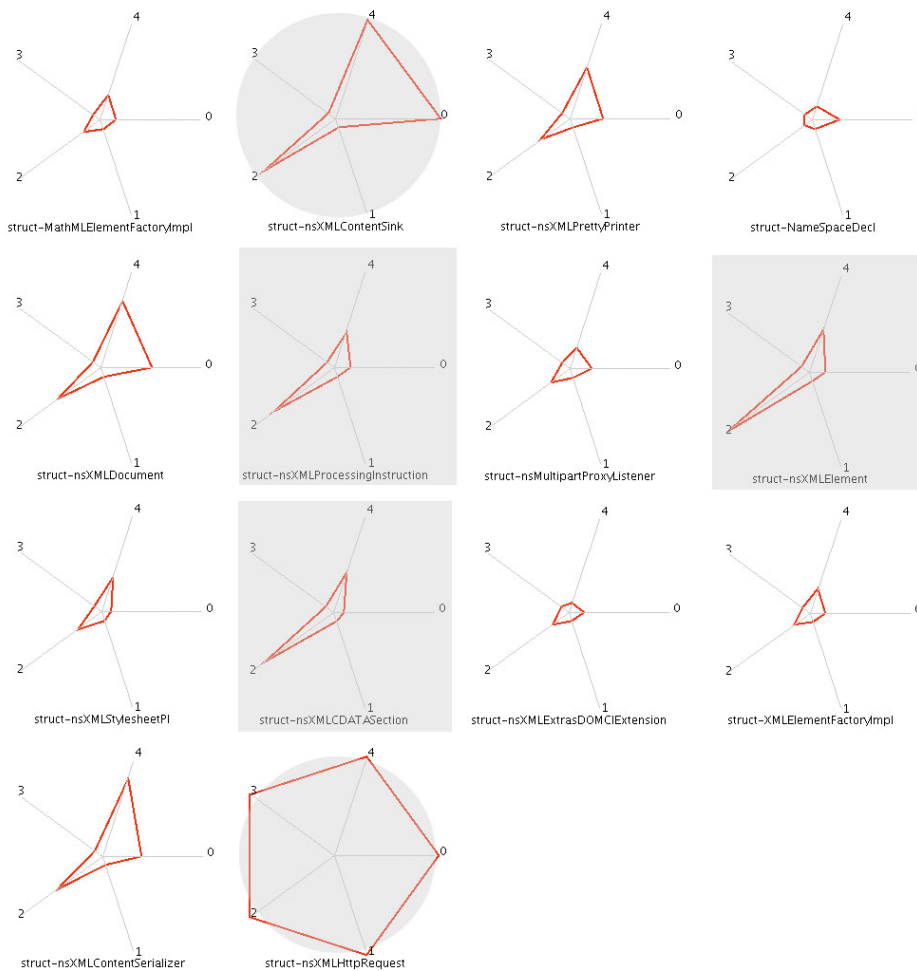


Figure 5.1: A simplified System Hotspot View [Pin05][LD03]. The metrics are the following: (0) `imagix_CIMemVar`, (1) `imagix_CIMemTyp`, (2) `imagix_CIMemFnc`, (3), `imagix_CIMemCl`, (4) length.

Figure 5.1 shows the results of this configuration in the Kiviat Navigator. There are the obvious eye-catchers, highlighted with faded grey circles, `nsXMLHttpRequest` at the bottom of the second column and `nsXMLContentSink` at the top of the same column. Both classes have the highest peaks in one or more metric values. The class `nsXMLHttpRequest` actually peaks in every metric shown. We have to keep in mind, that we just look at part of the whole system. Hence, we do not know how large the other existing classes are. There is even the chance that all other entities of our selection are extraordinary small, thus letting our god class suspect appear bigger than it actually is. In this case `nsXMLHttpRequest` has a total of 71 members, of which 53 are functions and 16 are attributes. Nonetheless, an entity like that is definitely worth a closer look. The class `nsXMLContentSink` shines in similar ways as `nsXMLHttpRequest`: Extraordinary large number of lines of code and variables.

Additionally we highlighted three classes that show large numbers of methods, but otherwise not extreme values at all. These are marked with faded grey square overlays.

At this point we gained some information about the entities in our pool, but we cannot make

any strong deductions yet. And there is definitely more to be known, so we decided to look at our selection from another perspective.

Provider/Consumer View.

We change to the Provider/Consumer View, a preset view, whose features we already discussed in the preset view catalog (refer to section 3.5 for more information).

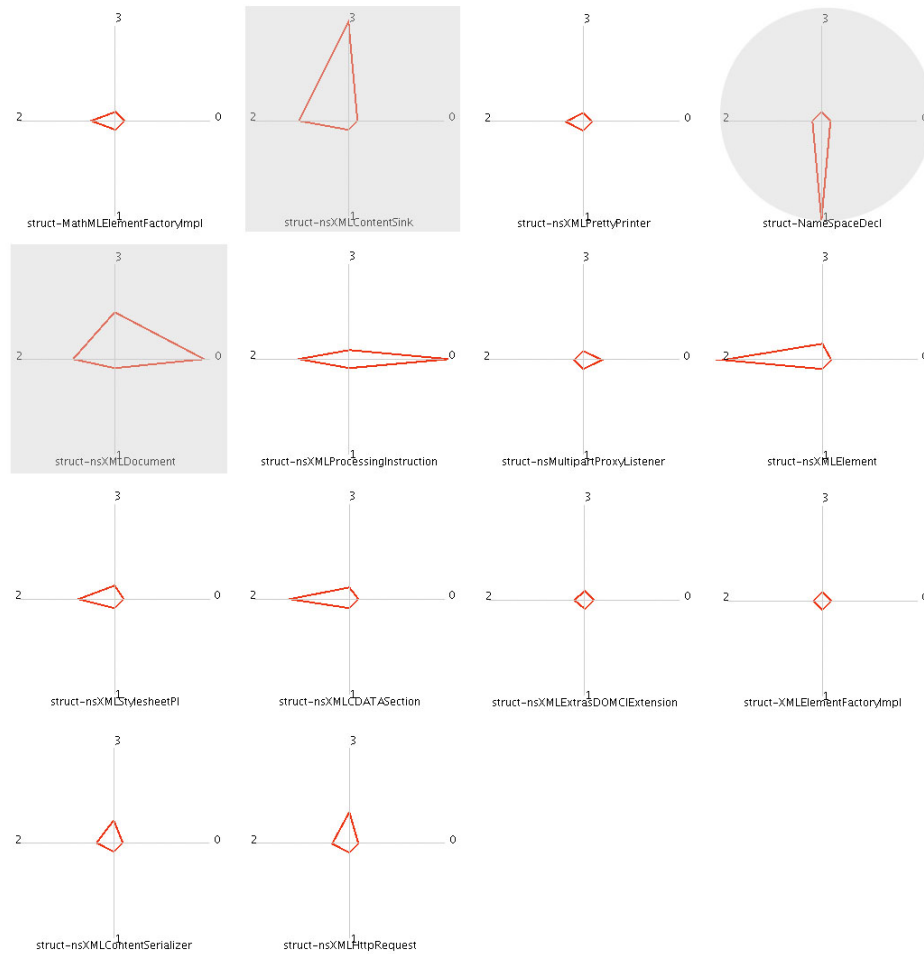


Figure 5.2: Provider/Consumer View. The metrics are the following: (0) in_invokesnrRelsDirect, (1) in_accessesnrRelsDirect, (2) out_invokesnrRelsDirect, (3) out_accessesnrRelsDirect.

The Provider/Consumer View might shed some light on our marked classes and what their purpose in the system is. Figure 5.2 shows all entities from this viewpoint. The first class that gets our attention here is `NameSpaceDecl`, marked with the faded grey circle. According to our catalog of patterns it is a typical data storage object and therefore a bad smell. A short look at the source code proves us right.

```
typedef struct {
    nsString mPrefix;
    nsString mURI;
```

```

    nsIDOMElement* mOwner;
} NameSpaceDecl;

```

Listing 5.1: NameSpaceDecl

The class has three non-encapsulated fields, that are heavily accessed from other classes. At least we know, it is a small member of the system, since it did not get our attention in the hotspot analysis, and it might be accessed solely within a single class. At this point analyzing the relationships' accesses is necessary to get further information.

There are two other classes that shine through direct field accesses; both marked with the square overlay. The class `nsXMLContentSink` reveals a consumer existence. The high amount of outgoing access relationships can also be considered a bad smell. This is also the second time this class shows up on top of the pile. So far we know: `nsXMLContentSink` is huge and accesses a high amount of non-encapsulated variables. An entity we should consider putting on the list for closer inspection, along with `nsXMLDocument`, another class with a peak in outgoing access relationships.

From the last view we remembered the three classes with an extraordinary number of methods. In this view we see, that two of them are typical consumers of functionality, and since none of them are accessed in any kind of way, they are most likely library classes. There is probably nothing wrong with them. Many methods go along with high flexibility, which is usually the way people want libraries.

The third of the former three classes shows incoming relationships. This seems to be a controller class with a high amount of method delegation. Listing 5.2 shows two example methods from `nsXMLProcessingInstruction`. We see that most methods in this class are delegators to `nsGenericDOMDataNode`, which confirms our assumption.

```

GetData(nsAString& aData)
{
    return nsGenericDOMDataNode::GetData(aData);
}

SetNodeValue(const nsAString& aNodeValue)
{
    return nsGenericDOMDataNode::SetNodeValue(aNodeValue);
}

```

Listing 5.2: Two method examples from nsXMLProcessingInstruction

Another striking revelation is the fact, that our supposed god class `nsXMLHttpRequest` shrank into non-existence from this perspective. There are no invocations or accesses in or out from this entity. A small amount of outgoing accesses are made, which does not make us any happier about this class. This seems to be a giant class using no one and used by no one. Since we need definitely more information a third view is applied.

Roots/Leaves View.

The Roots/Leaves view reveals certain inheritance features of our entities.

Figure 5.3 shows our pool of entities once again from another viewpoint. This time inheritance and overriding. We see, that there are multiple entities that participate in a parent-child relationship and two that are no subject to inheritance whatsoever. There are two classes that show a high amount of overriding. The one with most, `nsXMLElement`, counts 53 overrides. Furthermore, the two classes marked with grey squares are heavily integrated into inheritance structure. Both are worth a closer look.

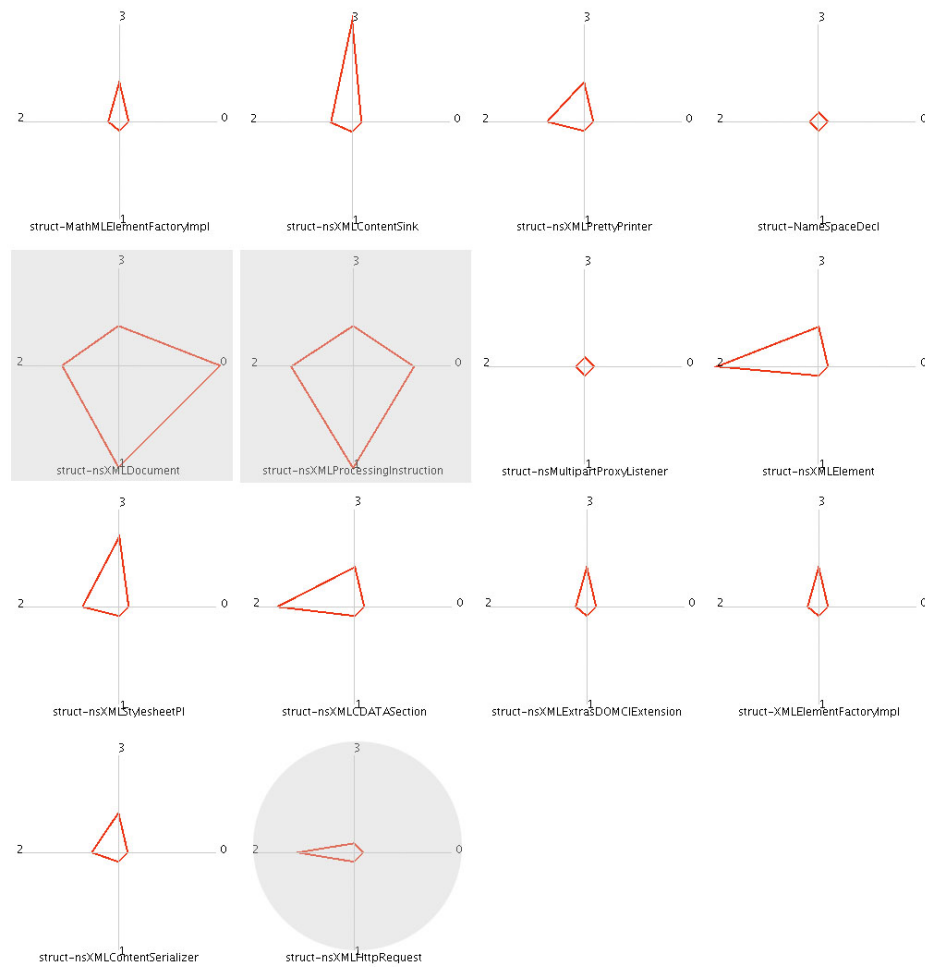


Figure 5.3: Roots/Leaves View. The metrics are the following: (0) `in_overridesnrRelsDirect`, (1) `in_inheritsnrRelsDirect`, (2) `out_overridesnrRelsDirect`, (3), `out_inheritsnrRelsDirect`.

Finally, new information is available for our two biggest classes from the hotspot analysis. The `nsXMLContentSink` class is subject to multiple inheritance. But even stranger is the fact that `nsXMLHttpRequest` seems to override one or more methods but is neither parent nor child! Obviously a paradoxical situation that arises questions about the validity of what we see. We chose to dig deeper and discovered that indeed something went wrong with the computation of the metric values of this particular class. In fact `nsXMLHttpRequest` has eight parent classes.

5.3 Summary

Table 5.1 lists all the classes we deemed worth a closer look.

What have we learned from this short investigation? First, we were able to identify at least three classes that can be considered bad smell. Moreover, we have another two classes on the list that are worth to be inspected in more detail. We think it is safe to assume that the preset views were able to reveal specific aspects of the software, although we were limited in many

Table 5.1: Table of Findings

<i>Class</i>	<i>Symptoms</i>
nsXMLHttpRequest	Exceptionally large — probably doing too much itself.
nsXMLContentSink	Exceptionally large, with a high amount of outgoing accesses (bad smell). Additionally subject to multiple inheritance.
NamespaceDecl	Its non-encapsulated fields are frequently accessed, which is a bad smell.
nsXMLDocument	Outgoing direct field accesses (bad smell). Strongly knitted into inheritance structure.
nsXMLProcessingInstruction	Strongly knitted into inheritance structure.
nsXMLElement	High amount of overriding.

dimensions. These shortcomings are in fact possibilities and benefits that may yet be reaped by enhancing the Kiviat Navigator.

The explorations paths are clearly ambiguous and the decision what view to apply after a first or a second perspective is all but trivial. The choices have to be made by software engineers and specific scenarios, that might be of general use, have not yet been found.

Conclusions

We started our thesis by taking a look at different concepts of information visualization. We learned that, although, the geometric aspects of source code data simplifies graph representation, the visualization of such large and complex structures is all but easy. Even a fine programmed algorithm will reach its limits. Inevitably, the time will come, when the information pool gets so large, that static layout techniques cannot cope with the challenge anymore. Navigation and Exploration concepts are necessary to give control of the data back to the user.

During the analysis of related work, we see how other tools, such as SHriMP, approaches these problems. They combined several different navigation strategies to give the user a maximum of flexibility, while still letting him see the whole system all the time. An approach that has its benefits, but proved to be not very applicable to our ground work, since our focus lies on metric comparison and not primarily on the visualization of hierarchic structures.

The concept of *incremental exploration* appealed to us. It is a navigation technique suited for especially vast systems, that cannot be beheld in just one glimpse. A fitting strategy at a point in the analysis of a software system, where the focus lies on the revelation of specific problems and not the visualization of everything, including completely "healthy" source code. There are plenty of good tools out there that help in understanding the hierarchical aspects of program structure. SHriMP for example, as well as the Eclipse development environment already provide a nice set of reverse engineering functionalities.

Having found our way of navigating, our focus shifted to the covering of view configurations. A task, that proved to be more difficult than anticipated. Even though basic views were already presented in the works of Lanza *et al.* [LD03] and Pinzger [Pin05]. The set of metrics available to us diverged from the others. We lacked the existence of composite metrics and were therefore limited to simple views, that focus mainly on visualizing maxima and not relations. We further discovered that our visualization implemented the representation condition only to a certain limit. This made it impossible to create preset views, that have their focus on metric relations within entities. Although, we must say, that this shortfall of flexibility is actually a future possibility that should be evaluated in detail. The changes necessary to open up our implementation to this concept are probably small ones.

The configuration of views leads directly to the configuration of view transitions. Obviously the idea of standardized exploration paths, using these views, is very tempting. We are sure such scenarios must exist, but the means to produce and evaluate them were not given during this thesis. We suggest a study, where different users analyze software systems, using the Kiviat Navigator and write down the decisions they have made and paths they have taken to reveal specific code symptoms. Such an effort could yield common ways of analyzing software, such as standardized exploration paths, that could ease the process of reverse engineering further.

Concluding, we think our approach describes a suitable way of exploring source code data,

with the use of kiviatic graphs. The implementation of our approach is definitely not the end of the line, but leaves open every possibility.

6.1 Contribution

In this thesis, we described an approach to investigate and navigate source code data. We evaluated the implementation of this approach and were able to show, that beneficial results can be found by it. This was done by starting a catalog of standardized *measurement mapping* configurations and analyzing the characteristic emergent patterns of these views. The sequential appliance of these views to a target software system leads to an increasing knowledge about entities. Having information of an entity from more than one perspective can rule "harmless" members out and also strengthen the evidence of potentially bad designed classes.

Additionally, we developed a basic framework for kiviatic graph visualization for the Eclipse platform, which we evaluated with a sample code investigation. The tool can be further enhanced in the future.

6.2 Outlook

On-going and future work is concerned with the following items:

- Our catalog of preset views is far from complete. New metrics and relations will open up new possibilities for revealing additional code symptoms. A systematical approach to the topic, that creates a library of views and pattern interpretations could be a way of further enhance our knowledge in this field and at the same time gain useful tools, that can directly be evaluated in the field.
- Like mentioned in the last section, sequences that identify more complex structures within source code data are of considerable interest. The identification of such common exploration paths might also lead to useful material.

Appendix A

Contents of CD-ROM

This thesis is distributed with a CD-ROM, that contains the following data:

<code>Thesis.pdf</code>	This document in Portable Document Format
<code>Abstract.pdf</code>	Abstract of the thesis in English
<code>Zusfsg.pdf</code>	Abstract of the thesis in German
<code>KiviatNavigator.zip</code>	The complete source code of the KiviatNavigator Eclipse plug-in

References

- [Chi06] Chisel group. <http://www.thechiselgroup.org>, September 2006.
- [CR06] Eric Clayberg and Dan Rubel. *Eclipse. Building Commercial-Quality Plug-Ins*. Addison Wesley, April 2006.
- [ESS92] S. C. Eick, J. L. Steffen, and Jr. Sumner, E. E. Seesoft-a tool for visualizing line oriented software statistics. *Software Engineering, IEEE Transactions on*, 18(11):957–968, 1992.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.
- [Han06] Christian Hanimann. Seal platform - towards an intergrated tool platform for software architecture and evolution analysis. Master’s thesis, Software Evolution and Architecture Lab, University of Zurich, 2006.
- [HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.
- [LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.
- [LN95] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA ’95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 342–357, New York, NY, USA, 1995. ACM Press.
- [MK88] H. A. Muller and K. Klashinsky. Rigi: a system for programming-in-the-large. pages 80–86, 1988.
- [MMF03] J. I. Maletic, A. Marcus, and L. Feng. Source viewer 3d (sv3d) - a framework for software visualization. pages 812–813, 2003.
- [PBS93] B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualization. *Visual Languages and Computing*, 4(3):211–266, 1993.
- [Pin05] Martin Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.
- [SDB98] J.T. Stasko, J. Domingue, and M.H. Brown. *Software Visualization - Programming as a Multimedia Experience*. MIT Press, 1998.

- [SDT99] Patrick Stezaert, Serge Demeyer, and Sander Tichelaar. Famix 2.0 - the famoos information exchange model. Technical report, Software Composition Group, University of Berne, Neubrückestrasse 10, CH-3012 BERNE, September 7, 1999.
- [SWFM97] M. A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. pages 38–45, 119, 1997.