

Using Diagrammatic Explorations to Understand Code

by

Vineet Sinha

S.M. Computer Science and Engineering, Massachusetts Institute of Technology (2003)
B.A.Sc. Computer Engineering, University of Waterloo (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author

.....
Department of ~~Electrical Engineering and~~ Computer Science
January 31, 2008

Certified by

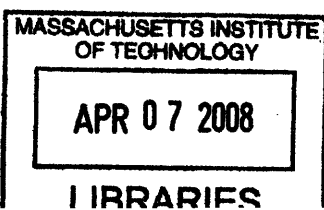
.....
David R. Karger
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by

.....
Robert C. Miller
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by

.....
Terry P. Orlando
Chairman, Department Committee on Graduate Students



ARCHIVES

Using Diagrammatic Explorations to Understand Code

by

Vineet Sinha

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2008, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract:

Understanding code is a significant challenge for developers. This thesis examines the limitations of current tools that use diagrams to assist code comprehension and demonstrates the value of four design principles:

- That diagrams should be based on familiar models such as UML class diagrams and layered architectural diagrams, so that developers can understand them without additional training.
- That the familiar diagrams must be able to focus on specific parts of a codebase relevant to the developer's task, to prevent users from getting overwhelmed with irrelevant information.
- That the focused diagrams need to support exploration of the codebase by directly interacting with the existing diagram.
- That the focused diagrams can be created by users' exploration as needed for their tasks in traditional code editors.

This thesis shows that understanding for software developers can be effectively supported by interactive exploration using focused diagrams of familiar representations of code. These ideas have been combined to build two tools: Strata, which displays using the popular layered architectural diagrams, and Relo, which is based on UML class diagrams. The tools have been evaluated using both controlled lab studies and field deployments. Study results have been positive, indicating merit in these ideas.

Thesis Supervisors: David R. Karger and Robert C. Miller

Titles: Professors of Computer Science and Engineering

*to my parents and brothers
who inspire me to leave no stone unturned*

Acknowledgments

Working on a PhD can often seem to be a daunting task. I am glad to have had the encouragement, support, and advice of a great set of mentors, colleagues, and friends.

I want to start by thanking my advisors, Prof. David R. Karger and Prof. Robert (Rob) C. Miller, for their help in guiding me with my research. Rob helped me understand how to do research while working with users, while David helped me understand when to ignore users while doing research :-). Rob must have saved me over a year by pointing me to latest interface techniques and helping in prioritizing tasks for my research. David helped me understand how to crystallize ideas and arranged for funding my graduate studies.

This work would have been significantly harder to do without the funding that helped me focus on both my masters and my PhD research. Funding from HP, Nokia, the Simile project, and the MIT Oxygen Project are greatly appreciated.

Beyond my advisors, I received great support from faculty here at MIT and in particular those on my committee. Like my advisors, Prof. Daniel Jackson and Dr. Howard Shrobe helped in development of the research, and asked the right questions to fully realize the impact of this work. Other faculty have helped by giving their valuable words of wisdom and by showing their confidence in me and my work - special thanks go to Prof. Larry Rudolph, Prof. Michael Ernst, Prof. Victor Zue, Prof. Tomas Lozano-Perez, Prof. Leslie Kaelbling, Prof. Randall Davis, and Prof. Sam Madden. The opportunity to get this feedback would not have been possible without the framework provided by the Electrical Engineering and Computer Science Department and the resources provided by the Computer Science and Artificial Intelligence Lab.

I also received help and feedback on ideas and suggestions for implementation approaches from members of my research groups. These research groups include the Haystack group and the User Interface Design (UID) group. Thanks go in particular to David Huynh, Jaime Teevan, Karun Bakshi, Michael Bernstein, Adam Marcus, Max Goldman, Greg Little, Kai Shih, Harr Chen, Nick Matsakis, Punyashloka Biswal, Solomon Bisker, Mike Bolin, Yuan Shen, Sacha Zyto, Jones Yu, and Dennis Quan. Along with these research groups, were also members of W3C and Simile, in particular Stefano Mazzocchi, and members of the larger Center for Reliable Software, and in particular Derek Rayside, Adam Kiezun, and Jonathan Edwards.

continued on next page

continued from previous page

This research also benefited from the contributions of a number of people outside of our lab. In particular, Elizabeth Murnane joined in fixing bug and implementing research ideas in the built tools. Cyrus Kalbrener provided code contributions that helped in increasing the performance of the work. A number of other people gave feedback on features that were not working properly, or those that needed polishing - special thanks go to Benedict Heal, Emerson Murphy-Hill, and Richard Nemec. Help on solving some of the problems from Kevin Wilkinson, Daniel Tunkelang, and Randy Hudson are greatly appreciated.

Additionally, some of the work in this thesis comes directly from work I did while at Accenture Research Labs. I am grateful for them giving me the opportunity to gain a better understanding of the problems faced by software projects being built at different companies. Beyond the labs resources I also appreciate having had the opportunity to work with lab members including Edy Liongosari, Scott Kurth, Kishore Swaminathan, and Mark Grechanik.

While working deeply on problems I have needed to have my sanity kept in check with the help and support of a great group of friends. One group includes my friends at the lab, including Ali Mohammad, Harr Chen, Adam Marcus, Federico Mora, Jacob Eisenstein, Sayan Mitra, Kinh Tieu, Yuan Shen, Gregory Marton, and Olya Veselova. Another group consists of those that I have lived with while working on this research - David Huynh, Harold Fox, and Hang-Pang Chiu. And those that that I have known via dancing and Tango, including Suelene Chu, Jordan Hunt, Long Ngo & Ulrike Kappes, Anna Custo, Panayiotis Lemonidis, Richard Huang, Eszter Hars, Bryan Ford, Sheila Erimez, Jacob Eggers, Shu-Yee Chen, Miriam Sorell, Sharon Kuo, Steve & Pamela Slavsky, Carlos & Tova Moreno, Eray Yuksek & Martina Gracani, Erick Eisack, Ting Chen, and Caleb Welton.

And my family... who consistently kept me focused on my research. Just as law enforcements might have perfected the use of good cop - bad cop roles for investigation, my parents and brothers seemed to have perfected the art of making me work on the most important tasks, either by sharing their wisdom, showing their love, annoying me, or by just reminding me that "I am still in school".

CONTENTS

1. INTRODUCTION	21
1.1. The Problem	21
1.2. Program Comprehension	23
1.3. Approach.....	23
1.4. Walkthrough.....	29
1.4.1 Layered Overviews with Strata	29
1.4.2 Detailed Relationships with Relo	32
1.5. Scenarios.....	37
1.6. Contributions.....	38
1.7. Outline	39
2. PREVIOUS WORK	41
2.1. Developer Behavior	41
2.1.1 Understanding Models	42
2.1.2 Use of Sketching	43
2.1.3 Working with Concerns	44
2.2. Overview Tools	45
2.3. Exploration Tools	52
2.3.1 Tree based approaches	53
2.3.2 Visualization based approaches	54
2.3.3 Discussion	58
2.4. Tools for Large Projects.....	59

2.5. Design Tools	61
3. SURVEY OF SOFTWARE IMMIGRANTS	63
3.1. Method.....	64
3.2. Results	66
3.3. Task Analysis	68
4. STRATA USER INTERFACE	71
4.1. Appearance	72
4.2. Supporting Interactive Layout	76
4.2.1 Guess Based Layout	76
4.2.2 Interacting with Layout	81
4.2.3 An Active Layout Engine	81
4.3. Supporting Interactive Exploration	83
4.3.1 Interface Behavior for Navigating	84
4.3.2 Exploration via Navigation Buds	84
4.4. Updating Module Definitions.....	85
5. RELO USER INTERFACE	87
5.1. Appearance	87
5.2. Supporting Interactive Layout	90
5.2.1 Building an Incremental Interactive Layout Engine	90
5.2.2 Rules Based Layout	91
5.3. Supporting Interactive Exploration	93
5.3.1 Navigation Buds	93
5.3.2 Levels of Detail	95
5.3.3 Autobrowse	96
5.4. Automated Diagram Management.....	96
5.4.1 Linked exploration	97
5.4.2 Automated Removal of Items	97
5.5. Supporting Communications using Diagrams.....	98
6. IMPLEMENTATION.....	101
6.1. Development Challenges.....	101
6.2. A Caching Architecture.....	103

6.2.1 Mapping Support	103
6.2.2 Basic Caching using Builders	104
6.2.3 Caching Compound Relationships	104
6.3. Agent Framework	105
7. EVALUATIONS	107
7.1. Using Strata with Projects	107
7.1.1 Methodology	109
7.1.2 Results & Discussion	109
7.2. User- Study with Strata	113
7.2.1 Methodology	113
7.2.2 Results & Discussion	114
7.3. User-Study with Relo	117
7.3.1 Method	118
7.3.2 Results & Discussion	125
7.4. Feedback from the Field with Relo	132
8. CONCLUSIONS	135
8.1. Summary of Contributions.....	135
8.2. Raised Questions	136
8.3. Looking Ahead.....	136
8.3.1 Design Patterns	136
8.3.2 Program Knowledge Extraction	137
8.3.3 Different Diagram Types	137
8.3.4 Collaboration and Communication	138
8.3.5 Support for non-software domains	138
REFERENCES.....	139

FIGURES

Figure 1 – Typical class diagrams.....	27
Figure 2 – Typical layered architectures diagrams.....	28
Figure 3 – Top level view when a developer opens Strata on the entire JEdit codebase.....	29
Figure 4 – Strata after the developer removes the modules other than org.	31
Figure 5 – Strata with the search module being selected.....	32
Figure 6 – Relo started by opening EllipseFigure.....	34
Figure 7 – After adding the method basicMoveBy (by using the more items menu).....	34
Figure 8 – Clicking on the class to show its buds	34
Figure 9 – After clicking on the inheritance buds.....	35
Figure 10 – Expanding the class AbstractFigure and the method addFigureChangeListener.....	36
Figure 11 – Asking for callers of addFigureChangeListener.....	37
Figure 12 – Visualization of Software Terrain Maps.....	46
Figure 13 – The SeeSoft Visualization.....	47
Figure 14 – Armin view on loading dependencies	48
Figure 15 – Armin view after running a short script.....	49
Figure 16 – Layered layout provided by Deref.....	50
Figure 17 – LDM on an old version of the Ant project.....	51
Figure 18 – A layered view by LDM of the ant project.....	52
Figure 19 – Results of the exploration session shown in the previous section with the Eclipse IDE	53
Figure 20 –Results of the exploration session shown in the previous sections with JQuery.....	54
Figure 21 – The call graph viewer of Field	55

Figure 22 – Class hierarchy browser of Field.....	56
Figure 23 – Part of an exploration session with SHriMP	57
Figure 24 – Part of an exploration session with the TkSee Visualizer	58
Figure 25 – Screenshot of Mylar	60
Figure 26 – Screenshot of Jazz [39]	61
Figure 27 – Survey Questions.....	65
Figure 28 – Typical layered architectures diagrams (as in Figure 2).....	73
Figure 29 – A Layered Architectural Diagram.....	74
Figure 30 – Strata display of the jEdit project.....	75
Figure 31 – Automatically building a layered diagram without dealing with cycles (for the jEdit project).....	77
Figure 32 – Dependencies in the top-level modules of the ant project.	78
Figure 33 – Dependencies in the top-level modules of the ant project (numbering modules).	79
Figure 34 – A depiction of the steps of the layering algorithm on the ant project.....	79
Figure 35 – Results of the layering algorithm.....	80
Figure 36 – Merging the results of the generated layers.	80
Figure 37 – Expanding the browser module in jEdit.....	82
Figure 38 – Breaking a module in Strata.	83
Figure 39 – Navigation Buds on the <code>buffer</code> module in Strata.	84
Figure 40 – Part of an exploration session with SHriMP (as in Figure 23)	88
Figure 41 – Relo showing part of the <code>JHotDraw</code> project. (as in Figure 9)	89
Figure 42 – Layout rule for directed relationships	92
Figure 43 – Layout rule for directed relationships	93
Figure 44 – Clicking on the class to show its buds (as in Figure 8).....	93
Figure 45 –Annotations in Relo	99
Figure 46 – Basic Relo Architecture.....	103
Figure 47 – Architecture with caching support for Relo.....	104
Figure 48 – Architecture with support for Strata.	105
Figure 49 – RSSOwl with cycle breaking	110
Figure 50 – Traditional partitioning with RSSOwl.....	111
Figure 51 – Diagram width with and without cycle breaking compared to ideal.	111
Figure 52 – Strata display of the jEdit project (same as Figure 5 and Figure 30).....	112
Figure 53 – Strata display of the <code>buddi</code> project	113
Figure 54 – Strata display of the view and controller of the <code>buddi</code> project	113
Figure 55 – One of the higher-level views of System A.....	115
Figure 56 – Business domain view of System “A” using Strata.	117
Figure 57 – Background for the Ant task.	119
Figure 58 – The Ant task.	120
Figure 59 – The test case for the Ant task.	121
Figure 60 – Hints provided for the Ant bug.....	121
Figure 61 – The Lapis task.....	122
Figure 62 – Hints for the Lapis task.....	122

Figure 63 – Questions Programmers Ask During Software Evolution Tasks – Part I.....	123
Figure 64 – Questions Programmers Ask During Software Evolution Tasks – Part II.	124
Figure 65 – Milestone 4 with Lapis using Relo.....	127
Figure 66 – Milestone 5 with Lapis using Relo.....	127
Figure 67 – Milestone 6 with Lapis using Relo.....	128
Figure 68 – Users progress improvement with Relo.	130
Figure 69 – User 5’s actions during the study.....	130
Figure 70 – User 8’s actions during the study.	131

TABLES

Table 1: Difficulty of understanding the project	66
Table 2: Users saying documentation technique was used effectively to assist in understanding project.....	66
Table 3: Comparing current and wanted techniques for all survey participants ..	67
Table 4: For software immigrants (developers examining internal code)	68
Table 5: Tasks done by user when they were examining the code.....	69
Table 6: The Java Relationships Model	94
Table 7: Projects used in determining Strata usefulness	108
Table 8: Results from expanding the projects in Strata.....	110
Table 9: Users performance on Study Tasks	128

1. INTRODUCTION

As software systems grow in size and use more third-party libraries and frameworks, there is a large need for developers to understand unfamiliar large codebases. This thesis presents two tools, Relo [79][80][81][82] and Strata, which support developers' understanding by creating diagrammatic representations of explored code. As the developer explores relationships found in the code, these tools automatically manage the context in a visualization helping build the developer's mental model. Relo and Strata help developers explore code and select the important subset of code artifacts and relationships to display using various visual constraints to ease the comprehension of the targeted code.

1.1. THE PROBLEM

The growth in size and complexity of software has resulted in developers facing increasing difficulties in comprehending and maintaining a coherent mental model of the code. While design documents can help in getting an understanding of the projects, they often do not exist. Even when such design details are available, they typically only reflect the initial design of the system, as design changes are often not reflected in the documentation. Furthermore, even when available, the design documentation often does not cover all aspects of the software system. Understanding is especially hard when using components and frameworks from many different providers. Even the rise of open source, commonly heralded for making code available for reuse, has contributed to these problems, since open source developers often skimp on documentation, letting the source code speak for itself [51].

1. INTRODUCTION

Techniques like object-oriented programming and design patterns have helped control complexity in large projects by allowing developers to create and use appropriate abstractions and encapsulate inessential details. Unfortunately, these techniques make certain parts of program comprehension harder, requiring a developer reading the code to follow multiple forms of relationships. For example, following a function call, once a simple task, now also requires keeping track of inheritance and polymorphism. This complexity brought about by the interaction of multiple types of abstraction mechanisms forces developers trying to understand code to remember the involved interactions. Developers thus need tool support to keep track of and appropriately show these different relationships.

Program understanding studies have found understanding to be an important and large task, with new developers to a project spending 80% of their time understanding code and more experienced developers in the projects spending around half of their time understanding code. Beyond the productivity overhead, difficulties in program comprehension can result in significant other problems. For example, new code will often not be consistent with the rest of the project and this inconsistency caused by the new code will gradually result in the deterioration of the codebase.

When providing the developer with the right code elements and relationships for a given task, these elements and relationships need to be shown in a manner appropriately emphasizing the right elements and the relationships. Consider UML Diagrams – Class Diagrams emphasize the classes, attributes, operations, and the relationships between them, while Sequence Diagrams or Collaboration Diagrams emphasize the interaction between the given classes/objects. While UML diagrams were created to support communication of code and design issues, using UML tools have their limitations. The reverse engineering capabilities provided by such tools to generate diagrams directly from source code have too much information. Users typically complain that the amount of information in resulting diagrams is overwhelming [2], and thus such tools only have a 10% adoption by developers [3].

Even when having a set of systematically created diagrams, Cherubini et al. [17] found that there is a need for the diagrams to be adapted to the current task of the user. Their study, conducted at Microsoft, observed the use of diagrams created for a software system. With the exception of new hires, diagrams were found to be of little use. The main limitation again was that the diagrams did not show the appropriate code elements – they either contained either too few or too many details when compared to the diagrams that would have been useful for the task.

Tools which do not overwhelm users have required users to write scripts to bring the shown information under control. Work in other domains to help users navigate and comprehend large information spaces have also had simi-

lar scalability limitations, and have suggested the use of analyzing users' tasks to get around the scalability limitations by providing task specific visualizations [31][37].

1.2. PROGRAM COMPREHENSION

Program comprehension has been studied for over 20 years (as described later in section 2.1). When new to a project, a developer's initial goal is to focus on understanding the code. However, understanding code is only sometimes the developer's primary goal. More often program understanding is secondary to other coding tasks [20]. A developer might be trying to fix a bug and while doing that might need to understand some part of the codebase that he has not seen before. For developers understanding is often secondarily needed to accomplish their primary task. Developers thus need to be able to focus on their primary task and have an intuitive interface tightly integrated with their development environment. Comprehension tools that require developer training or require the developer to author a script to use the tool will not be of much benefit to developers.

Developers have been shown to build a number of different types of mental representations or *mental models* while understanding code [23][88]. For large projects, experience developers build these mental models mostly using an opportunistic strategy focusing only on code elements relevant to the task at hand [54]. Supporting such an opportunistic code exploration strategy is therefore needed in helping developers' comprehension.

1.3. APPROACH

This thesis takes a user-centric approach to assist developers in code understanding. The code understanding process starts with developers examining the code structure. This structure involves both code elements, such as classes, fields and methods, and code relationships such as inheritance, method calls and package containment. Graph based diagrams can help comprehension by showing relevant code elements and relationships of the code structure. However, existing diagramming tools are not effective at helping users understand code. This thesis goes around the limitations of current tools and demonstrates the value of four code ideas:

1. Existing program comprehension tools overwhelm users by using an unfamiliar notation that must be learned. Showing code using **familiar** diagrams makes it easier to absorb the information. Such diagrams help reduce the amount of information that needs to be absorbed for the current task. These diagrams should be based on popular diagram types created by developers [18][28] and have been known emphasize important properties of the code. This thesis builds support for program comprehension

1. INTRODUCTION

by using visualization representing the two of the most commonly created diagrams types of code. The first, supported by *Relo*, is based on UML class-diagrams, which emphasize the details in classes and the relationships between them. The second diagram, supported by *Strata*, is based on commonly drawn layered architectural diagrams and provides an overview of the components of a project. Examples of such diagrams are shown in Figure 1 and Figure 2 respectively.

Showing familiar diagrams has a number of requirements. Current visualization tools show all nodes in a similar manner and all relationships in a similar manner – with the only differences being that of color based on the node type, such that Java packages have a different color than classes or methods (as opposed to packages looking diagrammatically like traditional packages). Similarly for relationships, inheritance in class diagrams is often drawn by developers as a relationship directed upwards with the arrow always being closed. *Relo* and *Strata* try to mimic such defaults in its diagrams. Furthermore, beyond appearance relationships can also be displayed via visual constraints. In *Relo*, vertical layout is used for inheritance hierarchies, while call trees are displayed from left-to-right and containment layout is used for items inside packages and classes.

2. Existing diagramming tools show the entire codebase in a single diagram and therefore overwhelm users. **Focused** diagrams are thus needed to show a specific part of the codebase relevant to the developer's task. Showing a relevant method in such diagrams does not mean that other methods in the same class need to be shown. These diagrams represents one of the many architectural views needed to understand the different aspects of the project [19][46]. Such diagrams are similar to those drawn by developers on paper [18] (described in section 2.1.2), but are not available in today's UML tools [9][10]. Such focused diagrams are important as they omit unnecessary details that might otherwise overwhelm developers.
3. Building focused diagrams requires knowing what parts of the codebase matter to the developer's task. Tools need to allow users to **interactively explore** and build a diagram. Existing diagramming tools show focused diagrams only if the developer selects a set of code elements before generating a diagram. They do not help developers find relevant code elements. Unfortunately, as is common in information seeking tasks, developers trying to understand parts of a codebase do not yet know which code elements are involved in their task. Exploration support can be provided by allowing users to follow relationships in a lightweight manner, such as by directly clicking buttons in the shown diagram. Similarly, buttons can allow for easy removal of irrelevant items, and can help diagrams represent a focused view of the current task, thereby assisting the

developer in remembering code elements associated with the current task. Supporting exploration allows for changes in the developer's understanding of both his task and the underlying code by supporting changes to the partial diagrams. In contrast, traditional diagramming tools only show a static output and behave in a manner similar to paper.

Relo and Strata show buttons for exploration when nodes are selected. Users can add or remove code elements and also access relationships on the various methods and follow them. Prior work on program comprehension [64][73][84][87] has shown that while developers may examine small programs systematically and exhaustively, large codebases are not explored that way; instead, developers follow an as-needed exploration strategy, examining only the artifacts they think they need.

4. Interactive exploration of diagrams helps developers understand code when it is their primary goal. However, program comprehension is often a **secondary task** to bug fixing, feature addition, performance tuning, refactoring, etc. In such cases, there is a need for the shown diagrams to be integrated within the IDE. Further, as developers continue to explore code in the IDE, there is a need to link their explorations in traditional editors to update diagrams. Thus a developer adding a feature to a project and getting lost in the codebase, should be able to ask the tool to show a diagram of his current exploration to provide a starting point for his understanding.

Thus Relo and Strata are designed to appear like commonly made code diagrams, displaying parts of large codebases and allowing user to interact with the diagrams and use the diagrams for exploring to help reduce developers cognitive overhead.

Further, developers examining an unfamiliar implementation by going through the code have problems staying oriented in the project and maintaining an organization of visible code elements [26]. Relo and Strata therefore show a diagram that remains consistent and shows nodes in predictable places based on both the needs of the diagram type and the current task. As developers understand code, their *mental model* consists of involved code elements and the manner in which they interact. This mental model changes as the developers get a better understanding of both the current code and the problem that they are trying to solve [12][42][56][61]. A program comprehension interface therefore needs to support developers understanding task by not only allowing developers to examine different parts of the code but also by consistently managing this changing mental model. Supporting such diagrams consistently means not only providing developers fine-grained control in adding or removing code elements to a diagram, but allowing such changes to only incrementally modify the diagram, i.e. instead of drastic glo-

1. INTRODUCTION

bally optimal re-layouts on each change there is a need for a layout that tried to maintain properties in the current diagram.

With support for these code ideas, additional capabilities can allow diagrams to be used for communication among developers or for allow users to build more complex queries based on the viewed items. Queries can allow users to ask complex questions to the tools. For example, Relo allows the user to select multiple nodes and then let the tool 'navigate about' to find out how the nodes are connected.

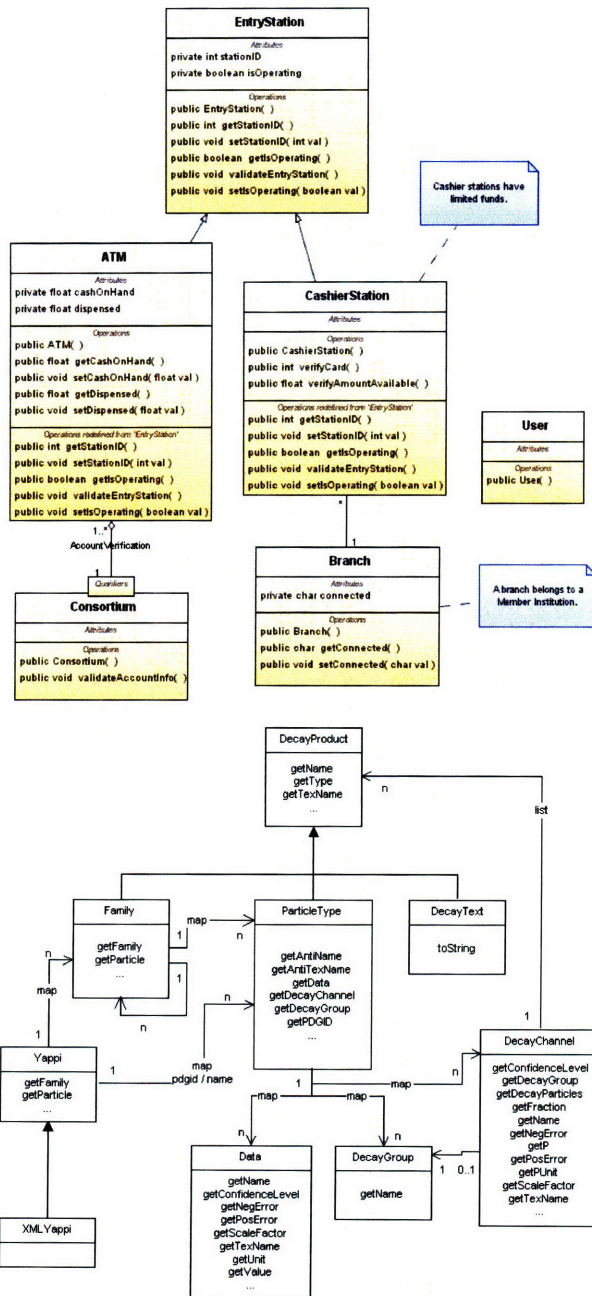


Figure 1 – Typical class diagrams¹

¹ The first images returned by Google on searching for “Class Diagrams” Sources: http://developers.sun.com/jenterprise/learning/tutorials/jse8/uml_class_diagram.html and <http://java.freehep.org/freehep1.x/yappi/ClassDiagram.html>

1. INTRODUCTION

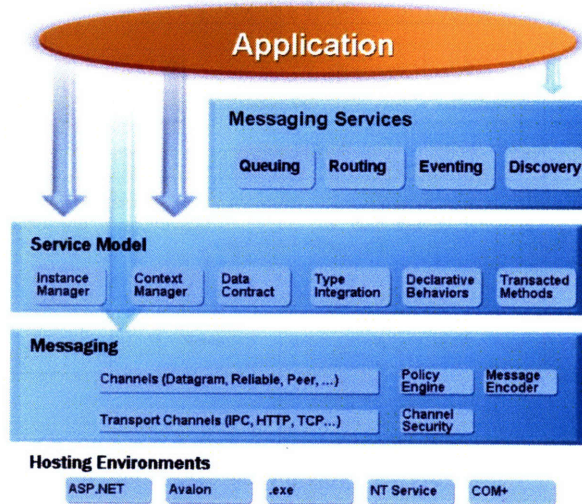
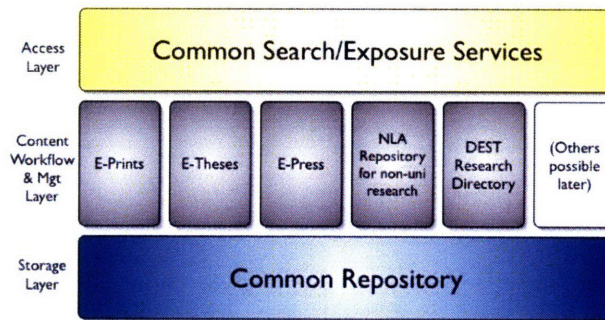
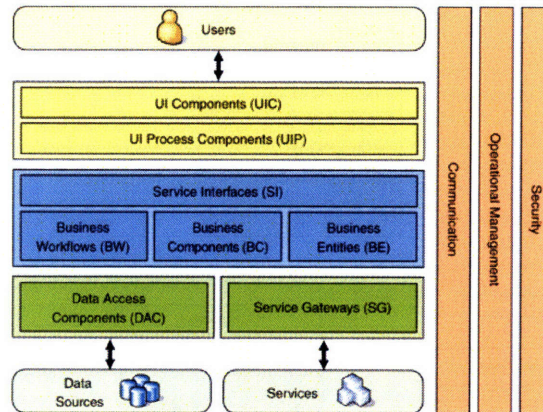


Figure 2 – Typical layered architectures diagrams²

² Source: The first images returned by Google on searching for “Layered Architecture”. Sources: <http://msdn2.microsoft.com/en-us/library/ms978689.aspx>, <http://ausweb.scu.edu.au/awo4/papers/refereed/treloar/paper.html> and <http://bartdesmet.net/blogs/bart/archive/2005/09/01/3517.aspx>

1.4. WALKTHROUGH

We present a walkthrough through two scenarios, one trying to get an overview of a codebase using Strata and another trying to show detailed relationships in a project using Relo.

1.4.1 Layered Overviews with Strata

Strata builds high-level diagrams of dependencies in software projects and actively helps developers to explore, understand, and get an overview of the underlying project. These diagrams are similar to commonly available layered architectural diagrams. The diagrams represent the current selection (or the entire project) and is built by aggregating dependencies and code elements in the project. A developer can then use Strata to interactively either focus on a relevant portion of the project or remove irrelevant portions. Developers can explore and find relevant (potentially crosscutting) concerns within the implementation and use them as modules for future explorations. The focus is on providing a mechanism to get a rapid high level visualization – to provide good defaults without needing developer intervention, and provide developers with exploration mechanisms to find the relevant portion of the code that they might be interested in.

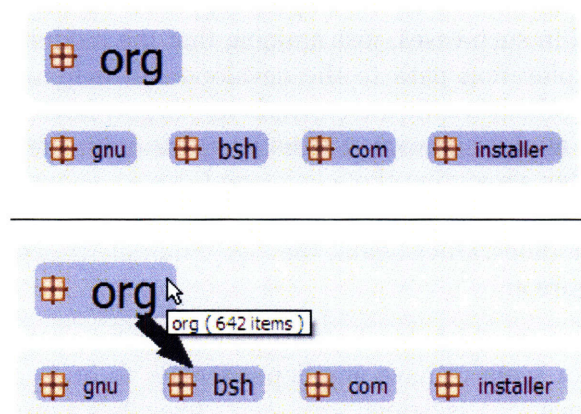


Figure 3 – Top level view when a developer opens Strata on the entire JEdit codebase. The diagrams show the `org` package builds on (and depends on) one or more packages at the lower levels.

(top: the default view, and bottom: when the mouse is moved over the `org` package)

Consider a developer working with Strata on the JEdit project [5]. JEdit is an open-source Java based editor consisting of a fairly extensive plugin and scripting framework. The project consists of over 500 classes and over 150,000 lines of code (as measured by `wc`). When working with such projects it is difficult to get an overview of the various components. Strata tries to provide support to help the developer understand the JEdit project by exploring through overview visualization of the code components. The developer needs

1. INTRODUCTION

to only right-click on the project in the Eclipse IDE and open Strata from the context menu, which produces a view similar to Figure 3.

The figure shows that at the highest level the project consists of a number of modules with the `org` module being the largest. Strata also shows the exact size of modules in tooltips allowing the developer to realize that the `gnu` and `com` modules consist of very few classes. The modules are shown in a layered view, with each module in a layer depending on one or more modules in the layer below it. In this case, the `org` module depends on a module below it, and moving the mouse over the `org` module indicates by an arrow that it depends on the `bsh` module. An experienced Java developer will likely recognize that the module `bsh` is related to scripting support³, the `installer` module is related to an installer for the editor, and the small `com` and `gnu` modules consist of overrides to the externally provided functionality provided in these modules. Since developers are mostly interested in the code for the current project, Strata by default shows only the dependencies in the provided source, and does not include code provided in external libraries.

At this point, the developer can select and remove the smaller modules from the view. Doing so causes Strata to automatically expand the `org` module, to show that it consists of the `gjt.sp` module and the `objectweb` module. Again, the module names are not very useful, but represent the best guess that Strata starts with, and in such cases, just noticing that the module is large can recommend an exploration path to the developer. In helping developers with top-down comprehension, Strata expands the largest module when there are three or fewer modules shown. Since there are only a few modules being shown (even after expanding the `gjt.sp` and `objectweb` modules) Strata expands the largest module again: `gjt.sp`. This module consists of the `jedit` and the `util` modules. Once again, the `jedit` module is expanded automatically to give Figure 4.

From Figure 4, a few high level observations can be made. The `objectweb` module does not seem to have any dependencies to or from the rest of the shown code. Since `util` is below `jedit`, `util` likely has a number of code elements depending on it. The `jedit` module consists of a number of modules dealing with, among other things, the `gui`, a `textarea`, and `search` support.

³ Developers not recognizing these modules can explore and discover their functionality as shown with the `org` module.

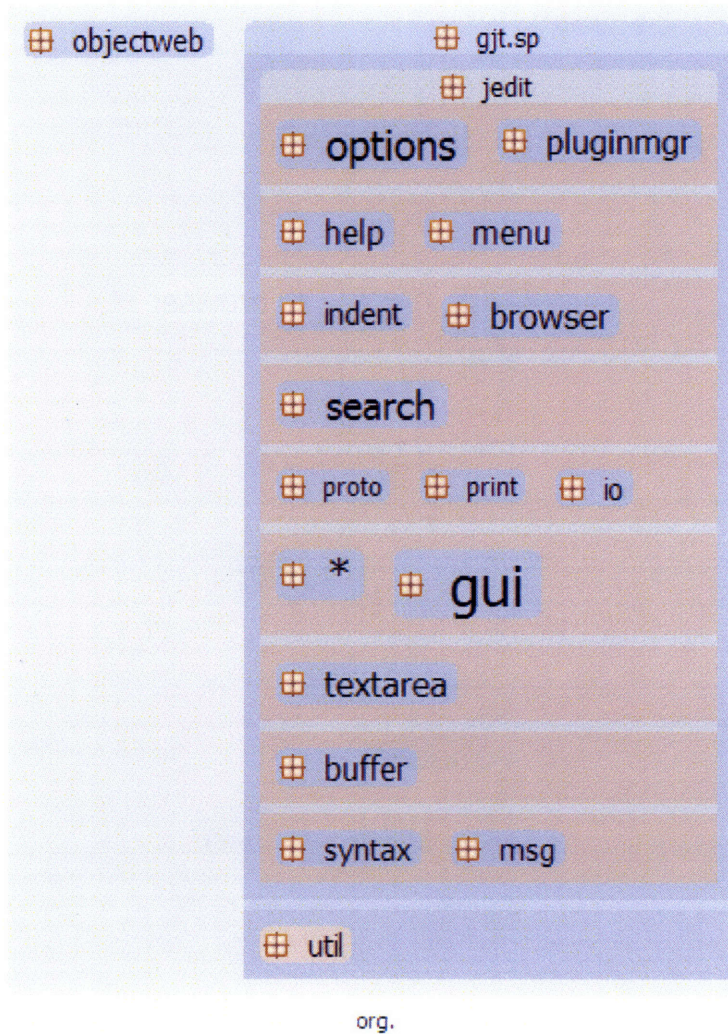


Figure 4 – Strata after the developer removes the modules other than `org`.

When the developer mouses-over the modules, dependencies to and from the module are shown. Strata uses colors to indicate which modules depend on the module being moused over. The current module is colored gray, modules which build on top of the current module are colored in a lighter shade, and the modules which the current module depends on have a dark shade. Modules that have dependencies both to and from the current modules are colored with the same color as the current module. Figure 5 shows the visualization with the developer having selected the `search` module – in this case beyond showing dependencies and highlighting appropriate modules, the developer is also shown buttons to show any other dependencies associated with the `search` module. The dependencies are shown using arrows, with thicker arrows indicating a larger number of dependencies.

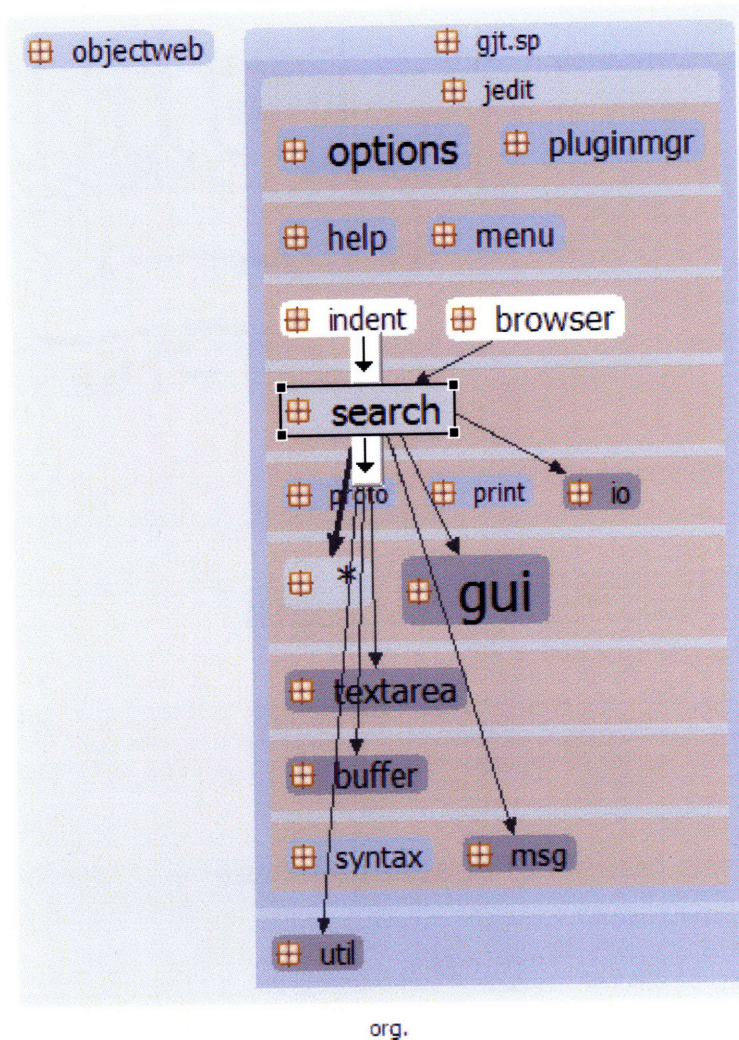


Figure 5 – Strata with the search module being selected

Again, the developer can continue the process. He can decide that certain dependencies represent minor temporary inconsistencies with the design, and can right-click on them to hide them asking the tool for an updated layout, or can continue exploring with the tool by asking it to remove some of the modules. Once at a detailed enough exploration level the developer can ask the tool to show the dependencies in the code, or to view the source of the involved modules.

1.4.2 Detailed Relationships with Relo

Relo is designed to help developers explore and understand small focused parts of large codebases. Such a small manageable parts of the code can represent a developers task and not include irrelevant details to the task, thus helping in developer productivity [13][67]. Relo visualizations are shown in

the commonly available UML Class diagrams which emphasize details of classes in the codebase. Relo further provides an interactive exploration interface for developers to view, select, add, and remove code elements, and presents them graphically to assist in the comprehension of the shown code.

Relo visualizations start with a single code artifact, such as a package, class, or method, from which a developer can browse different relationships to interactively add or remove code artifacts. As the user interacts with the diagram, Relo automatically lays out the diagram, with layout rules that try to put components in predictable places (consistent with UML diagrams) based on their relationships: e.g., vertical layout for inheritance hierarchies, left-to-right for call trees, and container layout for package and class containment. In addition, Relo allows zooming in to view and edit code using text editors embedded in the diagram. Developers can therefore abstract to a high level, or focus-in to see the actual code.

Relo is built with the intent of supporting developers exploring the static structure of code, in a UML like visualization. We illustrate how Relo would be used by a developer for typical comprehension task. For this example, we use a task similar to that tackled by JQuery [40]. The task involves a developer working with the JHotDraw [6] project, a GUI framework for building drawing applications consisting of figures like rectangles, triangles, ellipses, lines, etc. Suppose the developer needs to add a feature that operates on figures and would therefore like to understand how to manipulate figures. In attempting this task, the developer will try to understand the code, likely by taking a few steps:

1. Find a class implementing figures.
2. Understand it by examining a few methods in this class.
3. Go up the inheritance tree, to find a suitably general base class representing all figures.
4. Find code that manipulates figures by calling methods in this general base class.
5. Select an appropriate manipulating class, and examine its methods to duplicate relevant functionality.

A developer following the above steps will typically make rapid progress in the first three steps: finding a starting class (using simple heuristics and search queries), examining it, and selecting an appropriate base class. However, when the developer attempts step 4, i.e. selecting a method that is called for manipulating figures, and tries to examine the callers he will have difficulty in keeping track of the various examined code artifacts. The difficulty will occur because of the desire to maintain a context, by examining the roles of nodes connected by inheritance, containment, and method calls relation-

1. INTRODUCTION

ships, i.e. he will need to remember at the minimum 3 relationships and 6 code artifacts (corresponding to the 3 steps above), something that is larger than human short term memory [57].

This scenario would be simple with Relo. As the developer looks at the code, he will find that `JHotDraw` has a number of packages, with one being called `figures`. The developer would look at that package, and find that the class `EllipseFigure` is a relevant starting point for his/her exploration. The developer would then just need to select the class, and open it in Relo (as shown in Figure 6).

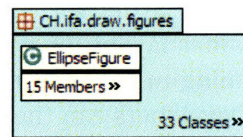


Figure 6 – Relo started by opening `EllipseFigure`

On finding the class `EllipseFigure` and starting Relo with it, the developer is presented with Figure 1. The figure shows that the class has 15 members, and the developer clicks on the menu to see a list. Considering the method `basicMoveBy` as potentially interesting, he clicks on the method name in the menu and thereby adds the method to the diagram for future examination (Figure 7). Once it is added the developer clicks on the class to be presented with a handle indicating the class inherits from another class (shown in Figure 8). The developer clicks on this handle to show superclasses, and therefore continues his exploration to find a relevant base class by clicking upwards (Figure 9).

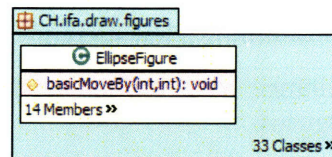


Figure 7 – After adding the method `basicMoveBy` (by using the more items menu)

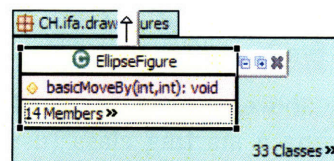


Figure 8 – Clicking on the class to show its buds

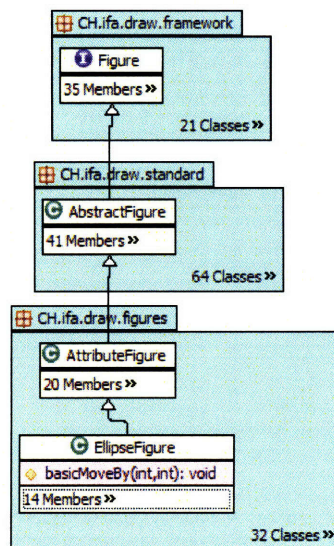


Figure 9 – After clicking on the inheritance buds

With the developer having an idea of the inheritance tree of figures in the project, he chooses to expand the `AbstractFigure` class. After double-clicking to see all public methods, the developer removes some methods irrelevant to his task (manipulating figures), and examines the remaining methods to select one for expansion. Deciding that the `addFigureChangeListener` method is part of the general framework for changing figures, the developer decides to expand it.

The developer is presented with the Figure 10, which shows the implementation of the method. After finding the implementation relevant, the developer will want to continue with the original plan of finding the manipulators of figures. In this case, the developer will want to find a caller of `addFigureChangeListener`. To do this, the developer collapses the `AbstractFigure` class and clicks the caller handle for the method in the interface (Figure 11). `Relo` is now acting as both a call-hierarchy browser and an inheritance-hierarchy browser.

Once presented with Figure 11, the developer can easily select the relevant classes that manipulate figures, and does not have to worry about the connecting inheritance, containment, and method calls relationships. As the developer continues with his task, he can go on to build a larger visualization and choose to refine the generated diagram at every step, so that the visualization helps in his understanding of the codebase.

1. INTRODUCTION

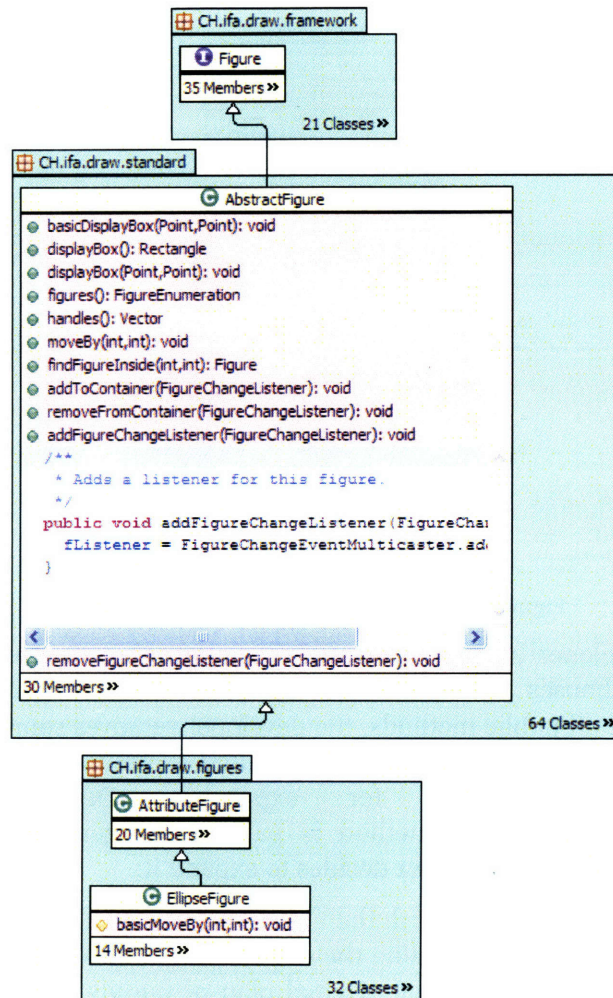


Figure 10 – Expanding the class `AbstractFigure` and the method `addFigureChangeListener`

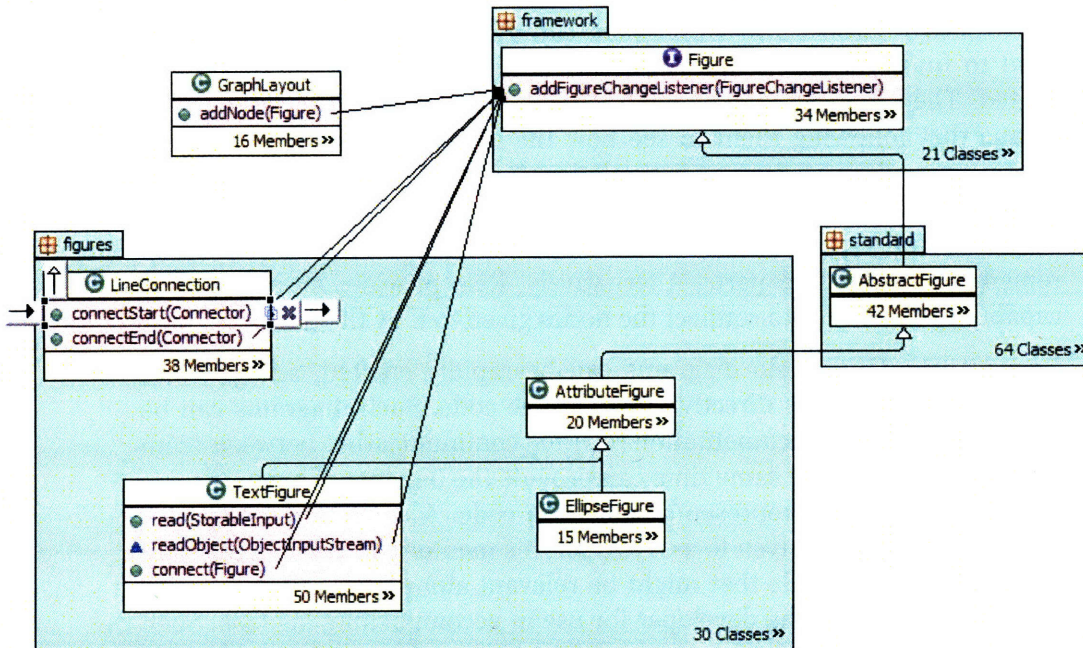


Figure 11 – Asking for callers of `addFigureChangeListener`

1.5. SCENARIOS

There are a number of scenarios that Relo and Strata can be especially useful for:

1. **Concrete Context Representation:** As a developer performs a task he often needs to explore code. During this time a Relo diagram can represent his mental model and supplement his short-term memory, by automatically creating and updating the relevant visualization of the code. At the same time, a Strata diagram can help a developer situate in the codebase as he explores it. The developer can be shown the modules being explored directly, those modules that depend on it, as well as those modules that are used by the current modules.
2. **Lost in Code:** As the developer explores code in a traditional IDE, he can often get lost in the code base. Relo's capability of linking to the IDE to track navigation in the background can be used if the developer gets lost or forgets an important relationship that was traversed. The developer can ask Relo to launch a visualization based on recent navigation history. The

generated diagram can help the developer see how recently visited code was related and can further suggest next steps accomplishing their tasks.

3. **Connecting Elements:** Developers can get into situations when they need to find and understand how two or three code elements are connected. The large number of relationships between these code elements means that following them to see how the items are connected also requires remembering the various relationships traversed and further involves a number of wrong connection hypotheses. Using a visualization to help keep track of the visited nodes and relationships can help assist in remembering explored parts. Alternatively, Relo provides an autobrowse capability which tries to connect the nodes given to it by the user.
4. **Communication:** Relo diagrams can be rapidly created, and produce documentation that is directly linked to the code. Such diagrams can be used as structured documentation [62] for communicating between team members, while at the same time can provide the benefits of having an accurate diagrammatic representation of the code. A common scenario for this is when a task is given to a developer, his mentor can create a diagram of the parts of the code that might be relevant along with comments and send the diagram to the developer for use in accomplishing his task. Such diagrams are a visual and more intuitive form of concern graphs [68] which have been shown to increase developer productivity.
5. **Impact Analysis:** Overview diagrams provided with Strata can be used by developers to gain a high-level understanding of components in a project. When a developer has a need for modifying a set of 4-5 classes that are involved in a task, he would also need to understand the impact of any change, i.e. (a) which classes build on the original set of classes and would feel the impact of any functionality change, or (b) which classes do the original set of classes depend on. For example, a developer modifying the encoding of identifiers in an application, will want to know which all classes depend on it and therefore how much of the codebase will need to be retested to verify a correct implementation. While non-visualization approaches can provide this information, showing a layered architecture view will more succinctly show the effect of the various nodes in the formed dependency.

1.6. CONTRIBUTIONS

This thesis shows that **understanding for software developers can be effectively supported by interactive exploration using focused diagrams of familiar representations of code**. These ideas have been combined to build two tools: Strata, which displays using the popular layered

architectural diagrams, and Relo, which is based on UML class diagrams. In particular, this work has a number of contributions:

- An approach and tools for allowing developers to have an intuitive controlled exploration of relevant elements for a development task by providing support for managing the amount and presentation of information to the developer based on his/her interaction with code elements.
- An approach for building visualizations allowing users to work with their current suite of tools while still being linked to a diagrammatic representation of their current or past exploration(s).
- A visualization technique and implementation to show multiple relationship types and code elements in an intuitive manner.
- A lightweight approach and tool to provide developers a high level understanding of inter-module and intra-module dependencies, i.e. to easily understand the various components that affect and build on a current module, and to easily understand the various components involved in the current module respectively.
- A survey of developers' experiences on the effectiveness of various documentation techniques.
- A survey of the amount of dependency cycles in high-level modules of popular publically available projects.
- A qualitative evaluation of (Strata) a light-weight tool for showing overview diagrams of modules by users on their own projects.
- An in-depth evaluation consisted of feedback from users in the field and controlled quantitative and qualitative of (Relo) a static visualization based program comprehension tool providing support for large codebases.

1.7. OUTLINE

This thesis starts with a brief overview of previous work that Strata and Relo have been informed from in Section 2. Results of a survey of developers experiences on the effectiveness of various documentation techniques are then presented (section 3). This thesis then presents the main idea of Strata (section 4) and Relo (section 5). The implementation of the tools is presented in Section 6, and evaluation in Section 7. We finally discuss future work in Section 8.

2. PREVIOUS WORK

A significant amount of work has been done in helping developers work with and understand large codebases. While a deep understanding of how developers work with and understand code has evolved, there has only been limited usage of program comprehension tools. Below we describe work on understanding developers' behavior and cover previous tools helping developers either gain an overview or deeply explore their codebases. We also describe other tools that have been built to help developers work with large projects

2.1. DEVELOPER BEHAVIOR

Ethnographic studies of developers' behaviors have shown this fact program comprehension to be an important and large task. However, partially because comprehension is often a secondary task, studies measuring it offer varying reports of their degree of importance for developers. Corbi reasoned that more than half of the effort in accomplishing a task for the programmer is in understanding the system [20]. Studies by Davison et al. have shown new project members spending 60%-80% of their time understanding code, with the number dropping at most down to a low of 20% as the developers gain experience in the code that they are working with [22]. Another study by Singer et al. found developers spending over 25% of their time either searching for or looking at code [76]. In a recent survey of 427 developers at Microsoft, conducted by Cherubini et al. [18], 95% agreed that understanding existing code is an important part of their job function. Further, over 65% indicated understanding existing code once or more times a day (with over 25% indicating understanding multiple times a day). Another study of 157 devel-

2. PREVIOUS WORK

opers at Microsoft by LaToza et al. [49] found that developers spent roughly equal amounts of time understanding code as other tasks such as designing, unit testing, and writing. On the other side a study by Perry et al. did find code inspection to only take 5% of developer time [60], though the number can be possibly explained because the study required developers to self-report their current task and program understanding typically is secondary to other coding tasks.

2.1.1 *Understanding Models*

Studies over the last twenty plus year of developer's understanding code have created a number of theories about the *strategies* used by developers, during the understanding *process*, and the developer's *mental model*, i.e. their mental representation of the code [23][88].

Developers typically take one of two strategies, using a systematic strategy or an opportunistic one [54]. A systematic strategy involves the developer going through the code in detail, tracing both control-flow and data-flow abstractions, while building an accurate model of the program. An opportunistic strategy in contrast is used more often by experienced developers in large projects and involves users focusing only on the code elements relating to the task at the cost of being more error prone. Most tools present complete models of the code and therefore support systematic strategies. In contrast, Relo and Strata also support opportunistic strategies by allowing developers multiple navigation steps from shown diagram to obtain diagrams focused on the developer's task.

Systematic and opportunistic code comprehension strategies typically result in a bottom-up model of the code being built by the developer. This involves the developer reading code statements and mentally grouping them in to higher level abstractions [73][74]. This building of developer's mental model has been found by Pennington [64] to happen in two phases: firstly by building a program model consisting of control-flow abstractions, and then by building a situation model consisting of data-flow abstractions and functional abstractions (program goal hierarchy). By supporting both opportunistic and systematic strategies, Relo and Strata can be said to mirror the developer's mental model and therefore support these comprehension processes.

In contrast to bottom-up building of a mental-model, developers also use a top-down strategy for understanding code [14]. This top-down strategy is used by developers either when they are familiar with a domain or are trying to find a starting point for their code exploration. This process involves them starting by building a high-level hypothesis, then verifying the hypothesis by looking for familiar structures, and using these verifications to form subsidiary hypotheses. High-level views of Strata support overviews of the code and therefore enable many types of such high-level hypotheses verification by developers.

Other theories of comprehension provide for processes that combine the above strategies. Letovsky [50] has proposed that developer's model is built up in conjunction with a knowledge base representing the developer's expertise and background, with the process of building the mental model involving three types of hypotheses: why, how, and what. von Mayrhauser added that a developer likely builds not only bottom-up models (program model and situation model) but also a top-down domain model while understanding [95]. Again, Relo and Strata provide support for opportunistic exploration and therefore ease in developer's building mental models of their code. A detailed categorization of these opportunistic questions is done by Sillito et al. [78] and has been used in evaluating the effectiveness of Relo as described in Chapter 7.

2.1.2 Use of Sketching

Relo and Strata use diagrams to bring together separated pieces of functionality into a single focused location. For these diagrams to be easy and intuitive to use, they need to resemble diagrams commonly sketched by developers. Cherubini et al. [18] studied such sketches and found that developers most commonly used boxes-and-arrows diagrams, representing entities and the relationship between them. Diagrams were often used to help keep developers oriented in the big picture. Boxes in diagrams were labeled with text, with the size of the box sometimes encoding the importance or the size of the entity being represented. Boxes were sometimes grouped into higher-order structures using large boxes or dividing lines. Boxes were also sometimes placed next to each other in lieu of arrows. Relationships between entities were usually represented with arrows. They were mostly directed and generally pointed rightward or downward (though some drawing types had different conventions like class inheritance where arrows were pointed upwards).

Relo and Strata support these ideas. They allow developers to explore through box-and-arrow diagrams with code elements placed next to each other to minimize arrows. Relo and Strata's approach of supporting diagram types support other attributes as found by Cherubini et al., also allows for extensions to such as iconic representations for entities (such as database via cylinder, OLAP via data cube, computer via CPU tower and person via a stick figure), circles for states in state-transition diagrams, and other entities such as processes, hardware devices, and UI screens. The other property found in the study that can be investigated for use in Relo would be for the usage of numberings to indicate sequence in diagrams. The study did call for tools like Relo which attempted to integrate reverse-engineering with support for light-weight sketching of the relevant parts of the code. Relo provides this by creating a model of the code in a database, and allowing users for exploring the model either directly in Relo or in the IDE with the explored items showing in Relo.

2. PREVIOUS WORK

Strata focuses on the higher level views when compared to Relo. It provides support for showing modules in the underlying hierarchical structures from the codebase. While not focusing on the more popular class-diagrams, it focuses on providing the connection to the big picture view of the project. In providing high-level views, Strata also aggregates modules and dependencies to present them to users in an organized manner.

2.1.3 Working with Concerns

In software development *concerns* are any matter of interest in a software system. They are often synonymous with features or behaviors. Containment based modularization represents one type of concerns, with concerns also potentially cutting across a number of modules. Examples of concerns include performance, logging and data integrity. A key principle of software engineering has been the *separation of concerns* – the idea of splitting a system into sub-components that have as little as possible overlap in functionality. Separation of concerns eases managing a system since it allows for focusing on particular sub-parts of the overall system. Relo and Strata diagrams can thus be said to be visualization of particular types of concerns.

The potential of effectively working with views of concerns has been long acknowledged. Initial work on concerns focused on representing concerns [35][91] and providing schemas for their interactions [89][36]. A concrete instantiation of concerns on code, called Concern Graphs [68] consist of code elements connected by few types of relationships (calls, reads, writes, checks, declares etc.). Concern graphs are graphs which are typically displayed to users as trees. When created ahead of time and provided to developers concern graphs have been shown effective in representing the relevant portions of code for a code maintenance task. Such concern definition will also be helpful for usage within Relo and Strata.

The main limitation of these techniques is in the effort required in creating the concerns. Approaches for building concern graphs have focused on mining navigation events and using various learning techniques towards inferring the graphs [55][67]. In contrast, Relo and Strata try to get such information directly from users by providing an interactive exploratory interface. Using an interactive interface results in the diagram representing the developers' task, and support for multiple tasks requires just having multiple diagrams, in contrast techniques mining developers' navigation events have limitations when needing to account for developers switching tasks. Further, mining a developer's navigation events for creating concerns graphs can only be helpful for subsequent tasks, and requires interfaces for retrieving indexed concerns.

Relo and Strata visualizations can be said to be interactive diagrammatic representations of concern graphs, in that they only show a manageable part of the code and do not include irrelevant details, allowing a developer to focus

on the important relationships. By providing an interface to help developers build concern graphs Relo and Strata leverage these strengths of Concern Graphs. When examining details Relo uses a relationship model similar to that used by concern graphs. For higher-level views Strata uses graphs consisting of nodes that represent multiple code elements and the different relationships being aggregated into dependencies with a count representing the strength of the dependencies as the number of relationships – this approach of aggregation of dependencies has been shown effective for high-level views [71].

2.2. OVERVIEW TOOLS

A number of approaches have tried to provide high-level views of software projects. These projects try to either help situate a developer in an artificial surrounding, provide statistical information to developers, or focus on helping developers understand the underlying architecture of the codebase. Among these projects Strata focuses on providing a diagram made from commonly used principles in developers sketches and provides for a lightweight interaction for developers to focus in on parts of the project that they need to examine.

SOFTWARE TERRAIN MAPS

Figure 12 shows Software Terrain Maps [25] which are created automatically for developers and consist of hexagonal tiles each representing a fixed number of lines of code. Terrain maps are laid in a manner so as to be stable as the code evolves and therefore are useful in situating developers as they accomplish their tasks. However, such a tool has a limited benefit in helping developers understand and examine code. Looking at the diagram does not provide any information on the major modules of the codebase and how they are connected. In contrast, users of Strata are able to quickly identify modules that they need to ignore and remove from a diagram or need to examine in more detail. Strata diagrams are also more intuitive to developers and are the type that developers would sketch themselves.

2. PREVIOUS WORK

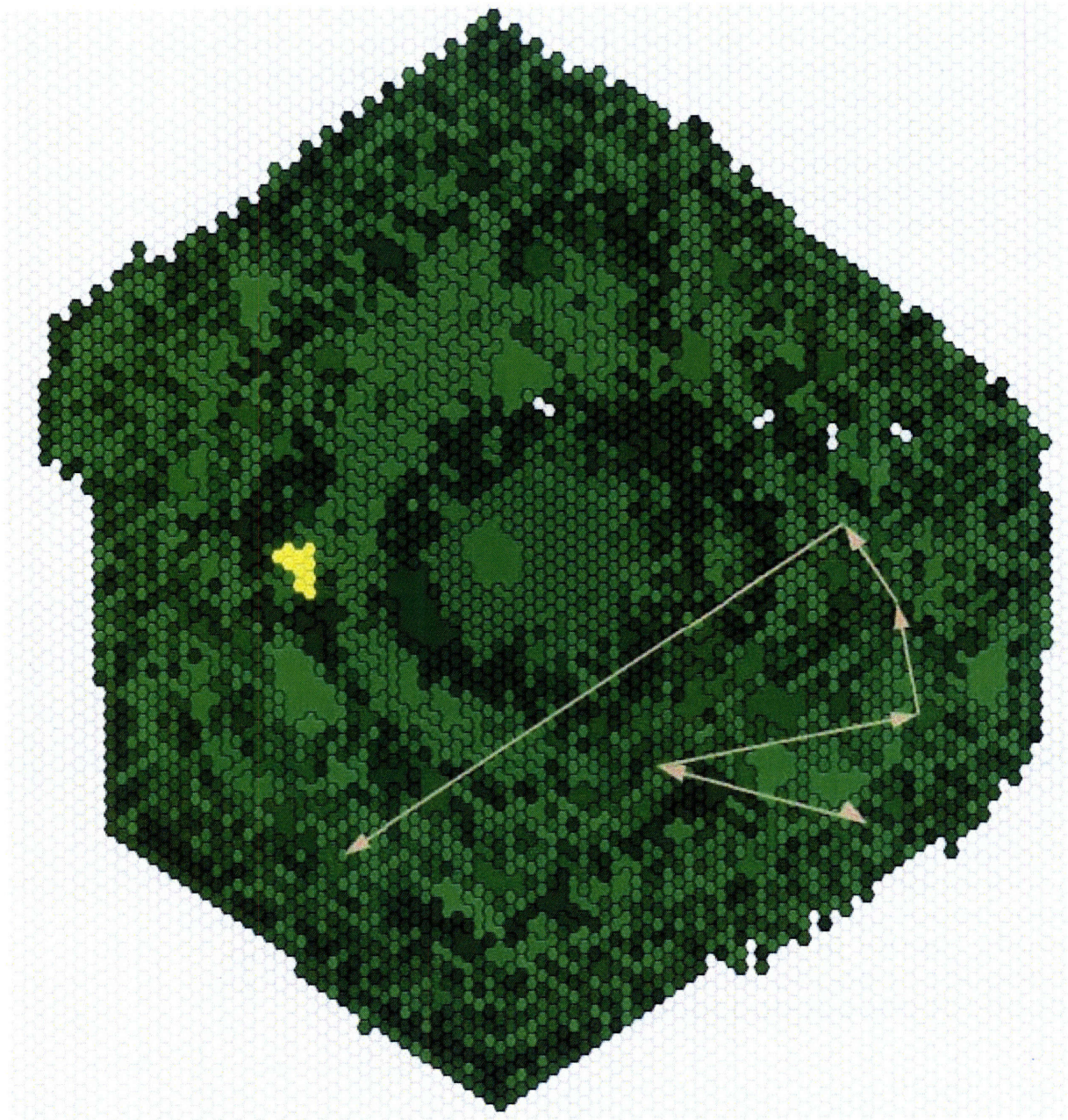


Figure 12 – Visualization of Software Terrain Maps

SEESOFT

The SeeSoft project [29], like Strata, also tries to visually represent large amount of code for exploration. It however uses a line-based visualization that maps each line of source code into a thin row on the screen with files in

the system representing a column on the screen (see Figure 13). While the visualization is helpful in presenting statistical information like the age of documents, the visualization is not able to show relational information such as code dependencies. Relo and Strata both focus on showing these relationships among code elements.

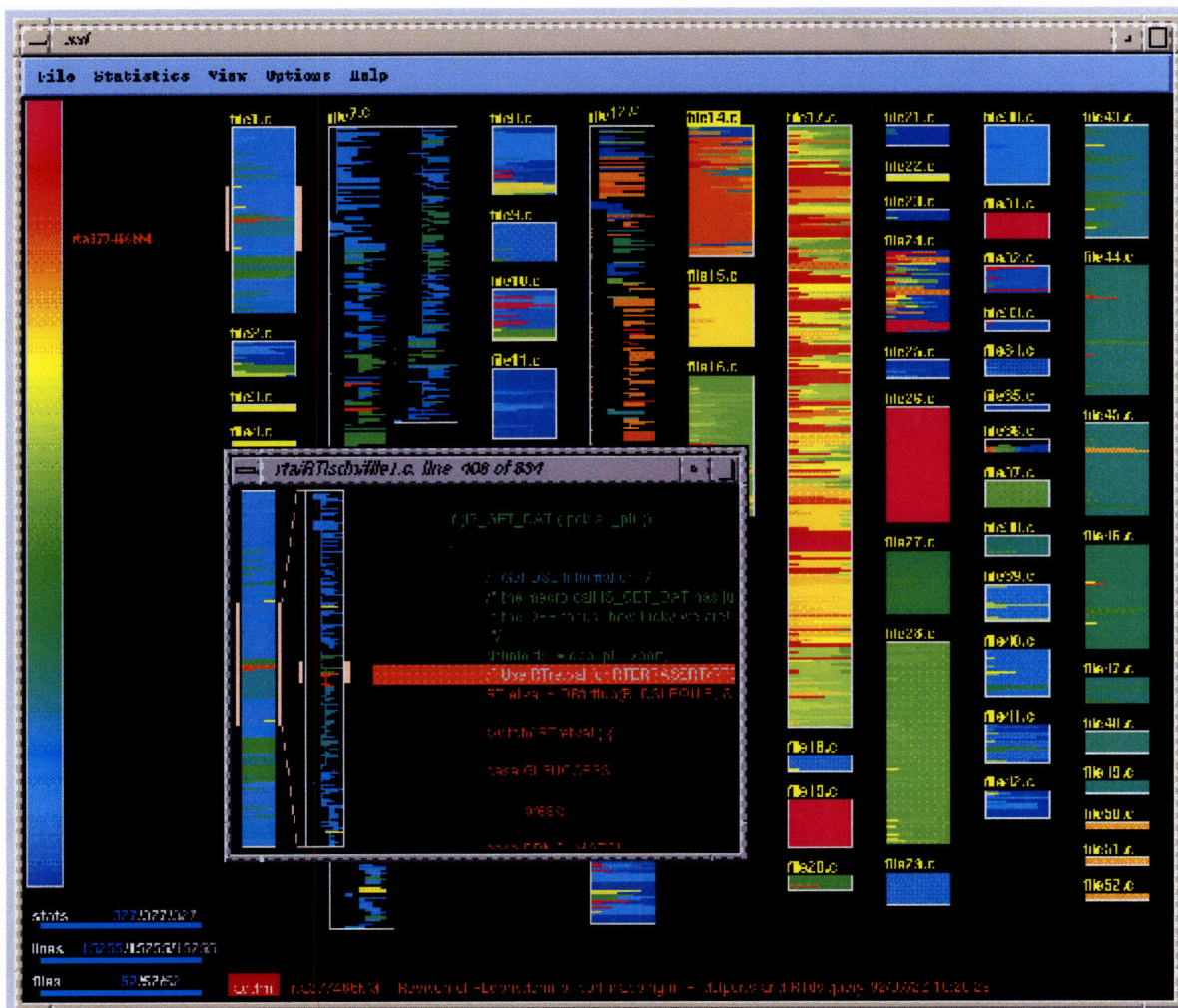


Figure 13 – The SeeSoft Visualization

DALI, ARMIN, AND DEREf

Approaches which try to help developers understand the code by providing an overview of the system have typically required expert input to present users with useful results. The Dali workbench [43] provides for automatic extraction of dependencies but expects users to provide define modules and their contents which are used for building views. While very useful, the workbench is not able to provide a useful view unless users provide to module de-

2. PREVIOUS WORK

scriptions. The ARMIN project [13] works similarly to Dali and goes further in easing the module definition process but still requires developers to provide scripts to perform the modularization. Figure 14 shows the Armin display on initial extraction of dependencies, with Figure 15 showing the view after a short script is run.

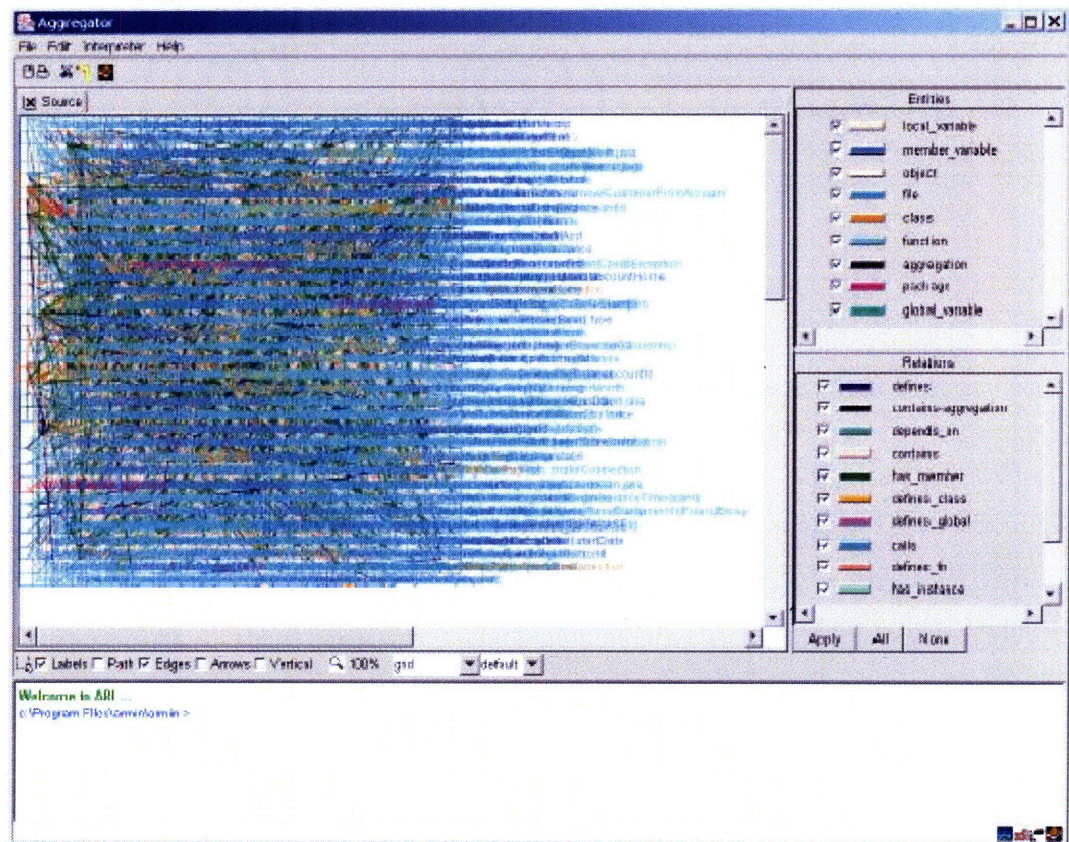


Figure 14 – Armin view on loading dependencies

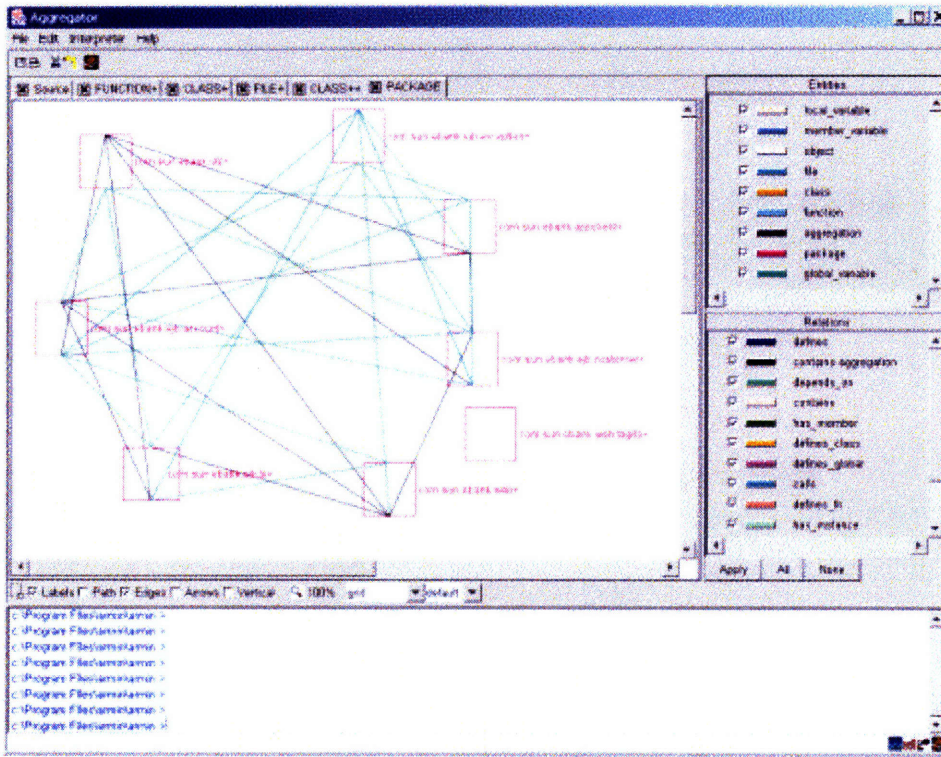


Figure 15 – Armin view after running a short script

The DEREf environment [93] takes things a step further (as shown in Figure 16) by showing modules in a scalable layered view as opposed to the radial layout used by Dali and Armin. Deref however also requires manual input from users to provide module definitions.

2. PREVIOUS WORK

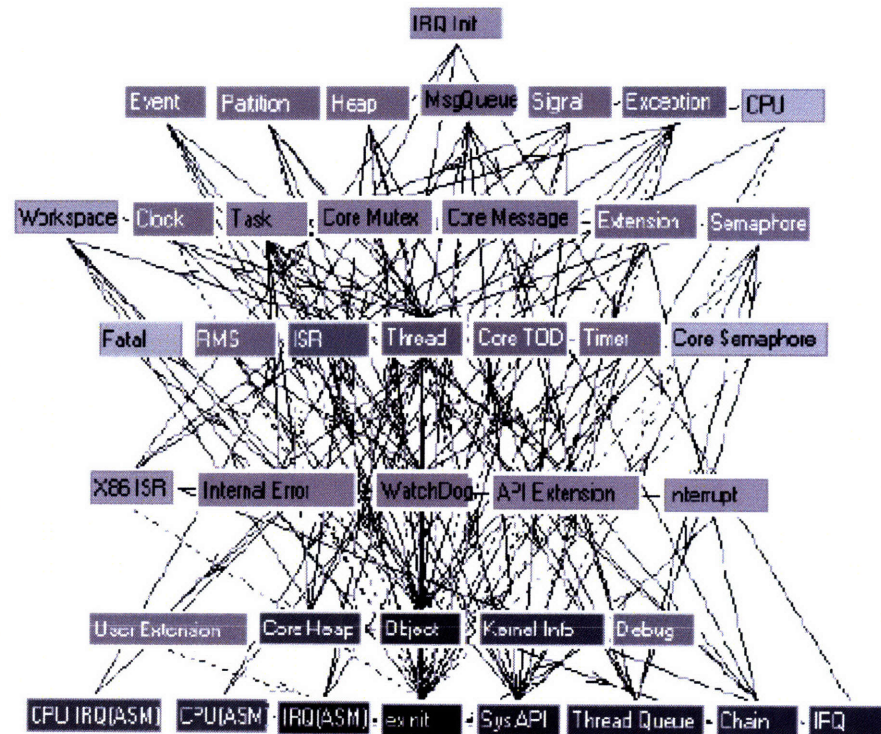


Figure 16 – Layered layout provided by Deref

In contrast to these systems Strata provides a fully automated approach providing layering based on the known package or directory structure, and organizes the modules into a layered view. By being completely automated Strata's results are more lightweight for users and they can more easily get high-level views of the project. Strata's defaults in module contents and layering can be changed by the user to provide accurate diagrams when such information is available. Strata also provides options to focus developers on the unexpected dependencies going upwards in a project, and provides exploration support so that developers can easily get architectural information for any part of the system that they are interested in.

LDM

An approach that has been effective in being automated has not been visualization related. LDM [71] consists of a tool that extracts dependencies and shows them without user-intervention in a matrix form. This approach (shown in Figure 17) while scalable requires user training to make sense of the display. Strata thus tries to bring the benefits of this approach in a more intuitive 'box-and-arrow' that is more easily grasped, and also provides exploration support needed for easily creating multiple diagrams.

While LDM does present a view of the matrix result in a diagrammatic representation (shown in Figure 18), their diagrammatic view has a number of limitations, such as showing modules with cyclic dependencies in a manner similar to modules having no dependencies amongst each other. For example, in the figure, inside the `ant` module the modules `util`, `types`, and `filter`, are have a cyclic dependency – something that is only noticeable when looking at the matrix view. Strata differentiates such kinds of dependencies and further provides an interface for developers to easily explore dependencies by asking it to open modules depending on the current module.

Strata is inspired from the scalability of the matrix representation of LDM. Strata further uses partitioning algorithms similar to LDM, and handles cycles in a similar manner as ordered in LDM’s matrix representaiton.

\$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
org.apache.tools	ant:taskdefs	+ cvslib 1																	
		+ compilers 2					2												
		+ rmic 3					2												
		+ condition 4					12					2	3	1					
		+ email 5					1												
		+ * 6	5	7	4	3													
		ant	+ listener 7																
			+ helper 8												1				
			+ input 9					3							4				
			+ filters 10					3						12	1				
			+ types 11	4	19	7		3	152				17		2	9			
			+ util 12	1	3	3	1		55	1	1	4	13		12				
			+ * 13	11	25	14	20	10	309	4	12	3	6	71	13				
util		+ org.apache.tools.bzip2 14					4												
		+ org.apache.tools.mail 15					1		1										
		+ org.apache.tools.tar 16					4												
		+ org.apache.tools.zip 17					5												

Figure 17 – LDM on an old version of the Ant project

2. PREVIOUS WORK

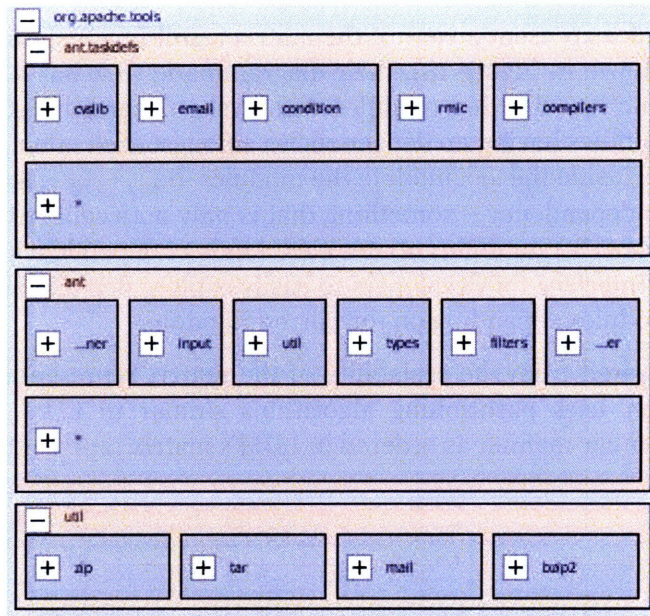


Figure 18 – A layered view by LDM of the ant project

2.3. EXPLORATION TOOLS

Approaches to user-directed exploration of large projects have typically done so by using multiple distinct views each supporting only a single predetermined relationship, like inheritance or method-call hierarchy (see Figure 19). Such views occur commonly as tree widgets in most IDE's, but result in a loss of context when attempting to work with more than one relationship. Developers using more than one tree view need to keep track of how the views are connected. For example, considering the exploration session in the previous section, will result in a traditional IDE looking like Figure 19. As happens often with IDE's, the developer ends up having used a number of tree views – the search view, the package explorer, the inheritance hierarchy view, and the call hierarchy view – and therefore needs to remember at what point he left interacting with one view and switched to the next. The same problem happens also with the editors, which are often all stacked as tabs, making it impossible to see how code elements are connected. To overcome the loss of context like Relo recent tools have tried to bring the different relationships together in a single view. Below we describe such work and discuss their assistance in helping developers explore and understand code.

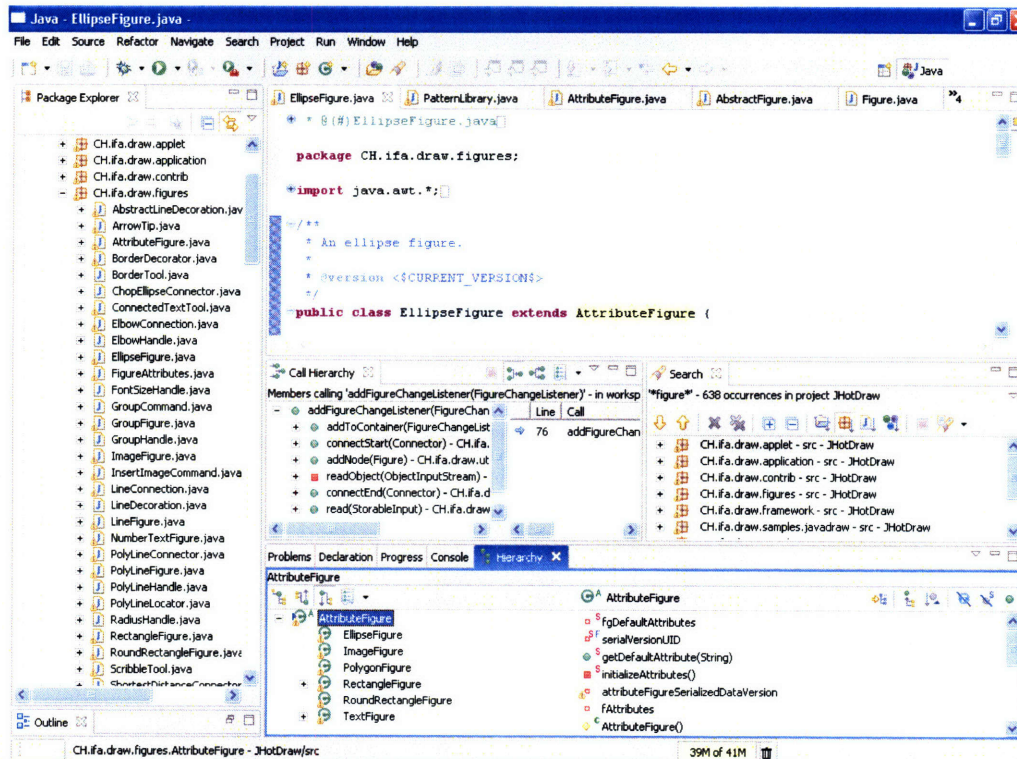


Figure 19 – Results of the exploration session shown in the previous section with the Eclipse IDE

2.3.1 Tree based approaches

The main approach for preventing the loss of context using traditional non-graphical widgets is JQuery [40]. This approach brings the multiple relationships together in a single tree. It allows developers to perform queries on nodes, in an as-needed manner, and then uses query results to populate children of the node. This approach, as shown in Figure 20, thus keeps track of the developer's exploration in the tree view. However, difficulties in using JQuery come from having the tree view's children relation represent different kinds of relationships at different levels: for some parts of the tree, the children may represent containment, but for other parts, they may represent method calls. In Figure 20 looking at the top-level makes it seem apparent that the user is looking at the contents of the `figures` package and that the `Figure` interface shown a level inside is part of this package. However, examining the codebase shows this to not be true, and that the `Figure` interface has been very intentionally been put in a `framework` package. To overcome this Relo shows the different relationships between elements in different ways using diagrammatic constraints such as visual nesting for containment, or left-to-right ordering for method calls.

2. PREVIOUS WORK

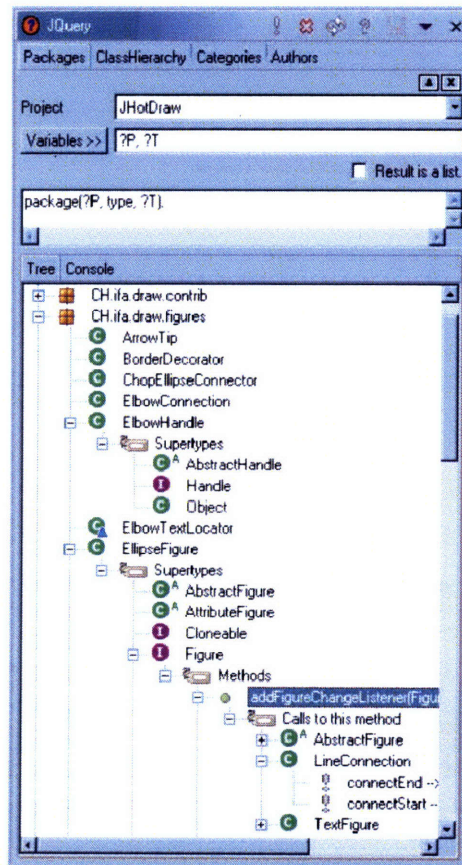


Figure 20 –Results of the exploration session shown in the previous sections with JQuery

2.3.2 Visualization based approaches

Visualization approaches have typically focused on presentation at the expense of exploration capabilities. While Reiss' FIELD [65] system (shown in Figure 21 and Figure 22) supported user-directed exploration using graphical widgets its benefits were limited as exploration would only be on a single relationship.

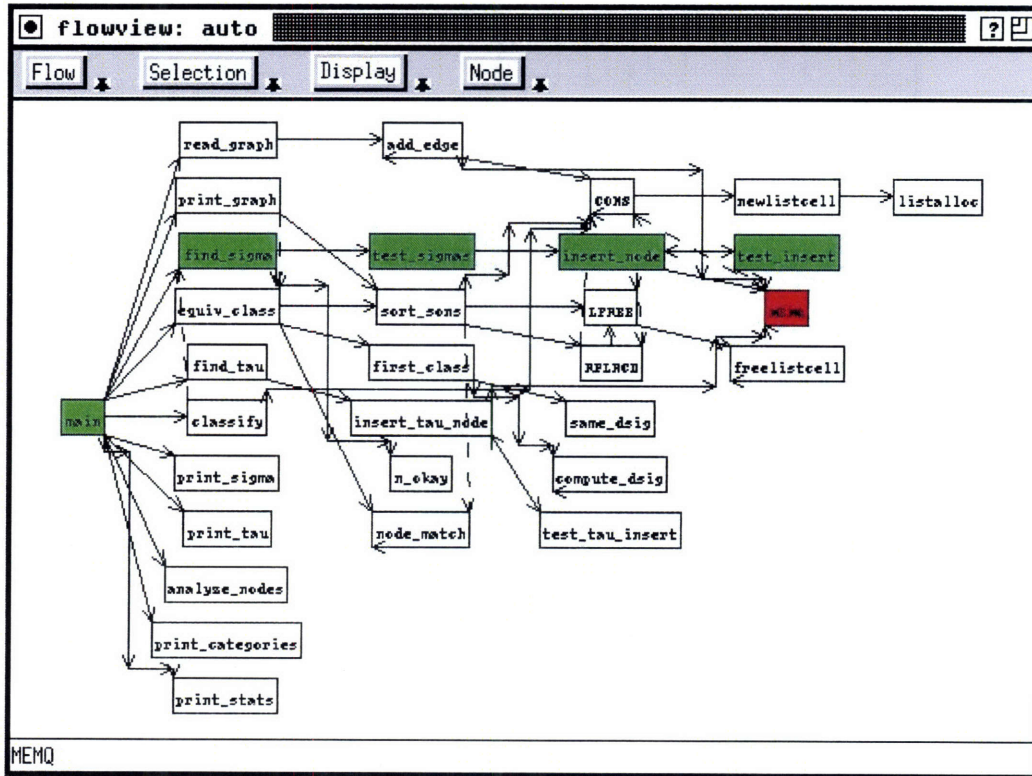


Figure 21 – The call graph viewer of Field

2. PREVIOUS WORK

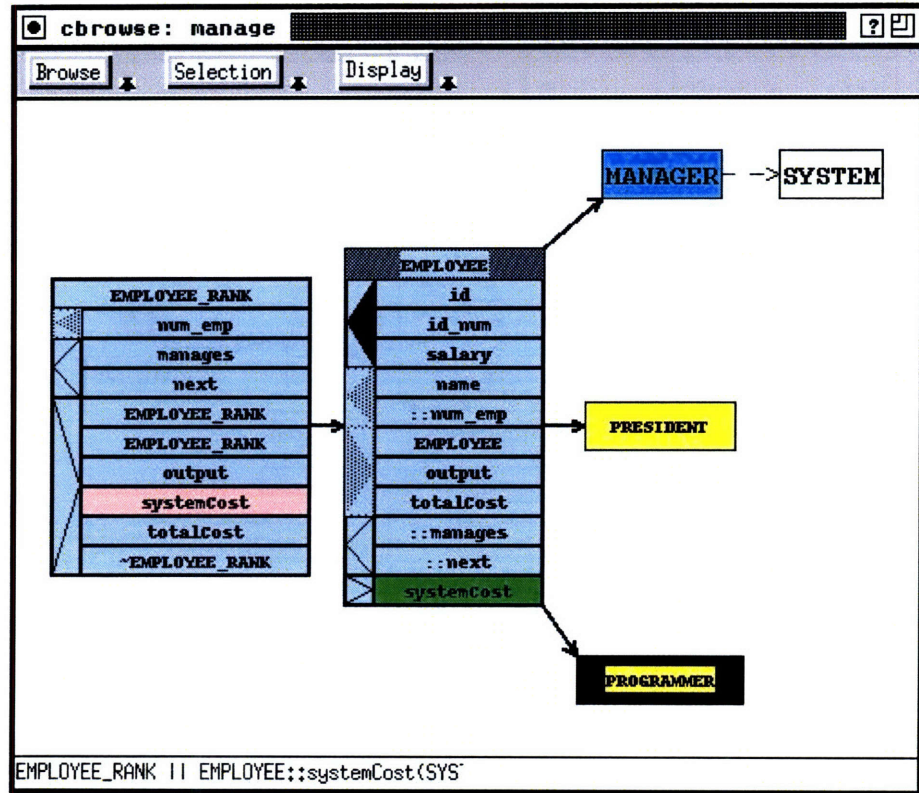


Figure 22 – Class hierarchy browser of Field

SHriMP Views [65] supports comprehension by using fisheye-lens distortion for zooming in on targeted pieces of code (see Figure 23). It allows users to select relevant portions and zoom in on information. Double-clicking on a package of interest will open-up the package, zoom-in on it, and show members of the package. However, user-directed expansion in Shrimp’s happens only along the containment axis – users cannot select a code artifact of interest and choose to expand the visualization only across a non-containment relation. Selecting a method in Shrimp and asking it to show the method’s being called, results in not only the called methods being shown but also every sibling of every method (and every sibling of every method container). This approach of showing all siblings even when interested in a particular relation tends to overwhelm users [85]. In contrast Relo starts with a single element and allows the user to direct the expansion (and contraction) of the diagram on important parts by explicitly choosing relationships. While Shrimp does provide capabilities for filtering relationships and nodes, these filters are applied centrally in a global dialog, and are not supported as part of the user’s interaction with the visualization to support his exploration. Thus, it is hard to get the system to only show inheritance for some nodes in the visualization and on method calls in other parts.

2. PREVIOUS WORK

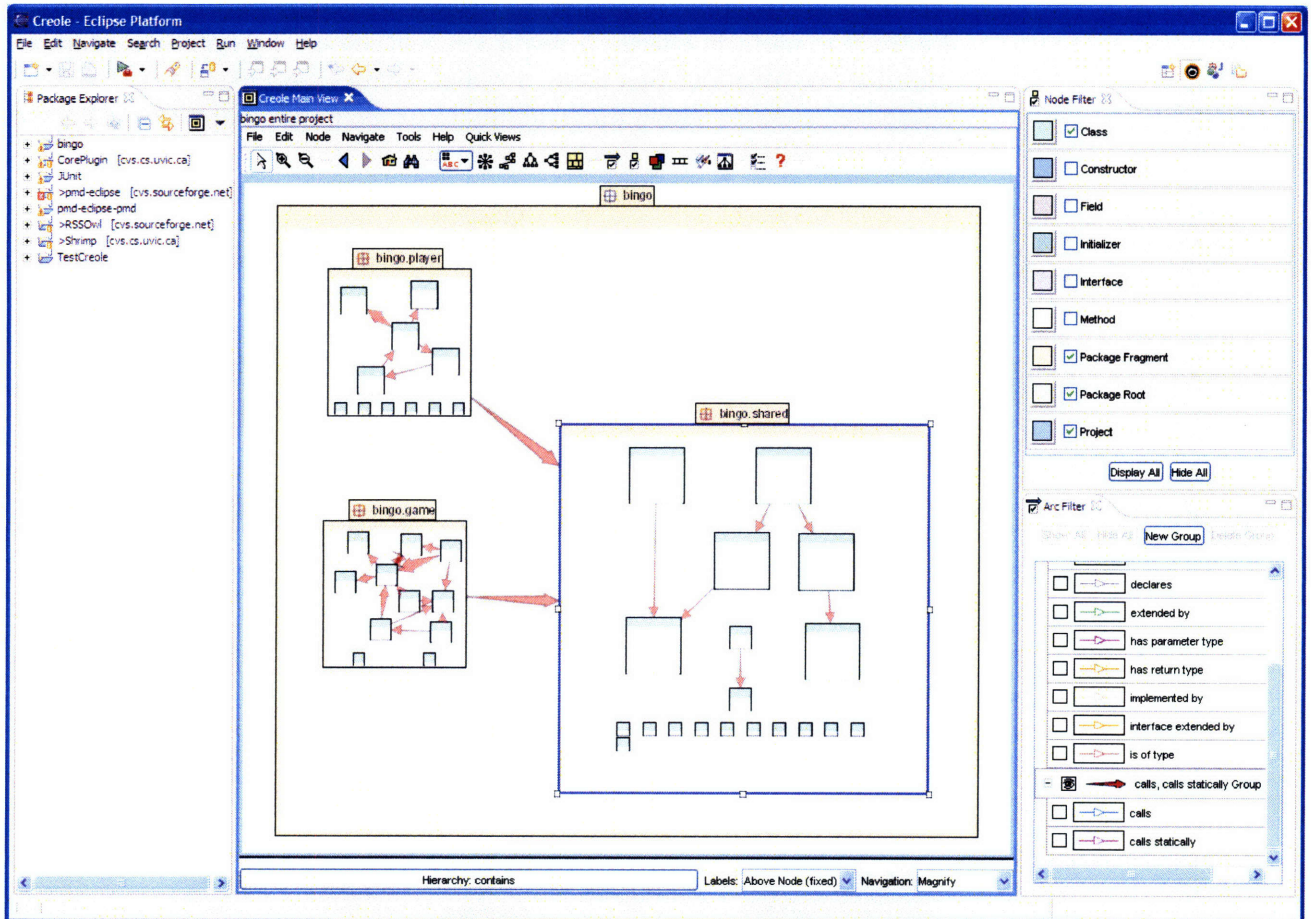


Figure 23 – Part of an exploration session with SHriMP

2. PREVIOUS WORK

Another visualization approach, the TkSee Visualizer [96] provides support for users to perform queries to build a visualization for graphical exploration, and displays relationships using a radial layout (see Figure 24). The TkSee Visualizer however limits users' ability to focus on relevant parts of the code by not allowing users to zoom in, zoom out, or remove irrelevant items from the visualization. It requires users to specify properties for queries in a dialog box outside the visualization for adding items, instead of allowing the users to leverage contextual information to browse towards the information need as shown to be needed by users [89]. Instead Relo performs these queries automatically based on the developer's selection of a code element and the developer's selection of an appropriate relationship to extend the visualization.

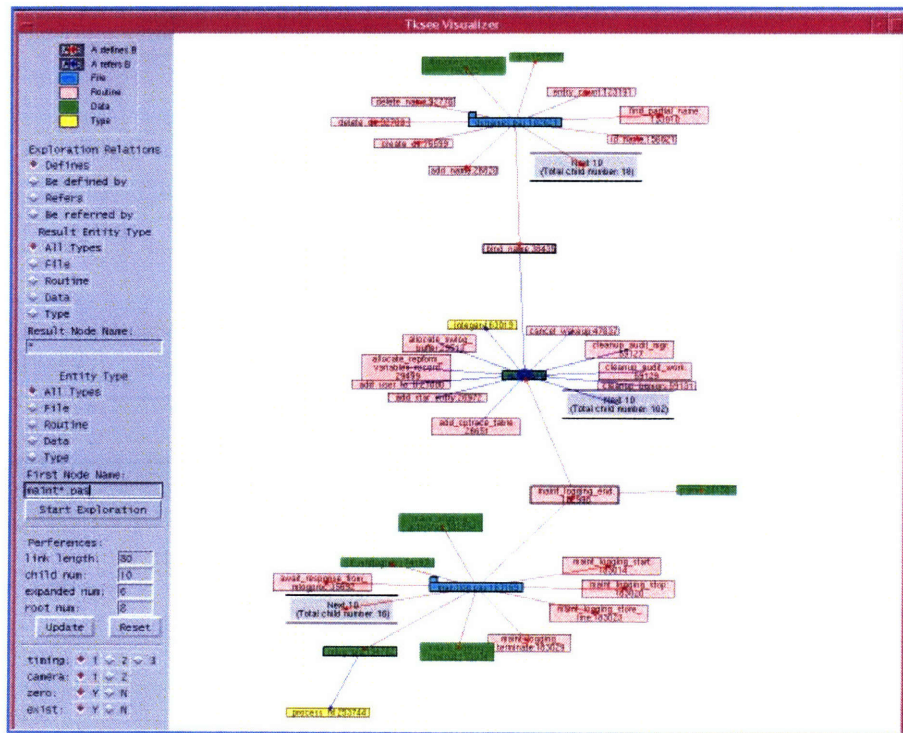


Figure 24 – Part of an exploration session with the TkSee Visualizer

Beyond improving on visualization approaches by providing exploration capabilities, Relo uses visual constraints to present nodes in expected locations. These generated visualizations are similar to sketches made by developers and have been proposed in studies examining navigation behaviors of programmers [45].

2.3.3 Discussion

Compared to other exploration tools, Relo takes a hybrid approach, leveraging the strengths of both tree based and visualization based approaches. Relo user-directedness reduces cognitive overhead, and uses a graph-based view

with automatic layout placing nodes and children in predictable locations. Further, Relo uses direct-manipulation browsing

2.4. TOOLS FOR LARGE PROJECTS

A number of research projects have attempted at building tools for working with large software projects. Like Relo and Strata these tools work by helping the developer focus on relevant parts of the code. Since many of these approaches provide benefits in ways orthogonal to Relo and Strata, they can be integrated for further helping developers work with large projects.

HIPIKAT

The Hipikat project [21] creates a full-text index of source code, the documentation, bug reports, mailing lists, newsgroup articles, and version logs. Once this index is created, developers can use the code as a starting point to find relevant documents from any of these resources. Hipikat makes it easy for developers to find documents and therefore useful information concerning the code, so that it can be examine and for the developers task. In contrast Relo and Strata works by trying to exploit the structure in the code. Hipikat results can be used with code artificats shown in Relo and Strata to provide more useful directions to users.

LASSIE

Lassie [24] is a knowledge based system that allows the capture of the varying relationships, so that developers can ask the system semantic questions. While Lassie's knowledge base has the overhead of requiring the knowledge to be acquired into the system to be useful, it does provide a higher-level understanding to users. Relo and Strata instead attempts to allow the developer to explore the relationships already available from the code, so that developers can make such semantic decisions themselves. Relo and Strata sessions can be considered as interesting starting point for building a knowledge base just-in-time for a project.

MYLAR

Mylar [44] is built as an integration into the Eclipse IDE (shown in Figure 25) and uses developers navigation events to develop a model of what the developer might be currently interested in. This model of the developers interest allows Mylar to hide items not relevant to the developer's current. Such systems, which depend on mining navigation events, like the concern leaning tools, have limitations in supporting multiple concurrent tasks. While Mylar does provide a task view, users need to remember to change update the view when switching from one task to another. Relo and Strata visualizations instead explicitly represent developer's tasks, and supporting multiple tasks by

2. PREVIOUS WORK

allowing developers to create multiple visualizations. While Mylar does help in reducing the amount of shown information to users, it has the further limitations that developers still need to remember how the shown code elements are related or connected for the current task.

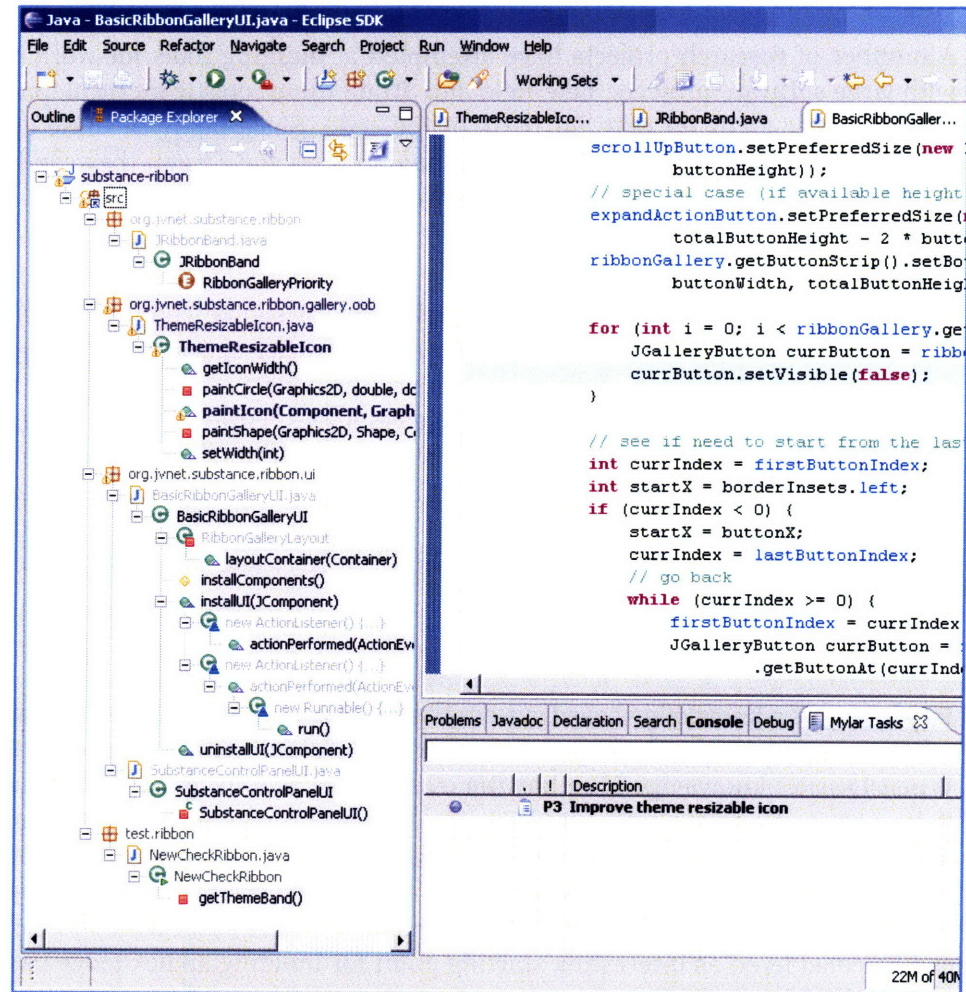


Figure 25 – Screenshot of Mylar

JAZZ

The Jazz project [39] (shown in Figure 26) provides another approach to helping developers in large projects by supporting awareness of team members activities and thereby enhancing collaboration in teams. Online team members are displayed with their pictures and status (in tooltips). Integrated chat views are anchored to lines in the code editor, and colored icons on files show changes by team members. This approach of increasing collaboration between teams by providing communications functionality directly in the IDE

is an approach orthogonal to the directions chosen by Relo and Strata, and can be provided by them.

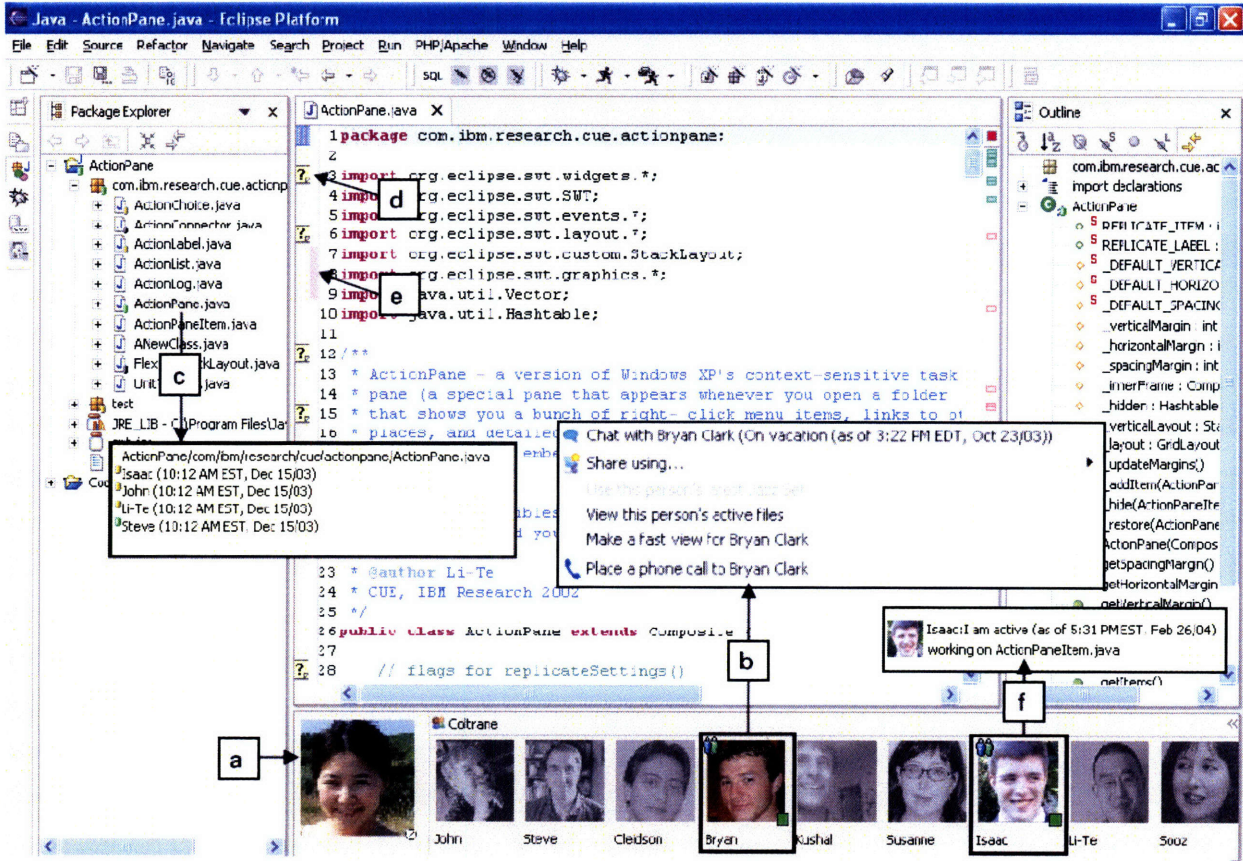


Figure 26 – Screenshot of Jazz [39]

2.5. DESIGN TOOLS

In contrast to Relo and Strata, traditional commercial design tools take a different approach for working with large projects. These tools like Rational Rose [9], TogetherJ [10], and Fujaba [4] provide reverse-engineering capabilities and provide another kind of support for program comprehension. The reverse engineering capabilities in such tools often create a single large diagram that overwhelms users. The design tools are mostly helpful for developers who already understand the code, allowing them to create diagrams as documentation and therefore providing them for subsequent developers. Relo and Strata instead focuses on providing direct support for exploration and other activities involved in the comprehension process. Models generated by such design tools can potentially be leveraged by Relo and Strata for providing users with navigation suggestions and annotations to help developers understand code more effectively.

3. SURVEY OF SOFTWARE IMMIGRANTS

Before building a tool to help developers understand code, we wanted to get their feedback on the usefulness of various understanding solutions available to them and the cases when they needed to use such help. In providing a tool helpful to most current developers we are targeting experienced developers as opposed to first time programmers. Such experienced developers who are new to a project or are new to a part of a codebase are called *software immigrants*. Software immigrants have been previously studied by Sim and Holt [75] and noticed patterns on the type and schedule of ramp-up tasks as well as the impact of environment and administrative issues. With regards to understanding code they noticed bottom-up comprehension strategies to be more effective and that the lack of documentation effected developer's ramp-up.

In order to inform the design of Relo and Strata, a survey was conducted of software developers focusing on their comprehension experiences. Previous research focused on the functionalities found and used in program comprehension tools [11], but there was a need for the usage of developers with regards to their tasks. Since using various types of documentation often assist in understanding code the survey also needed to examine which documentation techniques were used and wanted by developers when they looked at the code. It was believe that the different between what was wanted and what was used by developers would point towards capabilities that program comprehension tools should consider. This attempt is closer to the study of Robillard et al. [66], though instead of focusing on comprehension strategies, the focus

3. SURVEY OF SOFTWARE IMMIGRANTS

was to gain an insight on developer tasks when they looked at code so as to ease developers in doing such tasks in a comprehension tool.

3.1. METHOD

Given the difficulty of getting a large number of immigrants looking at one single project, we decided to study a number of open-source projects. Our goal was to get developers' opinions on their experiences understanding the open-source project's code. We therefore were looking for developers who had deeper experience than just the public API of the project; developers who had actually read at least part of the source code. Since we wanted to study developers attempting to comprehend the project, we also wanted to exclude the core developers of each project from participating in the survey, as they would have designed and written the source (not software immigrants). We did this by asking developer's to classify themselves into one of three categories (with regards to the project being surveyed): Core Developers, Examined Internal Code, or API Users. We expect that core developers to be the committers on the project working on improving the code, and API users to be the developers who use the codebase for their own projects and who likely did not look at the codebase of the project being surveyed. We expect those who marked themselves as having examined internal code to be the closest to software immigrants.

To prevent any particular project bias, we selected two of the largest organizations hosting and developing open-source projects (the Apache Software Foundation and the Eclipse Foundation), and tried to choose the projects with the largest number of developers using them. The projects were selected based on the suggestions of a developer working with each organization. The Apache projects were Ant, Struts, Geronimo, Cocoon, Xerces, Xalan, Tomcat, Derby, Lucene, and Batik. The Eclipse projects were SWT / JFace / Workbench, GEF, EMF, RCP, JDT-Core, (Text) Editor Framework, and Platform. After selecting the open source projects, we then created a short web survey (shown in Figure 27) for each project.

Links to the surveys were then posted on project-specific mailing lists. Each open source project typically has a developers' mailing list corresponding to the core developers of the project, and a users' mailing list corresponding to the clients of the project. Although we needed people ideally in between these mailing lists – developers who were advanced clients but not (yet) core developers – it was suggested not to mail on both the lists since it could possibly offend developers and be considered as spam. We therefore advertised our survey on each users' mailing list with the expectation that the more experienced developers would also be reading the users' mailing lists (to help people out). We collected survey results for a period of 1 week.

Understanding Software: [Project Name]
1. [Required] What is the depth of your experience with this Project:
<input type="checkbox"/> Core Developers
<input type="checkbox"/> Examined Internal Code
<input type="checkbox"/> API Client Only
<input type="checkbox"/> Never saw the Source Code (this survey does not apply to you)
2. How would you rate the difficulty of understanding the code?
<input type="radio"/> Can't really say
<input type="radio"/> Very Hard
<input type="radio"/> .
<input type="radio"/> Hard
<input type="radio"/> .
<input type="radio"/> Like other projects
<input type="radio"/> .
<input type="radio"/> Easy
<input type="radio"/> .
<input type="radio"/> Very Easy
3. Are there any particular things that the code did which made it easy to understand?
<input type="checkbox"/> Good Javadoc
<input type="checkbox"/> Good Examples
<input type="checkbox"/> Good Diagrams of code components
<input type="checkbox"/> Good Articles describing using code
<input type="checkbox"/> Good Articles describing code architecture
<input type="checkbox"/> Other: [_____]
4. What would be the most beneficial things that you would suggest to add to make the code easy to understand?
<input type="checkbox"/> More Javadoc
<input type="checkbox"/> More Examples
<input type="checkbox"/> More Diagrams of code components
<input type="checkbox"/> More Articles describing using code
<input type="checkbox"/> More Articles describing code architecture
<input type="checkbox"/> Other: [_____]
5. Try remembering when you had to look at the source of this project. What were you trying to do? ... And which parts did you find difficult to understand?
[_____]
[_____]
[_____]
6. Do you have any particular parts of the code (sets of classes, methods, etc.) that you think are particularly hard to understand?
[_____]
[_____]
[_____]
7. Any other comments about the understandability of the code?
[_____]
[_____]
[_____]

Figure 27 – Survey Questions

3. SURVEY OF
SOFTWARE IMMIGRANTS

3.2. RESULTS

The survey was filled 98 times, of which 20 entries were rejected since they were all filled from the same originating IP address within a span of 4 minutes. Of the remaining 78 respondents, 11 claimed to be core developers, 50 had examined the internal code, 30 had only used the publicly available API's and 2 had never seen the code⁴.

Table 1 shows the number of developers with each level of experience, and their estimated difficulty of understanding the code in their project.

**Table 1: Difficulty of understanding the project
Numbers organized by developers' varying experience with each project⁵**

Developer Experience with Project	Very Easy	Easy		Hard	Very Hard
API Client	0	5	7	8	1
Examined Code	1	11	10	13	2
Core Developer	0	5	0	3	1

To gain an insight on what currently works when trying to understand code, the survey asked developers for documentation techniques that they had found helpful for understanding the code. Table 2 shows the results. Several of the presented techniques were not directly listed in the survey, but were entered by respondents under the 'other' field in the survey. The table shows that, at least in Java projects (which were the only kind surveyed) the most effective techniques currently used are Javadoc in the code, usage examples, and articles describing the code.

Table 2: Users saying documentation technique was used effectively to assist in understanding project

Effective Documentation Technique	% saying used effectively⁶
Javadoc	44.8%

⁴ The sum is more than 78 since we allowed developers to select more than one of the options.

⁵ The survey results in this table only could 58 of the 78 entries due to a bug in the survey software used. The survey was actually conducted using a 9-point likert scale, but results in-between the values from 5-point likert scale shown above were all mapped to the same entry in the database, i.e. the survey results for the value in between 'Very Easy' and 'Easy' was indistinguishable from the results for the value in between 'Hard' and 'Very Hard'. These extra values were therefore discarded but are expected to show the same trends as above.

⁶ Percentage of 78 users. The sum is more than 100% since each user could select multiple techniques.

Effective Documentation Technique	% saying used effectively⁶
Examples	38.4%
Articles describing using code	37.2%
Articles describing code architecture	17.9%
Code structure / standards / architecture / design / pattern usage	14.1%
Diagrams of code components	3.8%
Mailing list	3.8%
Manual	1.3%
Test cases	1.3%

Next the survey asked what kinds of documentation developers *wished* they had. Table 3 shows the five most-wanted kinds of documentation. The table shows two main trends. One is that developers wanted more of all the documentation techniques that they were already finding useful, with the only exception being Javadoc – presumably because Javadoc is already used very effectively in most projects (as shown earlier in Table 2). The other main trend showed that two techniques are used significantly less than what was wanted by developer in understanding the code. These two techniques are articles describing the code architecture and diagrams of code components. Apart from these trends, respondents also asked for better code design in the projects and for improved FAQ’s (frequently asked questions) with pointers to examples.

Table 3: Comparing current and wanted techniques for all survey participants

Documentation Techniques	Current	Want
Articles describing code architecture	17.9%	44.9%
Articles describing using code	37.2%	33.3%
Diagrams of code components	3.8%	42.3%
Examples	38.5%	29.5%

3. SURVEY OF SOFTWARE IMMIGRANTS

Documentation Techniques	Current	Want
Javadoc	44.9%	24.3%

Further examining the data behind the Table 3 tells us that there are two different groups of people with different needs. The API-Clients (the developers who had not needed to look at the source code of the project) were asking mostly for articles describing code usage; in other words, they were looking for API usage articles. On the other hand, developers who had actually looked at the code more often asked for articles describing the architecture and diagrams of the code. Results for software immigrants, the group that can most benefit from a program comprehension tool, are shown in Table 4. The table shows that software immigrants want diagrams to help them understand the code, but are currently not getting them.

Table 4: For software immigrants (developers examining internal code)

Documentation Techniques	Current	Want
Articles describing code architecture	18.0%	46.0%
Articles describing using code	40.0%	26.0%
Diagrams of code components	0.0%	42.0%
Examples	44.0%	26.0%
Javadoc	44.0%	28.0%

3.3. TASK ANALYSIS

In order to gain an understanding of the functionalities needed to best support understanding code, the survey also asked developers to describe a task they were doing when they last looked at the source. While we would ideally like the tasks to be typical tasks, since developers were drawing on their memory probably some time after the event, these tasks are likely to represent the *memorable* ones, tasks that caused the developer pain and might benefit from tool support. From the free-form answers, we derived the taxonomy shown in Table 5, along with how many developers indicated each task. Based on answers, we split the table into two columns: one for the tasks that required a static analysis of the code, and the other for those requiring a run-time analysis. Items in the 'other' column refer to tasks that were too vague to determine whether they needed static or dynamic results.

From the table, it seems that support for 20 of the given 30 tasks (over 66%) can be provided by helping users explore static relationships in the code.

Table 5: Tasks done by user when they were examining the code

Task	Static	Dynamic	Other
1. Extend code	4		
a. Decide – using provided mechanism or modifying code	2		
b. Selecting appropriate class (to extend)	1		
2. Fix bugs	2	3	3
a. Find cause of compile-time/run-time errors	2	2	
b. Find cause of memory leak		1	
c. Find workarounds			1
3. Understand “architecture”	3	4	5
a. Understand lifecycle of class		2	
b. Understand threading		2	
c. Understand concepts	1		
d. Understand design intent	2		
4. Understand code	11	3	6
a. Use API efficiently	1		
b. Understand the “core” of project	2		
c. Understand run-time trace (when not reflected in code structure)		3	
d. Find & understand code for a task which is an example of what you want	3		
e. Understand relationships between multiple classes	7		
f. Understand how some functionality is implemented	4		
g. Understand generated class	1		
h. Understand roles of variables with 2-3 character names	1		
Total	20	10	14

4. STRATA USER INTERFACE

The Strata interface provides for understanding code using traditional layered architectural diagrams as a basis for an exploration interface. Layered diagrams are mostly useful in gaining a high-level overview of a project or part of a project. Such diagrams can help a developer understand the code and target parts of the codebase to add, modify, and fix functionality.

The parts of the project under consideration by the developer are referred to here as *modules* or *components*. The codebase is often further organized by having modules consist of multiple sub-modules. These modules and sub-modules form a basic part of a layered architecture. Ideally, these modules have sharp, obvious boundaries, with the dependencies organizing the modules into layers such that each layer depends only on the layers below it [72][90]. With such layered dependencies, functionality can be clearly and easily be tracked to the underlying modules. Furthermore, such layered dependencies increase the predictability of changes in the code and can limit the impact of the changes. Thus, managing a potentially large number of modules and reducing dependency cycles, often results in the modules being organized as a layered architecture.

Given that large codebases need to be organized using a layered architecture, Strata is designed to easily build such layered architecture diagrams. Efficiently using the code to create such architectural diagrams is expected to ease understanding of the system and promote project architecture reviews [32]. However, having the architecture available and synchronized with the application implementation has been difficult and costly [47]. Without active intention to maintain the architecture, code boundaries decay. This decay can happen because of developer error, rapidly approaching code delivery deadlines, or the presence of crosscutting concerns. An understanding of

the current state of the codebase as different from the intended ideal version can also provide an opportunity to improve module implementations.

Previous approaches to build these diagrams efficiently and automatically have had limited success partially due to the difficulty in dealing with the decayed boundaries – it is hard to build architectural diagrams from spaghetti code. An approach is therefore needed to generate layered architectural diagrams even with such decayed boundaries. Since it might not be possible to detect and know the important dependencies among spaghetti code, the approach will need to make good guesses in generating the architectural diagrams.

While good guesses can help easily create such layered architectural diagrams, these guesses might not be always right. In such cases, it is important to provide lightweight interaction to modify the generated diagram. Having a familiar interface with lightweight interactions to correct the diagrams allows users to not need training in the tool. This lack of needed training is important for supporting developers who might be trying to achieve some other primary task when understanding an underlying codebase. In contrast, existing tools like Armin [13] and LDM [71] require expertise in using the tool to obtain a valid architectural diagram.

Below we describe the requirements for building a familiar interface as needed for Strata. We then describe the approach to analyzing the code to create the diagram, along with the guesses made in such a process, and most importantly the approach used to interact with such a diagram to correct guesses made by the system. We then describe support for exploring with the Strata interface and show support for annotations and creating new modules for use within Strata.

4.1. APPEARANCE

Layered architectural diagrams examples can be seen in Figure 28 and Figure 29. The main features of such diagrams are used in shaping Strata's appearance: The architectural diagrams consist of rectangular modules which are shown in layers, as opposed to radial layouts such as those shown by some of today's tools (for example in Figure 15). Members of a module are laid out in a single row or column, and if there are too many items they are shown using a grid like layout. Modules that build on other modules are shown in a layer above the modules that they depend on. As can be seen in Figure 30, Strata uses these points to provide users a diagram consistent with other diagrams they have seen before.

4. STRATA USER INTERFACE

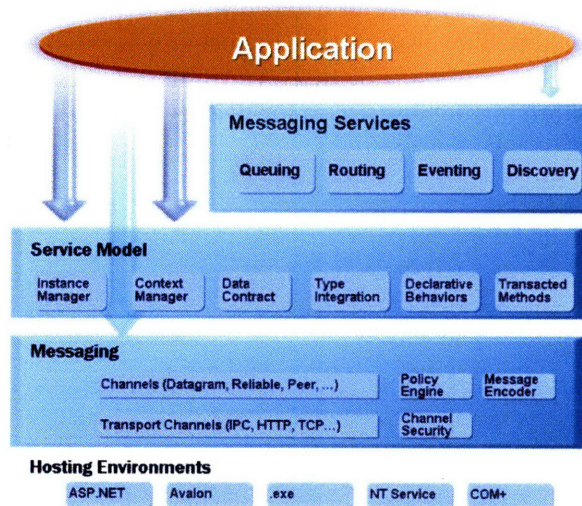
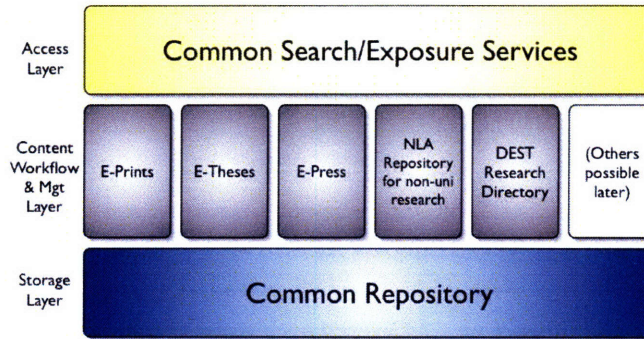
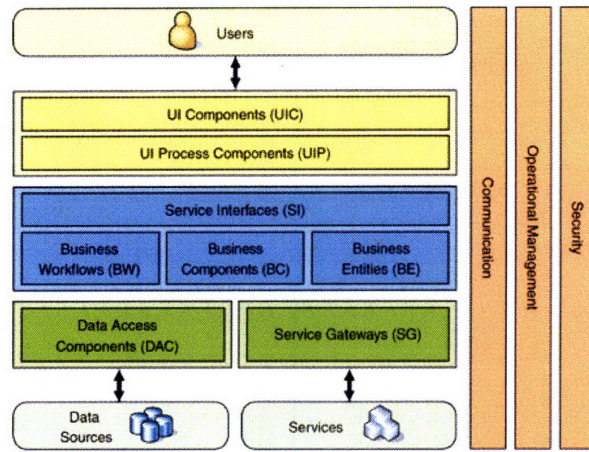


Figure 28 – Typical layered architectures diagrams (as in Figure 2)

4. STRATA USER INTERFACE

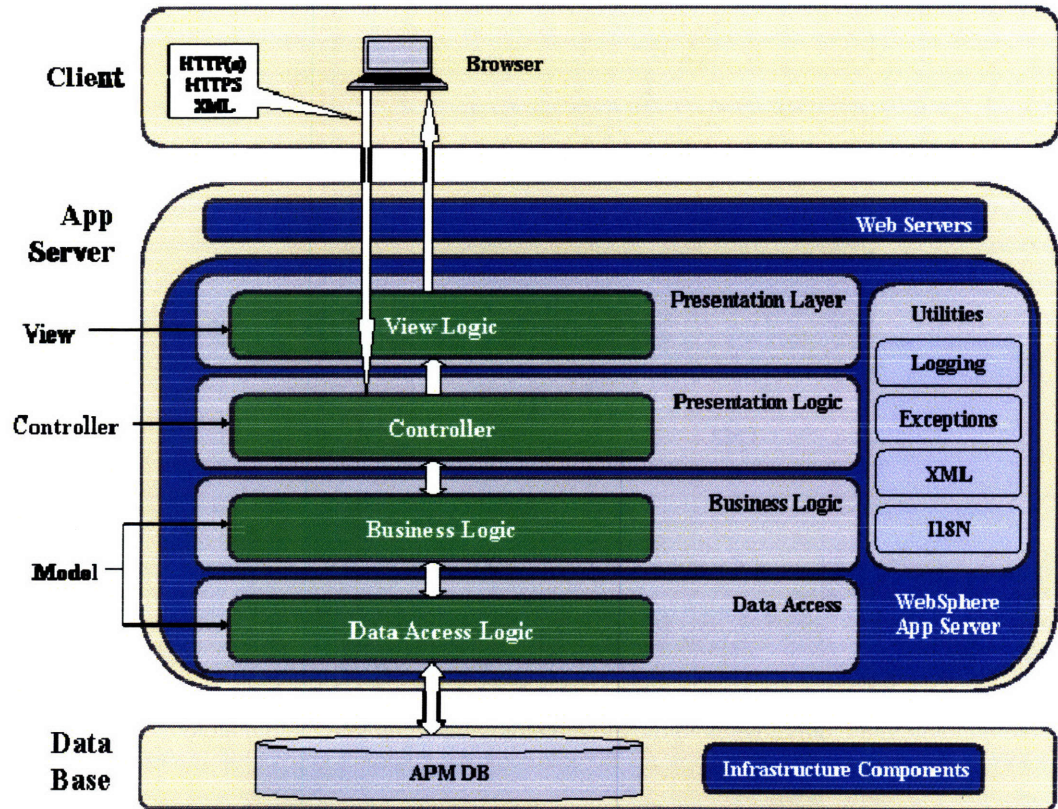


Figure 29 – A Layered Architectural Diagram⁷

One feature of existing layered diagrams that Strata does not support is that the shown arrows include both static and runtime dependencies. The runtime dependencies are not shown primarily because it is hard to extract them in an efficient manner. Once extracted, these runtime dependencies can be easily added to the underlying dependencies for displaying a more accurate diagram to the user.

⁷ Source: A typical architectural diagram used from one of Accenture's Project Documentation

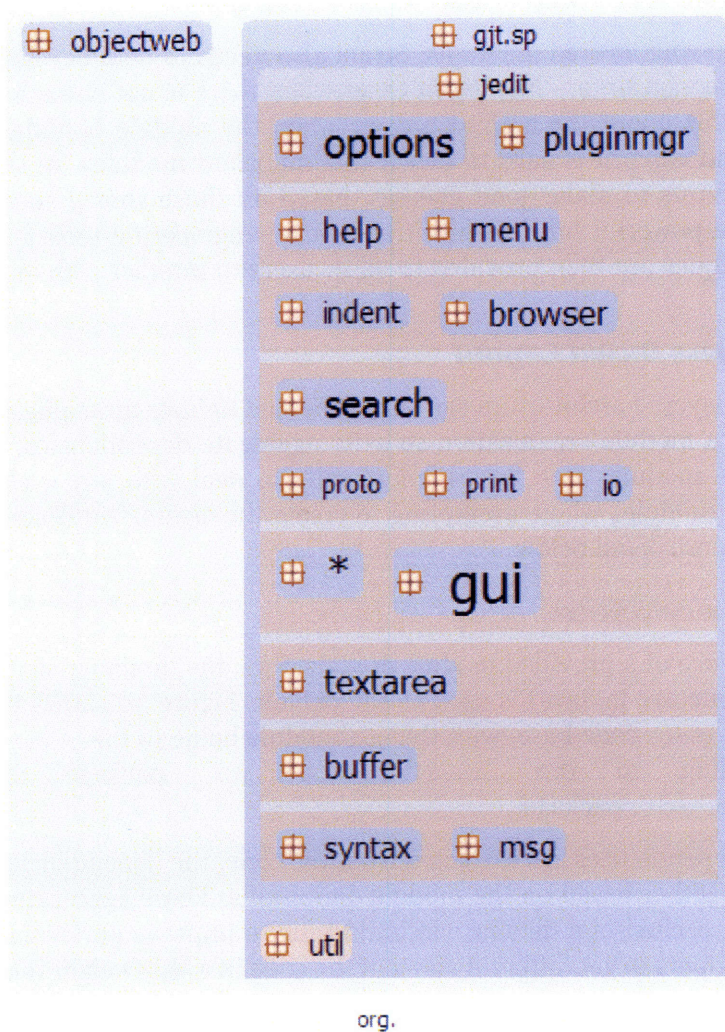


Figure 30 – Strata display of the jEdit project
(same as Figure 5)

In a given module organization, it might be the case that some modules are much more important and larger than others. Using the directory structure, the root of the codebase having some branches with very little code while other branches have most of the code. For example, in Figure 30 most of the jEdit code base is under the `org/gjt/sp/jedit` branch with the remaining (`objectweb`, `installer`, etc.) being supporting code. To support a developer focusing in on the largest parts of the code, Strata shows such modules larger than the rest of the modules. The module with the largest size is shown with twice the font size and twice the font weight (boldness) as the module with the smallest size, with the rest of the modules scaled proportionately. We measure the size to a module by the number of classes contained by it.

4.2. SUPPORTING INTERACTIVE LAYOUT

Beyond showing layered diagrams, Strata also needs to help developers with real world constraints – helping in cases when there is not enough information available about the architecture or when the module boundaries have decayed and when it is hard to detect the important modules. In such cases Strata attempts to make good guesses, based on those shown successful in other more powerful but less-intuitive reverse-engineering tools [71]. Interface techniques are then provided to allow users to interact with and correct guesses.

4.2.1 Guess Based Layout

Building a layered architecture can simply be said to have three phases: firstly for having a module organization so as to aggregate dependencies, secondly to split the modules into layers, and finally to deal with any cycles found among the modules when attempting to create the layers. We describe these phases in more detail below.

MODULE ORGANIZATION

In the absence of a provided module organization, the directory structure (or package structure in Java) is used as the default. Figure 30 shows the Strata display for `jEdit` code base, with the `gui` module being in the `jedit` module, corresponding to the `org.gjt.sp.jedit.gui` package and the `org.gjt.sp.jedit` package.

After the dependencies are extracted from the code, the dependencies are aggregated together based on the module definitions. These aggregated dependencies are cached in a database at build time to improve performance, and allow developers to get different views of the code in a lightweight manner.

BASIC LAYOUT - PARTITIONING

Once dependencies are aggregated Strata tries to perform a layout. For the layout, a basic partitioning algorithm is followed in splitting modules into layers so that the modules in higher layers have dependencies on the modules in lower layers [15][30][97]. These algorithms work on a set of given modules, iteratively processing and removing modules from the set. The modules that don't have any modules depending on them are processed by adding to the top layer, while the modules that don't depend on other modules are processed by adding to the bottom layer.

In some cases, there are too many modules with no dependencies on each other. This can make a layer too wide. In such cases, if a layer is wider than any other layer in the module it is split into multiple layers, such that its width is smaller than the next largest layer.

DEALING WITH CYCLES

The largest challenge in building the layers is in dealing with the presence of unexpected and unplanned dependencies between modules that often result in cyclic dependencies. Not dealing with such dependencies results in most modules showing up in a single layer, producing a useless diagram. For example, consider the difference between Figure 30 generated by Strata and Figure 31 generated by LDM [71]. Such cycles require Strata to have an appropriate strategy for dealing with cycles. We use a simple approach in Strata for dealing with the cycles. While an optimal solution will be helpful, it is more important to have a strategy in dealing with the cycles.

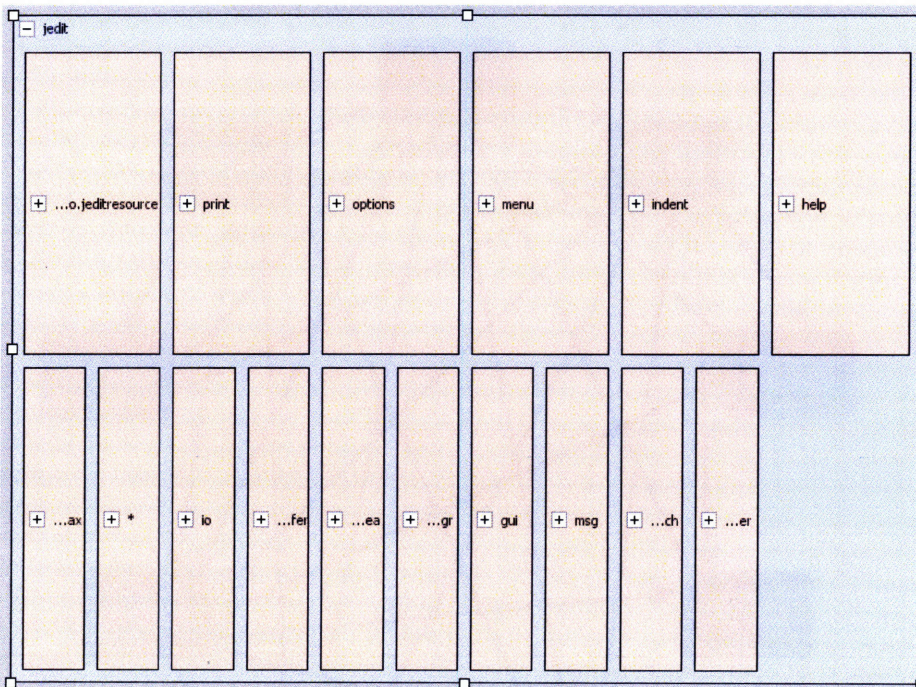


Figure 31 – Automatically building a layered diagram without dealing with cycles (for the jEdit project).

Modules in the top row have no dependencies on each other, while all the modules in a second row are part of an interconnected cycle (output as provided by LDM [71]).

In order to provide a good layered diagram in the presence of cycles, Strata makes the assumption that the cycle-inducing dependencies are unintentional and therefore fewer in number than the intentional dependencies. Strata thus needs to find and remove the smallest number of dependencies to make the dependencies graph acyclic. While this problem, referred to as the minimum feedback arc set⁸ problem, is known to be NP-complete [34] Strata uses a simple algorithm to approximate it.

⁸ In the phrase ‘minimum feedback arc set’, arc refers to the edges of the graph, feedback arcs refer to those edges inducing cycles, and minimum refers to the smallest such set.

4. STRATA USER INTERFACE

Strata uses the number of dependencies between two modules as a proxy for the importance of the dependency between the modules. For dependencies, we use inheritance, field references, method invocations, and construction of objects. The algorithm works iteratively on each cycle, in a 3-step process until all the cycles have been removed:

1. Find the *filter* strength: the filter is the smallest number of dependencies between any two modules in the cycle.
2. Break dependencies below the filter strength: this works to effectively guess the unplanned dependencies and break them.
3. Partition the filtered set: with the weakest dependency removed the partitioning algorithm can eliminate some cycles.

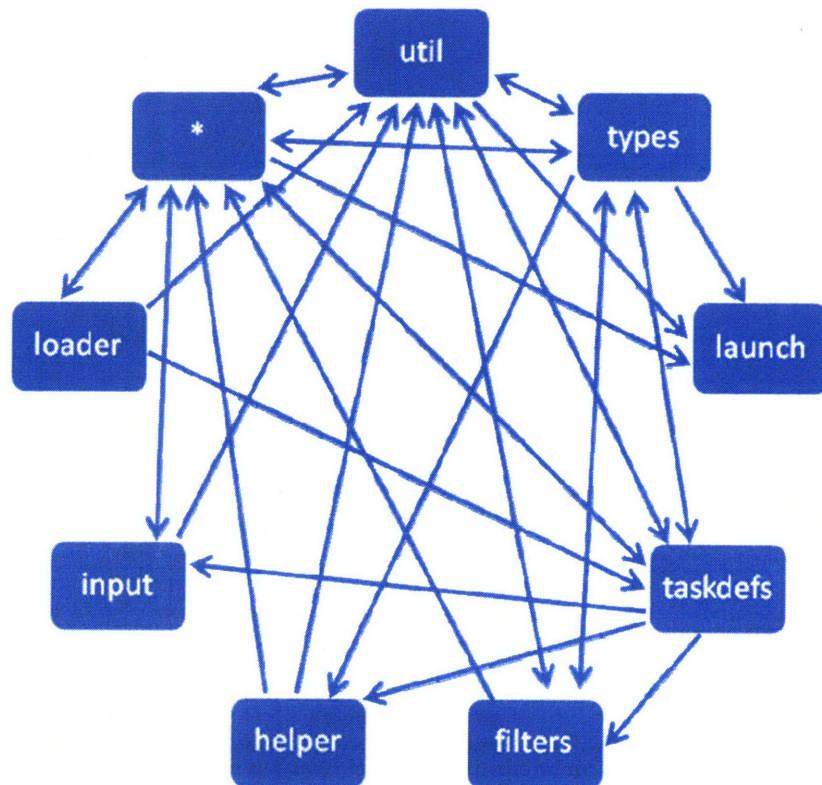


Figure 32 – Dependencies in the top-level modules of the ant project.

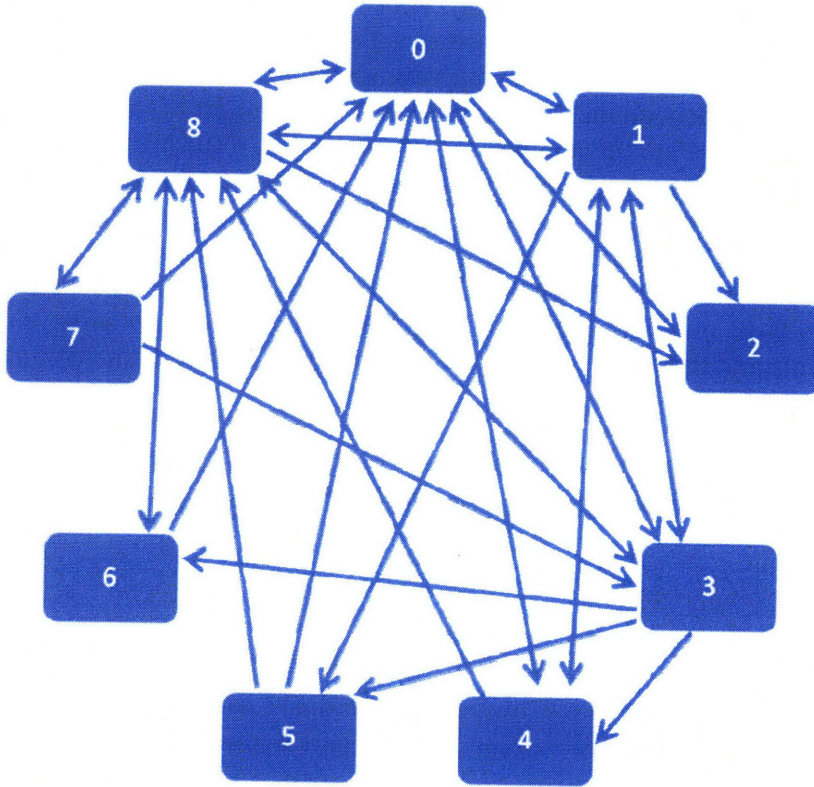


Figure 33 – Dependencies in the top-level modules of the ant project (numbering modules).

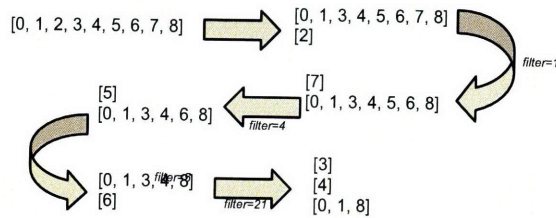


Figure 34 – A depiction of the steps of the layering algorithm on the ant project.

We describe the cycle breaking approach in the context of creating layers for the children modules of the `org.apache.tools.ant` module of the Ant project [8]. Showing the modules and the dependencies we get Figure 32. To simplify the description we represent each of the nine modules using a number in the 0..8 range (Figure 33). Attempting to partition the nine modules with the basic algorithm results in two layers: with the first one consisting of a cycle of eight modules and the second one consisting of the module '2'. This partitioning, shown in Figure 34, can be said to be a partitioning with a filter of 0. The algorithm now looks at the remaining cycle applying a stronger filter

4. STRATA USER INTERFACE

(and ignoring more dependencies). Thus in the above figure, the modules {0, 1, 2, 3, 4, 5, 6, 7, 8} get partitioned to get the cycle {0, 1, 3, 4, 5, 6, 7, 8} (with module '2' removed) without applying any filters. In the next step we find that filtering dependencies of strength 1 will break part of the cycle, and after partitioning we get the module '7' being removed, resulting in the cycle [0, 1, 3, 4, 5, 6, 8]. We continue the process of ignoring cycles by filtering dependencies of strength 4 or less in the next step, and going on until we have very small cycles. In Strata we stop breaking cycles that have 3 items in them. The algorithm chooses not to break all the cycles since it would otherwise result in a very large number of layers – with the ideal diagram not being too tall or too wide and therefore containing an equal number of rows and columns.

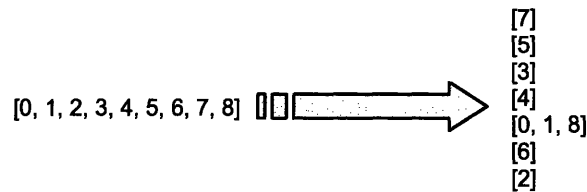


Figure 35 – Results of the layering algorithm.

Once we have finished partitioning the modules, we can combine the results of the partitioning in the order that the partitioning was done as shown in Figure 35.

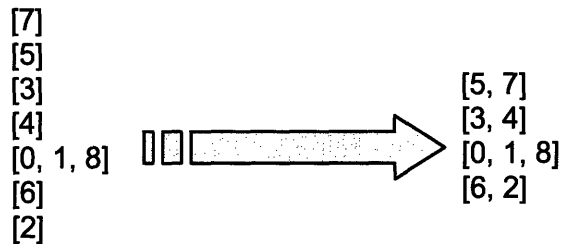


Figure 36 – Merging the results of the generated layers.

Breaking the cycles and applying filters can mean that modules in two adjacent layers have no dependencies on one another. For example, in Figure 36 module '7' does not depend on module '5' and therefore the two modules can be merged into a single layer. We thus move modules generated from the previous step to eliminate false layerings, i.e. cases when the modules do not depend on the layer below them. Once the layers are created they are merged by moving modules downwards till they depend on a module directly below them.

4.2.2 Interacting with Layout

With the guesses provided by Strata, there is a need to allow users to interact with it and correct these defaults. Modules can be easily moved to more appropriate layers, can be moved into other modules, or can be removed from the diagram.

In generating a layered diagram Strata uses a number of heuristics to make guesses and organize the code. These guesses can be incorrect and the interface needs to provide for a lightweight manner to correct them. In particular, a module guessed to be on a wrong layer, can be easily corrected by just moving it. Another type of guess made by Strata is in using the directory structure for defining modules boundaries. Strata uses the concept of *breaking* a module, to layout and display the sub-modules at a higher level. Users in Strata can open up modules to show the layers inside a module. Such opened modules have all the sub-modules drawn within the parent module. A user wishing to see the layers with the sub-modules laid out without the parent containment can ask the parent module to be broken.

Consider part of the `jEdit` project expanded in Figure 37. Expanding the `browser` module shows the classes that it contains and the layers that they form. A user might decide that the `browser` module as defined by the directory structure is incorrect and therefore would want to organize it at a higher level. Breaking the `browser` module results in Figure 38.

4.2.3 An Active Layout Engine

The Strata layout is *active* and is applied by default whenever new items are added to a diagram, such as by a developer dragging-and-dropping a module on the diagram. When a module is removed from the diagram, the diagram is *crunched* – modules are moved to new layers that might exist as if the module had not been shown in the first place. Crunching is performed since the user might have felt that the removed module was not relevant to the diagram. However, when a user explicitly moves a module to a new layer, these automatic layout rules are disabled. The disabling of the automatic layout happens because the moving of the module to a new layer is likely an indication by the user that the layers involving the moved module were incorrect.

4. STRATA USER INTERFACE



Figure 37 – Expanding the browser module in jEdit.

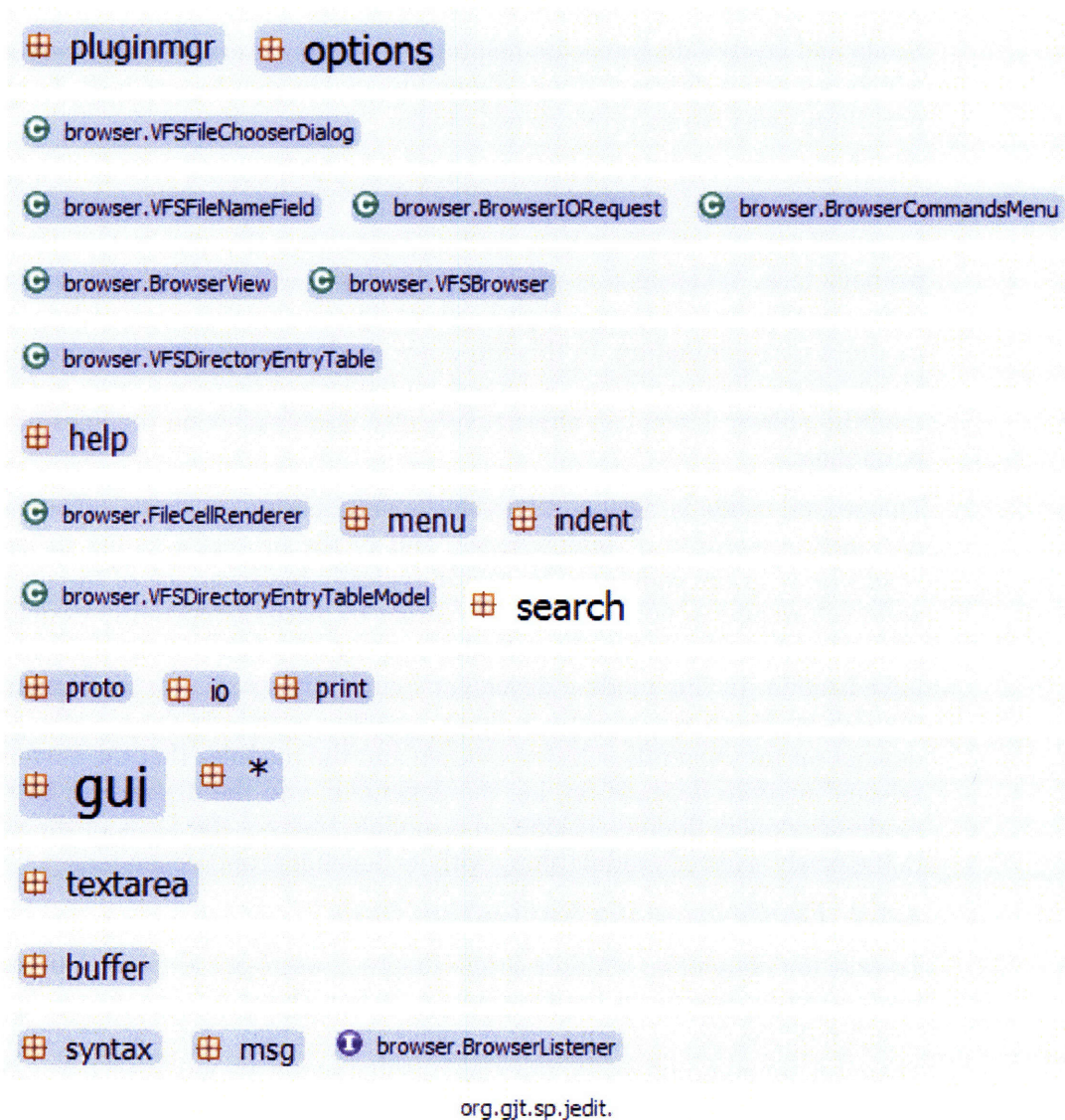


Figure 38 – Breaking a module in Strata.

4.3. SUPPORTING INTERACTIVE EXPLORATION

Strata provides a number of ways to help users navigate around using a layered view. These include behavior depending on Strata’s use as a primary or secondary mechanism, support for exploration with buttons called navigation buds, and additional query support built to support overview related tasks.

4.3.1 Interface Behavior for Navigating

Strata and layered diagrams are helpful in providing users with an ‘overview’, which can mean different things depending on the developers’ task. To support these tasks, Strata provides two modes.

The first mode is for a developer intending to understand a project or a sub-component. This typically happens when the developer is introduced to a large codebase for the first time, and understanding the code is typically his primary task. When in this mode the developer will want to dive into the code, starting from getting a high-level overview to selecting interesting sub-modules for examination. In this top-down mode Strata supports exploration by automatically expanding the largest module when there are three or fewer modules shown. Users can alternatively also double-click on modules to expand them.

The second mode is designed for developers focusing on doing other tasks like feature addition or maintenance, and in the process wanting to know where he is in the codebase. He would typically have an idea of the overall architecture and would be looking to Strata to place the currently examined piece of code in the architecture. Strata supports this behavior by providing a linked mode. In this mode a developer’s currently selected code element in the IDE is shown as a module in Strata together with both the modules that depend on the currently selected module and the modules that the currently selected module depends on, i.e. the dependers and the dependees. In this mode developers double-clicking on a module changes the focus of the view to the newly selected module along with its dependers and dependees.

4.3.2 Exploration via Navigation Buds

To ease understanding within a Strata diagram, users are allowed to interactively explore directly in the diagram. A button shown on the currently selected modules (as in Figure 39) provides the primary exploration support. These are for common exploratory actions like removing modules, expanding them to show their children, and showing which modules depend on the current module. Navigation buds help developers to realize the available options and provide a means to easily explore the code.

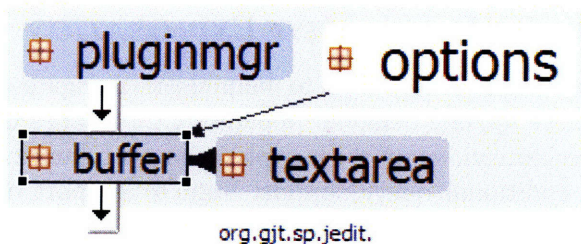


Figure 39 – Navigation Buds on the `buffer` module in Strata.

When users select a module (`buffer` in this case) and click on the ‘show depending’ navigation bud, Strata shows the modules that depend on `buffer`. As shown in the figure, while the `options` module does depend on `buffer`, other modules depend on it as well.

Strata also allows users to explore into the dependencies shown in a layered diagram. This allows a user to understand the cause of the dependencies. A developer can right click on a dependency and ask to Show Dependency Cause, will have the source and destination modules opened up showing only the modules causing the dependency.

4.4. UPDATING MODULE DEFINITIONS

Strata uses packages and directory structure as the modules hierarchy. While the package structure often represents a useful organization of the code, there can be multiple organization schemes that might be relevant. In some software projects that were examined for this thesis, the package structure was organized according to technology domain concepts, but an alternative organization might be related to business domain based concepts. Developers can explore using Strata, focus in on a set of code, and then save that view. These saved module definitions can be shared with other developers or later used in other explorations.

5. RELO USER INTERFACE

Developers trying to understand code and examining details need to see the different relationships involved among the target code element. Relo is therefore based on UML class diagrams. UML class diagrams are the most popularly used of the different UML diagrams [28], and are therefore expected to be familiar to most developers.

Beyond using a familiar notation, like Strata, Relo follows the core principles outlined in the thesis. Relo focuses on preventing users from getting overwhelmed by the amount of shown information – a common occurrence when examining details. Relo therefore takes the approach of showing the bare minimum information in partial focused diagrams – any items that are not explicitly indicated to be relevant are not shown to the developer. Further, in helping developers examine the code and explore around, Relo also provides users with lightweight means of interactively exploring around the codebase.

Below we describe the appearance of Relo visualizations in our attempt at making them familiar to users. We then discuss the requirements on Relo to maintain a familiar representation while allowing users to interact with the diagram, by providing support for an interactive layout engine. Next we discuss the interactive exploration capabilities. We then describe support for tracking the explorations in the background and providing automated diagram management support in Relo. Finally, we describe support for using Relo for communications.

5.1. APPEARANCE

Software visualization tools have traditionally shown code in basic box-and-arrow diagrams (for example in Figure 40) and require developers to follow

5. RELO USER INTERFACE

external legends to understand the main components shown in the diagram. In contrast, Relo is designed to appear familiar and therefore appears similar to the popular UML class diagrams (Figure 41). Packages, classes, methods, and fields are easily distinguishable using expected representations together with familiar icons from the IDE. A developer looking at a Relo diagram does not need to refer to a legend to differentiate a method from a class, or a class from a package. Further, as in UML class diagrams, class members are shown vertically using a stacked layout with no space between members, while the various classes are laid out in a less constrained layout. While such a representation causes challenges in understanding relationships among class members, the representation has strengths in showing relationships among multiple classes. Relo relationships are also similar to typical UML class diagrams with inheritance relationships not being drawn as single straight lines from the start to the end, but as stepped lines with upward pointing closed arrows.

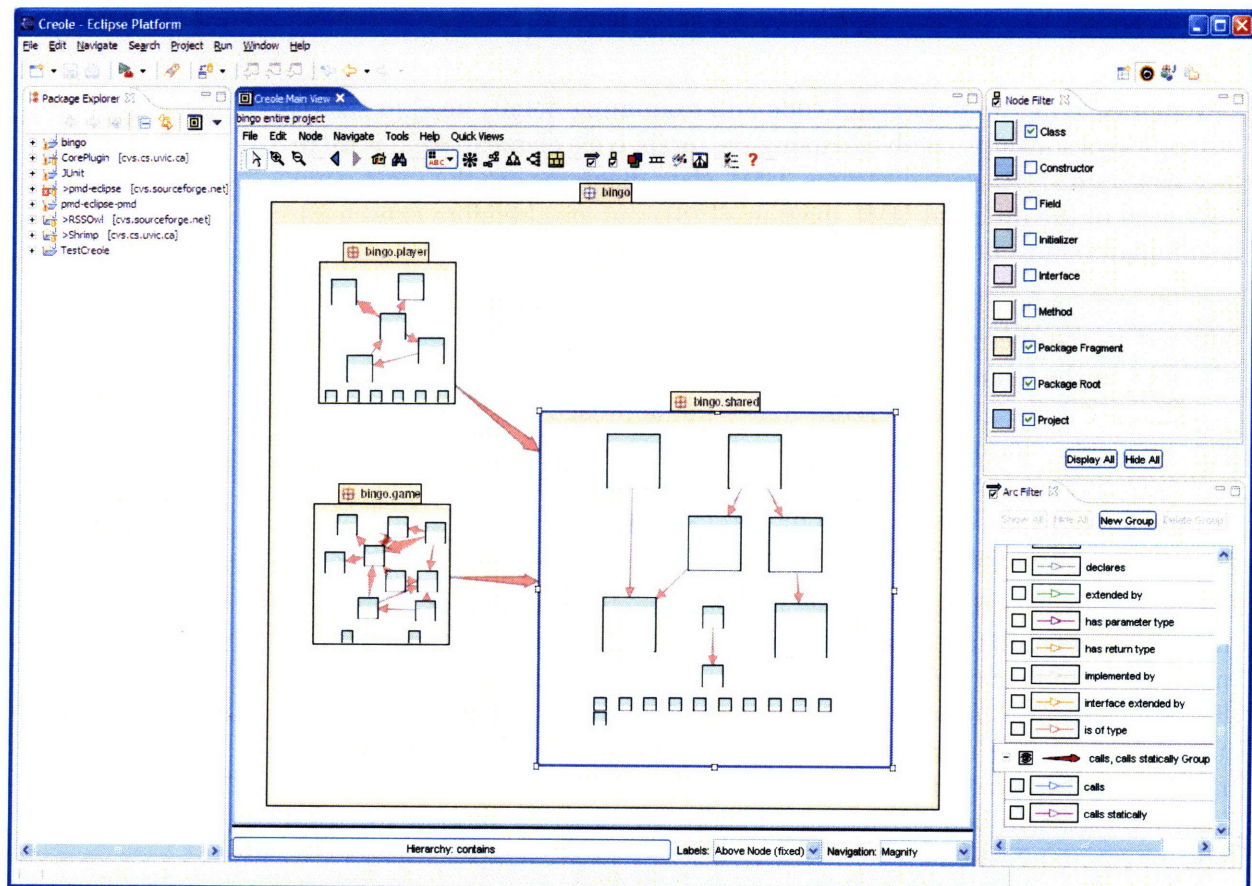
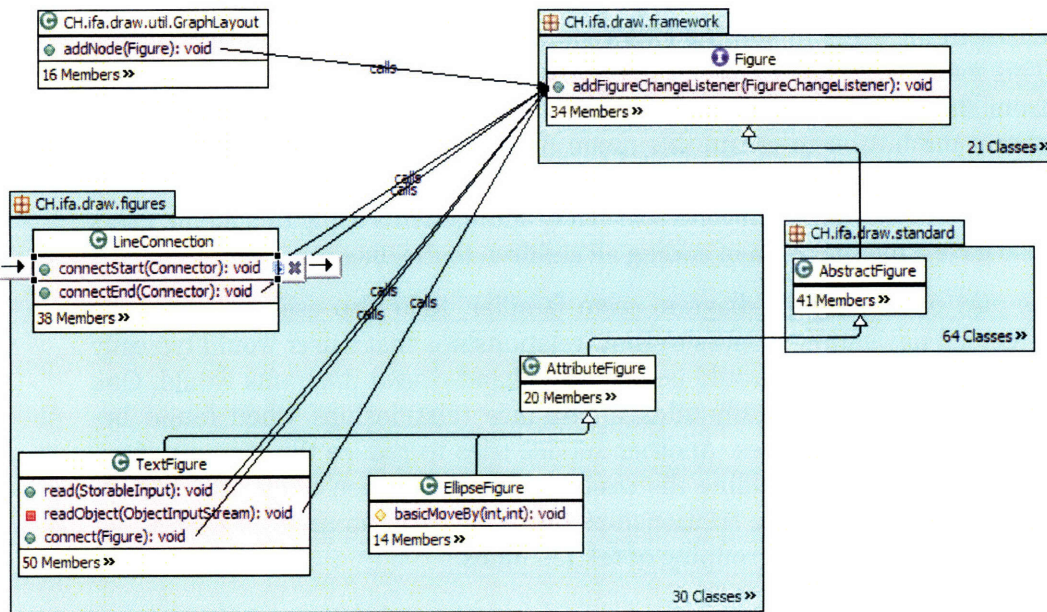
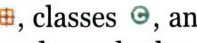







Figure 40 – Part of an exploration session with SHriMP (as in Figure 23)



**Figure 41 – Relo showing part of the JHotDraw project.
(as in Figure 9)**

Figure 41 shows part of the JHotDraw project. The figure shows the `CH.ifa.draw.framework`, `CH.ifa.draw.figures`, and the `CH.ifa.draw.standard` packages. The `LineConnection`, `TextFigure`, `EllipseFigure`, `AttributeFigure`, `AbstractFigure`, and `GraphLayout` classes and the `Figure` interface are shown. To distinguish between the different items, the standard Eclipse icons for packages , classes , and interfaces  are used, with methods using different icons based on whether they are public , protected , or private .

Relo uses partial diagrams to prevent users from getting overwhelmed. It therefore shows as little information as possible. However, users also want visible code elements to be easily identifiable. This is possible when showing an element with its container element, for example in Figure 41 when showing the `TextFigure` class inside the `CH.ifa.draw.figures` package. However, when an element is requested to be added to the diagram, Relo tries to show as little information as possible, and therefore does not automatically include these container elements. Such shown code elements therefore have their name shown fully qualified – a Java method appearing by itself is prefixed by its containing package and class. In Figure 41, the `GraphLayout` class is therefore shown as `CH.ifa.draw.util.GraphLayout`.

The approach of showing nodes only when explicitly requested often results in cases when not all of the children of a node are shown. In such cases, there

is a need to indicate to users that more children exist than the currently shown nodes. To show the existence of more children and to provide a mechanism for users to add the children, users are shown a *more items* menu. This menu consists of a list of the remaining children and selection of items from the menu results in the nodes being added to the diagram. Figure 41 shows the button to create this menu on classes to allow users to add methods and fields, and shows the button on packages to allow users to add classes and interfaces. Beyond selectively adding items using this menu, users also have a menu option of adding all children to the diagram.

As part of making the diagram seem familiar, Relo also uses automatically triggered navigation services to show relationships that a user would typically consider obvious and would be shown in hand-drawn diagrams would. One such service automatically adds inheritance relationships when found between any visible classes. Another service tries to reduce the amount of information shown by adding the container element when there are multiple children shown. Adding relationships make the diagram seem to have lesser information via visual grouping of related items.

5.2. SUPPORTING INTERACTIVE LAYOUT

Providing a layout for Relo presents a number of requirements not found by in popular graph layout algorithms. Since Relo presents nodes in a class-diagram like view, nodes need to be shown in expected locations for such diagrams. In common UML class diagrams, method call relationships go from the left to the right; inheritance runs from the bottom to the top, and containment is shown by nesting. Code elements in Relo therefore should to be laid-out using these same constraints.

5.2.1 Building an Incremental Interactive Layout Engine

The first algorithm used by Relo, and the most popular one available in most graph tools, was based on the orthogonal graph layout algorithm [70][77]. These algorithms work by building a representation of the various horizontal and vertical constraints and then iterating to minimize edge crossovers. Such algorithms work hard on providing a globally optimal layout. However, with this approach the addition of even a new node or edge to a diagram often causes all nodes to be repositioned. Thus using such a layout algorithm results in a lack of consistency of node positioning as the diagram is being built, and can distract users from their comprehension related task.

In order to have a fluid automatic layout of the graph, a simple approach is to use a force-directed approach for laying out code elements. Such approaches implement node attraction and repulsion as forces and iterate the node positions to minimize the applied forces [94]. These forces were extended for use in Relo to support directed constraints and graph containment. The presence

of such forces results in the layout engine having a number of local minima. These local minima result in the layout not being perfect but behaving predictably. Furthermore, newly added nodes do not cause the remainder of the graph to significantly re-layout. The significant re-layouts do not happen as doing so requires nodes to move on top of other nodes and the node-node repulsive forces prevent this from happening. However, while force-directed layouts do support incremental layout, adding nodes does cause the forces to change and does still result in minor unexpected movements of nearby nodes. Additionally, such force-directed approaches are limited in their need to model all constraints as forces, and perform poorly when needing to support complex constraints. Adding support for simple constraints like aligning disconnected or vertically-connected nodes in a column are hard – and exponentially harder when one of these nodes has a relationship associated with it.

Relo therefore has evolved to use a simple rule-based engine with support for common layout cases built into the engine via a set of rules. While the engine might not perform well in some edge cases, the rules are designed for commonly occurring scenarios and thus nodes in such scenarios are well laid out. A user can benefit from the auto-layout for most of the time, but when things get messy can either layout the nodes himself or consider eliminating nodes.

5.2.2 Rules Based Layout

The Relo engine uses a simple set of rules for the layout. The rules depend on the *origin* for adding nodes. The origin is the perceived users' focus on the diagram:

- In an empty view, the origin is at the center.
- When a user is trying to follow a relationship, then the origin represents the originating node where the request was made.
- In other cases, such as with linked exploration, choosing the location of the most recent addition can result in the diagram only expanding on one side, and thus getting too wide or too tall. In such cases, the location of the most recent addition is used only half the time, and the rest of the time the origin is chosen to be the center of the diagram – the selection of which strategy to use as the origin in such cases is chosen randomly.

Once the origin is found, new nodes are positioned on the origin. Placing nodes at the origin allows such nodes to be automatically aligned with the originating node or position when the rules are enforced. The rules are repeatedly asserted to enforced them by an engine which then also check and breaks any node overlaps. There are two main rules that need to be supported:

- Support for Directed Relationships – A large part of the complexity of the layout in Relo is in supporting directed relationships, i.e. the sup-

port for some relationships to be shown from top-to-bottom or from left-to-right. In these cases, nodes are laid out to follow the directional constraint associated with the relationship. The rules provides more sophisticated support for when there are multiple nodes that need to be positioned with the same directional constraint and same target node. Just enforcing the directional constraint first and breaking overlaps second will cause the placed nodes to not be organized with respect to the first part, i.e. directional constraints. In such cases, the rule lays out the nodes in a more organized manner, as shown in Figure 42.

- Support for Undirected Relationships – In this case nodes have more freedom to be positioned in contrast to the directed relationships. Nodes are simply laid-out in a clock-wise manner around the target node. The layout start in a similar manner as for directed relationships, but then nodes go around the target node so as to minimize the relationship length. Figure 43 shows how the rules work for the layout of an undirected relationship.

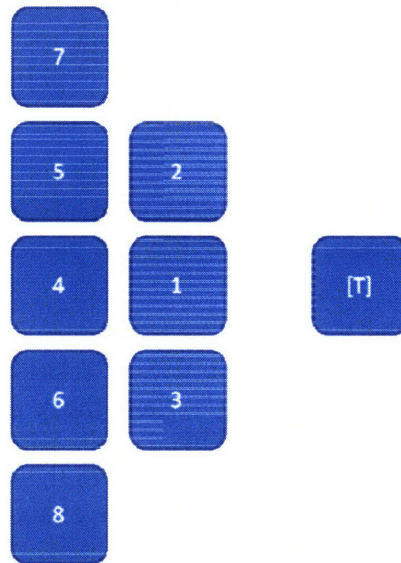


Figure 42 – Layout rule for directed relationships

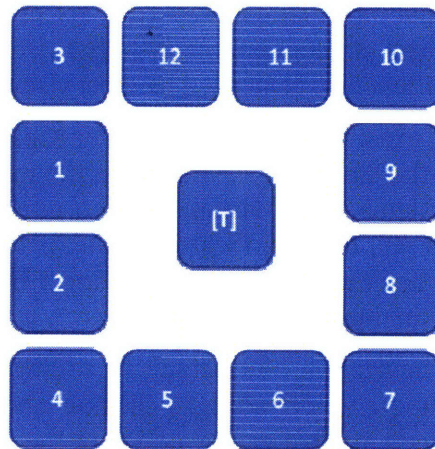


Figure 43 – Layout rule for directed relationships

The engine does not directly support the layout of unconnected nodes. In such a case, nodes are all positioned on the origin, but the overlap breaking part of the engine positions the nodes in available places.

5.3. SUPPORTING INTERACTIVE EXPLORATION

Relo provides a number of mechanisms to allow a developer to interactively explore and effectively focus on code elements that matter to his current task. Such exploration capabilities target small incremental changes in the diagram so that a developer can maintain a consistent mental-model of the code.

5.3.1 Navigation Buds

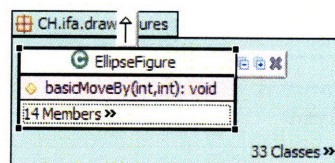


Figure 44 – Clicking on the class to show its buds (as in Figure 8)

A developer can browse the code in a Relo diagram using *buds* to navigate and extend the visualization with simple clicks. A bud is a context sensitive button on the currently-selected code element. For example, as shown in Figure 44, when the `EllipseFigure` class is selected, it will sprout buds for the different relationships that could be followed from the class – in this case the extends relation. Clicking on the bud will make the visualization grow by showing more items having the relationship, i.e. following the selected buds relationship. Buds are only shown when they will result in a modification of the view, i.e. as in Figure 44 a class that is not extended by other classes will not show the extended-by bud. Buds provide support for a browsing behavior commonly observed in users trying to *home-in* on information based on the

5. RELO USER INTERFACE

surrounding contextual information [89]. They are provided instead of navigation property dialogs or context menus, that are used for configuring shown or filtered relationships as needed by most visualization tools.

The navigation buds, while primarily designed for exploring, also provide features for controlling the diagram growth. When the mouse hovers over a navigation bud to add items to the visualization, a preview is provided of the number of items that will be added when the bud is clicked – such previews have been shown to be helpful to users in getting a better understanding of the information they are looking at [27].

While clicking on a navigation bud default to adding the items on the diagram, the user can choose to instead right-click on a bud to get a menu of the names of the code elements that can be added to the diagram. Further, Relo limits the number of items that are added during exploration – when five or more items need to be added to the diagram, clicking on the button again shows the menu of items to allow the user to only add relevant items.

RELATIONSHIP MODEL

Elements of Relo visualizations have been chosen heuristically based on formative evaluations with users. Relo elements include packages, classes (including nested and anonymous classes), fields, and methods (including constructors). The relationships used by Relo are shown in Table 6. The most common relationship explorations that a developer will want to perform are made available as navigation buds, to make them easily accessible. Complex relationship explorations, such as showing the entire subclass tree of a class, are available through a context-sensitive menu.

Table 6: The Java Relationships Model

Navigation Bud	Context Menu	Relationship	From	To
✓	✓ ⁹	Inheritance	Class	Class
✓	✓	Method override	Method	Method
	✓	Field access	Method	Field
	✓	Field modify	Method	Field
✓	✓	Field type	Field	Class
✓	✓	Containment	Package	Class

⁹ For inheritance in the context menu we also allow users to open the inheritance hierarchy, i.e. the transitive closure of the relation.

Navigation Bud	Context Menu	Relationship	From	To
			Class	Class
				Method
				Field
	✓	Method param.	Method	Class
✓	✓	Method calls	Method	Method

This relationships model is similar to that of concern graphs [67]. When created ahead of time and provided to developers concern graphs have been shown effective in representing the relevant portions of code for a code maintenance task. While concern graphs are typically displayed to users as trees, Relo is more diagrammatic and focused on exploration capabilities. Instead of focusing on describing these concerns, we focus on those relationships that could be used in understanding code. In our model, instantiations of classes are represented by calls to one of its constructors. We have chosen not to show relationships to and from local variables as they represent low-level code details which are not needed when understanding code interacting with multiple classes as is often the cases in last projects.

These relationships can be extended to provide exploration of an object model by performing lightweight analysis [41], however, we chose to focus on the static relationships in Relo.

5.3.2 Levels of Detail

Relo diagrams try to show as little information as possible. The user can then semantically zoom-in by double clicking on an element or selecting the expand bud ('+') to show more details. For classes, this means starting with only the class name, and at the first expansion level showing the children members having public access, followed by protected and private access. For methods, expansion shows the method implementation in an editor view. Instead of expanding a class to show all public members, the user can also use the *more items* menu to get a list of members and add only the relevant items. This keeps the diagram from growing too fast and at the same time provides the user with fine-grained controls.

Developers can also reduce the size of the diagrams. They can collapse code elements by clicking on the '-' handle. Alternatively, developers can selectively remove code element from the diagram by clicking on the 'x' handle. Since Relo is providing a view of the codebase navigation buds do not remove items from the actual codebase, but only remove the items from the diagram. When

items are removed from the view, to ensure that developers think the items are hidden and not deleted, the system animates them moving to the *more items* menu. Developers can also rapidly remove multiple code elements from the visualization by selecting multiple items and clicking on the group's navigation bud.

5.3.3 Autobrowse

As part of exploring through the code, developers often need to know how various code elements are connected. This might be to find out how a particular piece of the underlying system is launched or to find out how to update a particular interface using some data already available to the system. When trying to connect two code elements the high branching factor of the code can get developers lost easily. Autobrowse is a feature of Relo designed to help in these situations. It tries to simulate a user browse through the code and adds successfully found code paths to the diagram. Autobrowse requires two or more starting points, given to the system by the user selecting and right clicking the code-element. It then locates and shows short paths of relationships between the selected elements, thereby showing how the elements are connected. It does this by performing a simple breadth first search for hidden artifacts that are connected (and therefore relevant) to the given items. Some relationships, like inheritance, are more important in cases like this and for any given path length they are searched first. The search terminates as soon as at least one path is found, displaying the found path. Developers can repeat autobrowse to add longer paths.

Items removed from the diagrams are added internally to a list of items likely not relevant to the current task. When autobrowse is performed, any items that are in this likely not relevant list are ignored. This can allow a developer to select two code elements, run autobrowse, and if he finds the found path is not relevant it can be removed to run autobrowse again. Since the system tracks the removed items, it will not automatically add these items, thereby allowing the developer to see other paths. Such support for removed items are helpful to tell the system to ignore the obvious pieces of code connecting the entire system, like logging capabilities.

Beyond doing a breadth-first search, other approaches such as those done by Holmes and Murphy [38] can be used to improve the quality of results to users.

5.4. AUTOMATED DIAGRAM MANAGEMENT

Relo builds on the basic exploration capabilities by providing a set of view-based agents that are either live (continuously monitoring the visualization) or are triggered explicitly by the developer.

5.4.1 *Linked exploration*

While users might want to use Relo explicitly with the intent of understanding code, i.e. in a primary mode, Relo also provides support for those developers wanting to use the tools provided by the traditional environment (IDE), but still get the benefit of an incremental visual exploratory environment. Like Strata, Relo automatically tracks explorations made in other views of the IDE and extracts the relationships traversed. Use of the package explorer, call-hierarchy view, or the type-hierarchy view, result in the respective containment, method call, or inheritance relations being inferred and added to the Relo diagram. This allows the developer to work in a environment having more software development features, and at any time decide that he has possibly lost context and would like a Relo visualization to help him.

If the developer gets lost, say because he can't remember how the various tabs/views are connected to each other, he can open Relo which will use the exploration history to build a diagram. Since the exploration history may be large, Relo first shows a dialog box that allows the user to choose which elements to show.

With this bootstrapped diagram generated, the developer use Relo to explore further. Alternatively, Relo continues to track the developers' exploration and updates the visualization to help provide context to the developer, thus causing Relo to actively mirror the navigation.

5.4.2 *Automated Removal of Items*

With code elements being added explicitly via exploration in the diagrammatic interface or indirectly via linked exploration, the need for removal of irrelevant items exists. Such code elements exist in the diagram for one of two reasons: they either were mistakenly thought relevant at one time, or based on the current knowledge of the problem are not expected to be relevant.

Relo uses a simple approach similar to the degree of interest model which has been shown effective in capturing a task context to reduce items being shown [16][44]. Relo uses a score consisting of three components. The first is a selection count being increased when an element is selected, the second component represents the age of the item – on every action the selection count is reduced. The third component represents the interconnectedness, and is the number of shown relationships to or from the element.

Relo also provides an auto-remove mode, which automatically removes elements when more space is needed to display new items. When enabled items with low scores are treated as not being relevant and are the first to be removed when the visualization takes a large amount of space. The heuristic used for taking a large amount of space is $\frac{3}{4}$ of either the height or the width of the diagram being taken.

During linked exploration developers are primarily working in the IDE, and might not be focusing on the diagram to manage and remove elements shown. Auto-remove mode is therefore automatically enabled during linked exploration. The developer can also manually enable or disable auto-remove at any time.

5.5. SUPPORTING COMMUNICATIONS USING DIAGRAMS

By providing a lightweight method to provide various forms of annotations, developers can use Relo to not only understand an underlying codebase, but also to describe and communicate this understood knowledge along with domain knowledge via enhanced diagrams.

As developers understand code, their understanding moves from a structural model to a model consisting of data-flow and functional abstractions [13]. Relo helps users maintain these forms of understanding by providing support for basic types of annotations. For example, a system using the model-view-controller architecture, might typically have the associations between components carefully separated into factories in the code; however, a particular view of the code could have such relationships added by the developer. Relo allows developers to create named relations between items being shown and add comments to the explorations session, to allow the developers to represent formed higher level abstractions when examining the code [1]. These annotated diagrams can then be saved for future reference or for communicating with other developers.

Relo provides a panel which can be used by developers to add annotations to the diagram at any time during the exploration process. Once added, these diagrams can be saved using the IDE's menus, and thereby shared with other developers.

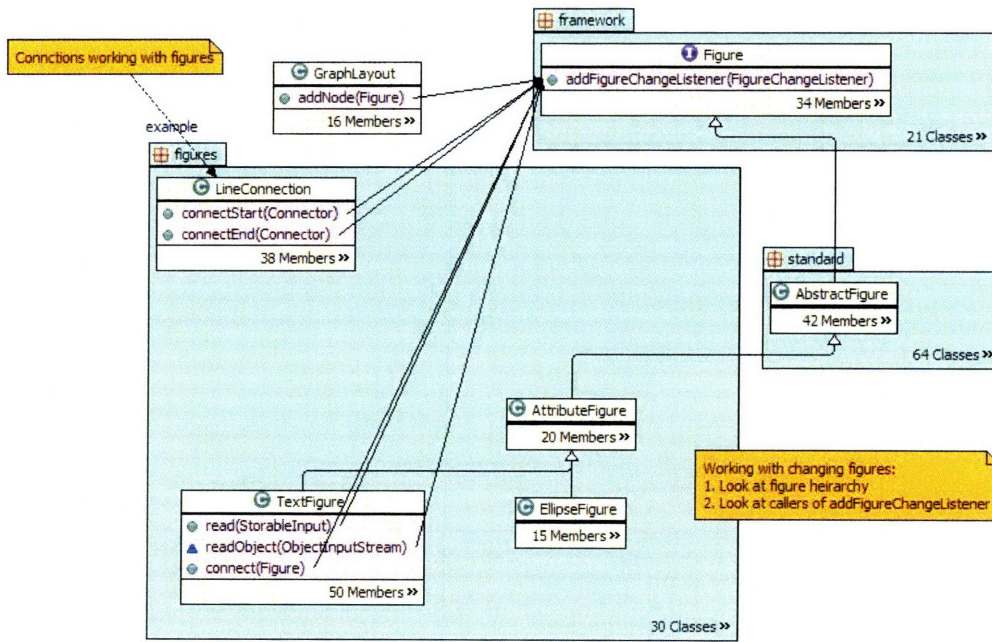


Figure 45 –Annotations in Relo

6. IMPLEMENTATION

Building a visual exploration environment for helping developers understand code has a number of unexpected challenges. These challenges are described below along with a description of core architectural components.

6.1. DEVELOPMENT CHALLENGES

The first prototypes of Relo and Strata took a month each to be built. Since then the tool has been iterated for a 3-year period. While the core ideas could have been evaluated easily in a non software-engineering domain, the need for their realization has only come about in applying the ideas towards developers. We therefore focused on building the tools out towards evaluating them in real-world situations. We describe some of the unexpected development requirements we realized when building such tools.

1. Developers are typically performing program comprehension to assist in some other primary task. This requires a program comprehension tool to support developers that are currently performing their primary tasks. Relo and Strata were therefore built on top of the widely used Java IDE Eclipse. However, the requirement of integrating into the developers' primary task has two key sub-requirements.

Firstly, the program comprehension tool needs to be integrated into the developer's current location of doing the tasks, i.e. his development environment. Beyond the obvious requirement of presenting a view inside the IDE, this requires an architecture to support the basic concepts of undo-redo commands and drag-n-drop. Further, developer's navigation events need to be tracked for conversion and use in Relo and Strata. These navigation events are typically in the form of multiple IDE events that need to

6. IMPLEMENTATION

be processed: text selection, structured (Eclipse Java Elements) selection, window openings, Eclipse Java Actions, etc.

Secondly, the tool needs to support a large breadth of functionality that the developer might use while performing his primary task. While functionality like following inheritance and method call relationships within Relo are obviously needed, capabilities like inferring these relationships when navigating in the IDE or providing integration with these IDE views into Relo are needed. Additionally, working with other IDE views like search and the various debug views is needed.

2. Eclipse being an Open Source IDE has its functionality reusable in many ways. However, using it in ways that it has not been used before often results in various IDE bugs being exposed. These include the lack of a common framework for reusing Views and Editors, the system to fail silently in certain cases for fire selection events, and the inability to persist view settings between Eclipse perspective switches.
3. Having good response times with large projects requires the caching of dependencies instead of a just-in-time parsing of the code. Building these dependencies quickly can be done by parsing the binaries at build time (instead of parsing the source), but doing so limits updates when compile errors exist. Further, since projects can be large these dependencies need to be written to disk instead of having significant memory requirements and therefore not working with large projects as can be experienced with some program comprehension tools [65].
4. Providing support for Java code not only requires supporting a significant portion of the Java Language Specification [33], but also supporting undocumented defaults used by the various compilers. For example, while Java developers have heard of and used anonymous inner classes, the Java compiler generates and uses anonymous methods for static initializers and for providing access to private methods. The Java parser needs to further resolve references in classes to those fields and methods in base classes.
5. Providing a tool for download to allow users to try and give feedback can take significant resources. Beyond traditional deployment obstacles like working well with 3rd party tools and ensuring that there are no thread deadlocks, development tools need to also worry and check both the version of the Java VM that is being used to run the tool and the version of the VM that is being used for the users tools.
6. In obtaining usage data about the various functionalities of Relo and Strata, there is also a need to capture and merge logs of usage of the core IDE. Beyond checking for the IDE selection service failing in the midst of getting data, there is a need for the logging code to ensure that it does not

adversely affect the IDE. For example initial attempts at logging users activities both slowed down the IDE and also unintentionally disabled some of the IDE's functionality. The logging service had to be modified to change the capture frequency and use different tactics to log different components.

6.2. A CACHING ARCHITECTURE

The need for integration into the developer's environment led Relo to be built on Eclipse. This required the building a mapping engine to connect the various representations used by Eclipse and Relo, then the support for basic caching for use in Relo, and finally support for compound relationships for use in Strata.

6.2.1 Mapping Support

Figure 46 shows the basic architecture of Relo as built in the initial prototype. In the figure, the Eclipse module represents the core IDE with Java support (JDT) built on top it. The Java support consists of two components the UI and the Core (consisting of the Java compiler). The diagramming capabilities are provided by using Eclipse GEF (graphical editing framework). Since Relo needs to be launched from the interface, it depends on and listens to the JDT-UI module. Further, in order to be able to query the source code Relo needs to use the JDT-Core directly as well. To allow for the seamless use of the JDT-UI and the JDT-Core, Relo provides for a mapping module to connect them to a fixed identifier representation.

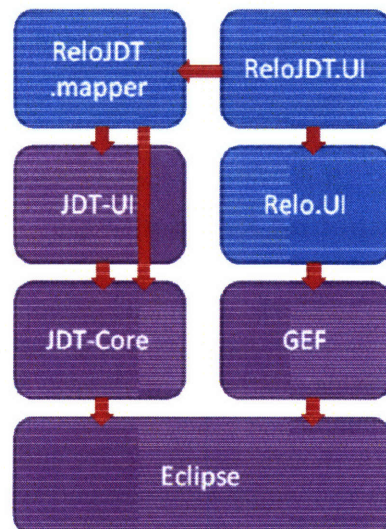


Figure 46 – Basic Relo Architecture.

6.2.2 Basic Caching using Builders

For Relo to work rapidly on large codebases and have services checking non-visible portions of the codebase without consuming a large memory footprint, Relo needs to translate the underlying codebase into a triples database. This triples database is based on the W3C standard RDF [48] and therefore allows for easier extension of Relo to other languages and domains. All relationships are represented in the form $\langle source, relationshipType, destination \rangle$ and the mapping engine connects this representation to the Eclipse builder framework (for building the cache) and both the JDT-UI and the JDT-Core. Supporting a new language thus requires just adding the relationships to the database and providing the mappings to and from the various Eclipse objects. Figure 47 shows the Relo architecture with support for caching via a store. As the diagram shows the Relo JDT mapper has been further optimized to get relationships from the binary Java .class files and therefore be able to extract relationships faster than parsing the source files.

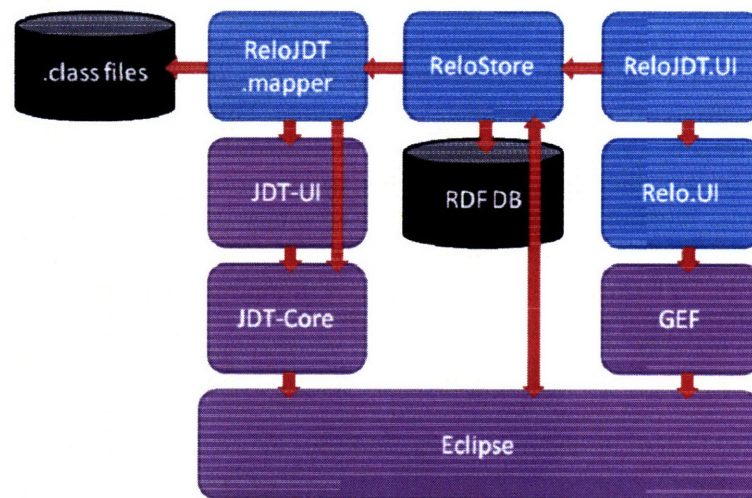


Figure 47 – Architecture with caching support for Relo.

6.2.3 Caching Compound Relationships

Support for Strata has been built on top of the diagramming and caching infrastructure provided by Relo. The architecture, shown in Figure 48, provides additional support for caching compound relationships. Beyond particular relationships between fields, methods, or classes, Strata shows users aggregated dependencies at higher levels and caches them in the store. In the architecture 'RC' is used to indicate the Relo Core and is a set of reusable components, while the Relo module provides support explicitly for the class diagram browsing.

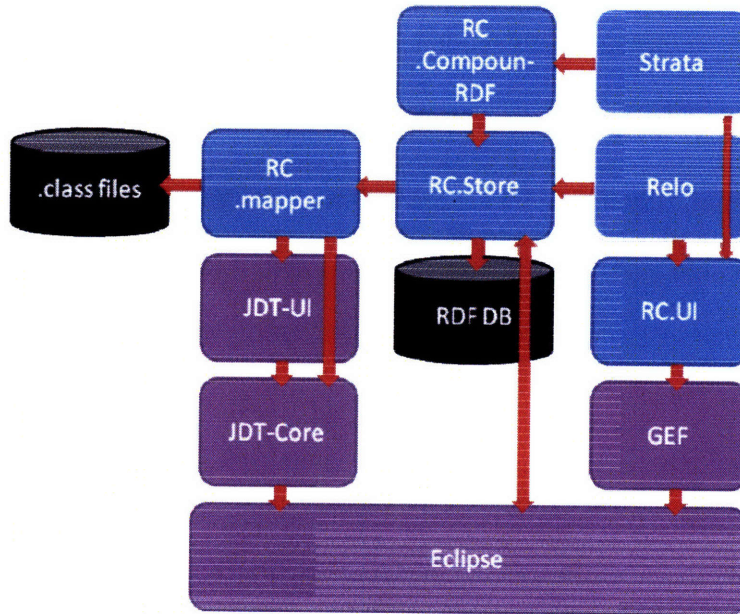


Figure 48 – Architecture with support for Strata.

6.3. AGENT FRAMEWORK

In providing an incremental exploration environment Relo also contains an agent framework. This framework allows agents to monitor shown code artifacts and make minor modifications to the generated visualization. For example, one such agent automatically draws the containing class or package when there are multiple artifacts that share the parent. This way by visual grouping Relo is able to reduce the information that needs to be understood by the developer even though it is adding information to the visualization. Similarly, Relo also draws direct inheritance relations between elements shown in the visualization.

Agents listen to one of a number of events, such as code artifact selection, creation, or other actions, and then either vary properties of the visualization, add elements, or remove them. These simple agents thus work together in providing a tool that feels intelligent. In cases when too many agent based modifications might happen, agents can apply their changes sequentially and use a short delay before their actions are taken thus giving users a feeling of control as the information is being organized by the system. For example, when a user clicks on the calls handle on a method, the called methods are first shown along with their calls relationships. After a half-second delay the parent assertion agent is triggered which might add the parents to the methods. This might be further followed by inheritance relationships being added to the diagram. Instead of overwhelming the users with all these items being added at once, the changes are ensured to be simple and are not only add-

6. IMPLEMENTATION

ed gradually, but can also be undone, and each agent can apply a set of its own rules to only perform actions when the perceived information shown is reduced.

With support for automatically adding elements by agents, Relo also needs to ensure that items removed by the user (say by using one of the handles) are not automatically added by an agent. For such cases, Relo keeps track of all user requested removals, and does not allow agents inferred code elements to be added once such an element has been removed by the user.

7. EVALUATIONS

Our primary focus during the evaluation of Relo and Strata was to evaluate the usability of the tools with multiple developer's given very limited training. This was because developers mostly understand code secondarily to some other task. Additionally since studies of most program comprehension activities and tools have focused on very small and therefore artificial code bases, we focused on evaluating the tools on projects and tasks from the real world.

7.1. USING STRATA WITH PROJECTS

The initial design for Strata was verified and polished using a set of open source projects as part of a preliminary formative evaluation. Design knowledge was extracted from examining the code, and by reading various forms of documentation. During this phase, we noticed that most tools only provided limited support for organizing the module dependencies, and that organizing these dependencies was important in getting an overview of the underlying codebase. Strata thus provided support for organizing these dependencies into layers and did so primarily by breaking cycles found in the code. We therefore performed an evaluation of Strata on an externally created collection of projects and measured the quality of the cycle breaking on these projects.

7. EVALUATIONS

Table 7: Projects used in determining Strata usefulness

Project	Description	Hosted	Col- lected ¹⁰	Size	
				LOC ¹¹	Classes
Azuereus	Popular GUI and Implementation of the BitTorrent protocol	sourceforge	Nov '06	434,676	2,277
Buddi	Program to manage personal finances and budgets	sourceforge	Nov '06	31,186	128
Cargo	API for managing J2EE containers	codehaus	Feb '07	64,726	518
Carol	Library for abstracting away different RMI (Remote Method Invocation) implementations.	objectweb	Feb '07	20,664	145
Sphinx	Speech recognition system	sourceforge/cmu	Nov '06	98,553	382
Commons-Codec	Implementation of common encoders and decoders	apache-jakarta	Nov '06	10,988	41
Dnsjava	Implementation of the DNS protocol	sourceforge	Feb '07	33,634	184
jEdit	Configurable text editor for programmers	sourceforge	Nov '06	159,908	514
jMemorize	Tool involving simulated flashcards to help memorize facts	sourceforge	Nov '06	17,187	95
Jmol	Tool for viewing chemical structures in 3D	sourceforge	Nov '06	100,131	291
JRuby	Implementation of the Ruby programming language	codehaus	Nov '06	82,081	427
JTrac	Web Application for issues-tracking	sourceforge	Nov '06	10,288	80
Radeox	API for rendering wiki markup	codehaus	Nov '06	11,811	179
Rssowl	Newsreader supporting RSS	sourceforge	Nov '06	81,647	201
TvBrowser	Extensible TV-guide program	sourceforge	Nov '06	149,564	777
Zimbra	A set of tools involving instant messaging	sourceforge	Nov '06	579,288	2,744

¹⁰ All projects were extracted from the head of source repository at the collection date.¹¹ LOC = Lines of Code

7.1.1 Methodology

The collection used for this evaluation was an externally provided collection of java projects (used in [53]). The projects were from popular open source web sites, including sourceforge.net, codehaus.org, and objectweb.org. 16 projects were selected based on popularity, and are listed in Table 7, along with descriptions, project sizes, and information on extracting the collection again.

For the study, we took each project and opened it in Strata. Since we were evaluating the effectiveness of cycle breaking, each project was opened with cycle breaking both enabled and disabled. When opening the projects, we noticed that when all the modules are expanded most projects have some cycles in them. We therefore wanted an objective method for selection of how many modules and which ones we wanted to show.

Strata uses a top-down expansion interface since having too many modules being shown can overwhelm users. We decided that a typical scenario for a developer would have around 10-12 modules. Having more might result in the users getting overwhelmed and having fewer might not be helpful for a developer trying to decide relevant components for his exploration. We decided that a typical developer would get the 10-12 modules by expanding the largest modules if either (a) there were less than 6 modules, or (b) the largest module was more than twice the second largest module. The expanded modules were then broken and layered amongst the rest of the modules.

We logged qualitative observations and extracted data on the number of modules shown, number of rows and number of columns.

7.1.2 Results & Discussion

Table 8 shows the data collected from expanding the various projects. The table includes the number of modules shown when the projects were expanded, the average number of characters per module, and the number of row and columns when our cycle breaking algorithm was applied or not.

7. EVALUATIONS

Table 8: Results from expanding the projects in Strata

Projects	Avg chars per module	Modules shown	Cycle breaking		Without breaking	
			# rows	# cols	# rows	# cols
sf-rssowl	13.84	13	6	3	3	11
sf-zimbra	19.21	37	25	4	4	32
sf-tvbrowser	12.06	30	11	5	5	19
sf-jmol	10.36	19	6	4	5	7
sf-jtrac	5.71	7	3	3	2	4
sf-jmemorize	11.25	8	6	2	2	5
sf-dnsjava	15.63	11	4	4	3	8
sf-cmusphinx	7.3	10	7	2	6	4
sf-azuereus	19	12	5	4	3	7
sf-buddi	5.833	6	4	2	1	6
owf-carol	4	6	3	3	3	3
jakarta-commons-codec	17.76	13	5	4	4	6
ch-jruby	6.17	17	11	2	1	17
ch-cargo	14.58	17	10	3	8	6
ch-radeox	5.25	8	7	2	5	4
sf-jedit	10.89	19	14	2	3	16

Any diagram that has too few layers will not be helpful. For example, the RSSOwl project being opened in Strata with cycle breaking is shown in Figure 49 and with the traditional partitioning algorithm is shown in Figure 50.

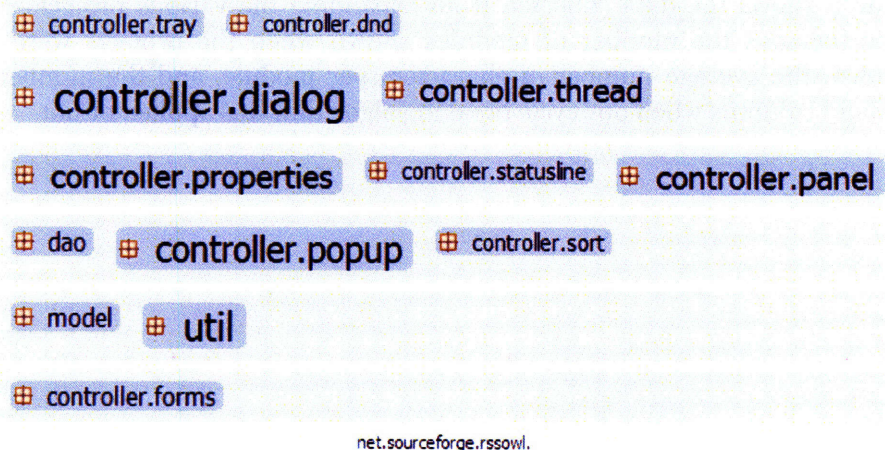


Figure 49 – RSSOwl with cycle breaking

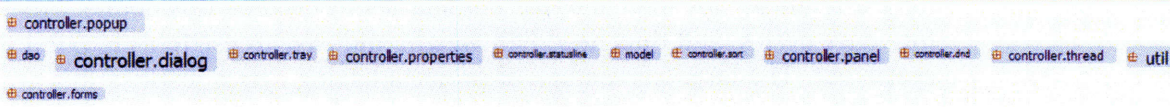


Figure 50 – Traditional partitioning with RSSOwl

We therefore plotted the number of columns as a percentage of the total number of modules in the diagram (Figure 51). The chart also includes the idea width of the layered diagram. We calculated the ideal width of the diagram to be the number of columns that would make the diagram a square – here estimated to be the square root of the number of modules. Another alternative would be to use the average number of characters per module and use an estimate of the height of a column.

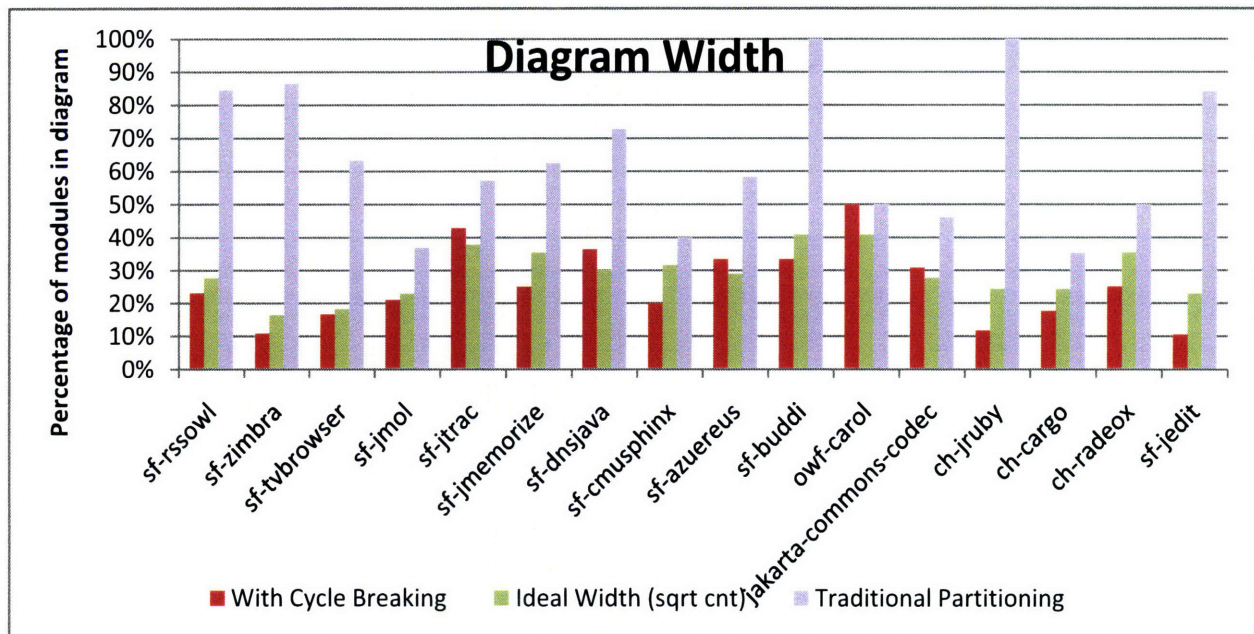
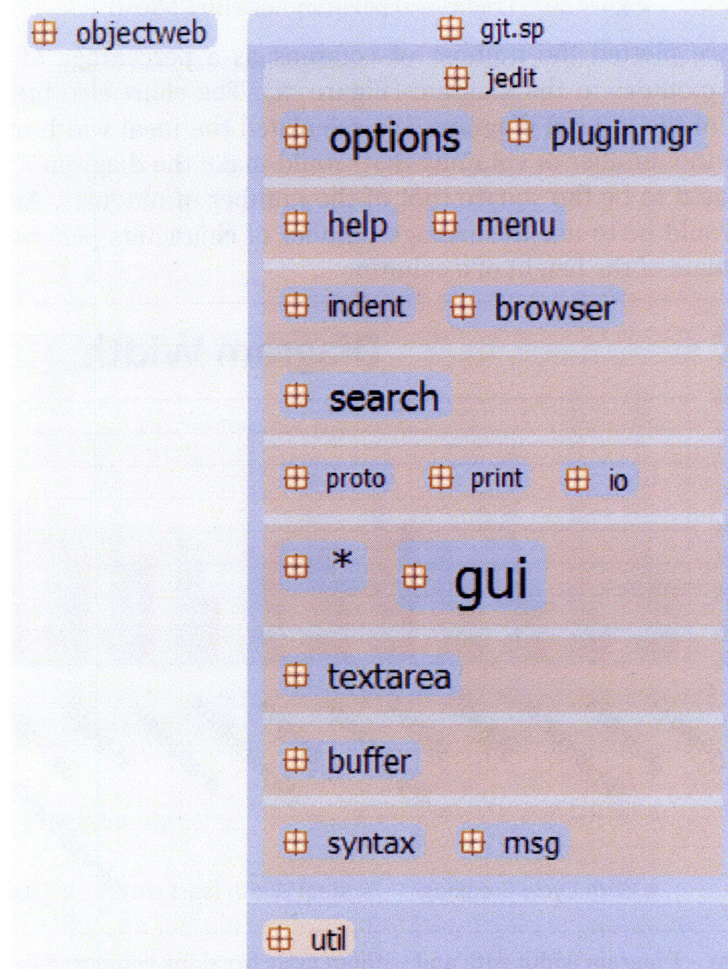


Figure 51 – Diagram width with and without cycle breaking compared to ideal

From the figure it can be seen that traditional partitioning (non cycle-breaking) diagrams are often very wide. 75% of the projects (12 of 16) had the diagram width consisting of half of all the modules, i.e. half the modules that were in the diagram were all on one row. On average the traditional partitioning diagrams width were 35.1% greater than the ideal width, while the cycle breaking algorithm resulted in an average of 3.6% less than the ideal width.

Among the 16 projects only 4 (25%) had no cycles at the level we expanded. And only 1 of these had no cycles at all when expanded completely. The project that had no cycles was the apache commons codec project, which is a library of independent codes and therefore can be expected to have independent modules and no cycles. In Chapter 0, we argued that software projects

have a tendency for their module boundaries to become less defined, therefore have cycles, and thus need lightweight and easy to use tools to point the cycles out. The large percentage of projects having cycles justifies this argument.



org.

**Figure 52 – Strata display of the jEdit project
(same as Figure 5 and Figure 30)**

We analyzed the cycles on the various projects to determine if our cycle breaking algorithm broke the cycles in places that made sense. We found that as expected the `util` modules were often at or near the bottom (also can be seen in Figure 49). Similarly, as can be seen with the `jEdit` project in Figure 52, the algorithm often put the `gui` modules in higher layers. In the `Buddi` project, as can be seen in Figure 53, the `view` and the `controller` depended on the `model`. Further, as in Figure 54, among the `view` and the `controller`,

the `controller` classes mostly depended on the `view` classes with the exception of a few core `controller` classes (like the accessibility support).

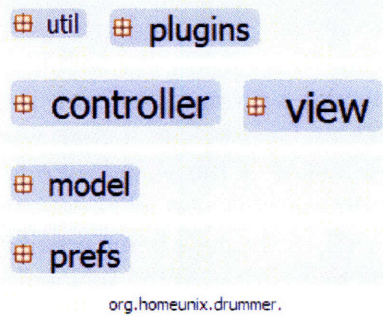


Figure 53 – Strata display of the `buddi` project



Figure 54 – Strata display of the view and controller of the `buddi` project

7.2. USER-STUDY WITH STRATA

With support for cycle breaking built and tested with various projects we wanted user-feedback of developers using Strata on their own codebases. Our goal was both to verify the need for the tool, but also to get feedback on layout, and understanding requirements of the tool. We did this by conducting studies with developers on seven projects.

7.2.1 Methodology

We worked with the primary developers on seven projects. System A and System B were proprietary and we were asked to not include some details not

relevant to the case study (such as the names of the projects and the clients they were actually implemented for).

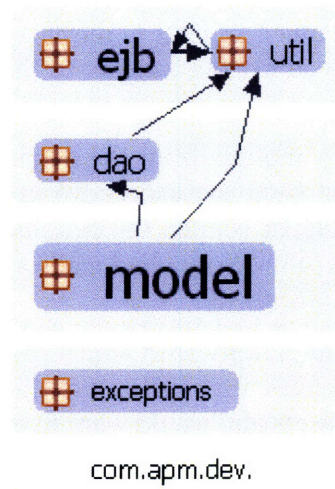
We performed a demonstration of Strata on one of our test codebases, and asked the developers to install Strata and give us feedback on the tool. We asked a number of questions regarding the tool, the approach, and their expectations of it. In particular, we wanted to know if they liked the layered approach and the manner or layering, or if they wanted a different form of module layout. We also asked if the visualizations they observed matched their understanding of the systems, and what deviations they identified (such as Strata not behaving well, or the code being possibly wrong). We tried to get an understanding of the helpfulness of the tool and gather any feedback that they might have had on it.

7.2.2 Results & Discussion

Developers using Strata generally found the tool helpful. At the beginning of the session, some were initially suspect of the benefits of Strata. This was partially because they were typically very busy and wanted to get back to their work, but was also magnified by most projects' tendency to organize at the highest level modules that represent overrides of externally provided functionality. Users needed to be informed that if the modules seemed irrelevant at the top-most level, they should choose the biggest or most familiar module and double-click on it for expanding (and that they could look at the other modules later). Once the top most module was expanded, developers began to quickly get an understanding of the benefits of Strata.

All except one user explicitly mentioned that they liked the layers with one user saying "what makes sense here is organizing by dependencies". Part of what the developers liked about the diagram was its compactness compared to traditional diagramming tools mentioning happily that "it was like trees but it also had arrows".

Users liked being able to mouse over modules so that the dependencies to and from it were shown. Users felt that the layers building upwards were natural.



**Figure 55 – One of the higher-level views of System A.
Figure shown with all non-downward dependencies being drawn**

System A, shown in Figure 55, shows the display after expanding the largest module for System A and shows all dependencies that need to be examined, i.e. dependencies that are not going downwards. From the figure, it can be seen that some of the modules use the classes in the `exceptions` module. In addition, as in this case developers found that the layerings were close to expectations.

In 6 of 7 cases, the developers were able to very quickly point out “the one wrong thing with the diagram”, and were curious as to the reason for the deviation. The reasons for deviations varied. Sometimes they were related to likely errors in the implementation of the underlying project. In one case the deviation pointed the developer to a part of the project where they would want to conduct a more extensive review with the shown information. And in another cases it reminded of refactorings that needed to be completed. The developers also mentioned that the layering gave good insight into the code, and one developer mentioned wanting to use the diagrams as a starting point to get new developers to understand the structure of the code. 5 of 7 developers mentioned finding interesting things that they knew at one time but had forgotten.

Users appreciated that the sizing indicated importance of the modules, and gravitated towards the larger ones. But this also made them ask for better support for the important items. Other than emphasizing modules based on code size some users wanted to emphasize modules based on the “time I [just] spent in it” or “time I spend [editing it] over the last month”. There was no general agreement on the right metric for importance and emphasizing the modules. Related to choosing the appropriate metric is that there seems a need to also bias weights related to their context when launching Strata. By

7. EVALUATIONS

default Strata shows dependencies from the current project to any code being used, which also includes other projects in the current workspace. Users liked this but wanted Strata to adjust the weights to the current project and emphasize those modules.

3 users wanted to move a module from the default. For one user this happened because Strata had been too aggressive in ignoring dependencies to break cycles and the developer wanted the two modules of the cycle to be in the same layer. In 2 cases (28%) Strata had made an “obvious” mistake which the developers wanted to quickly fix.

Developers appreciated the fluidity with which they could look at more granular aspects of their projects, especially when they would find unexpected dependencies in their projects and would want to examine the set of involved modules in more detail.

As can be noticed from Figure 55, a limitation of the default views shown in Strata was that they sometimes showed the modules along technology boundaries – placing all code accessing the database in the `dao` module and the business logic in the `ejb` module. Having modules organized along technology boundaries is not the default way of thinking about the project. While the technical separations were useful, the business domain diagrams were expected to be the default. The developer did appreciate being able to expand the modules in Figure 55, group classes and save them as modules, to quickly create their own module descriptions for opening them with Strata (as shown in Figure 56).

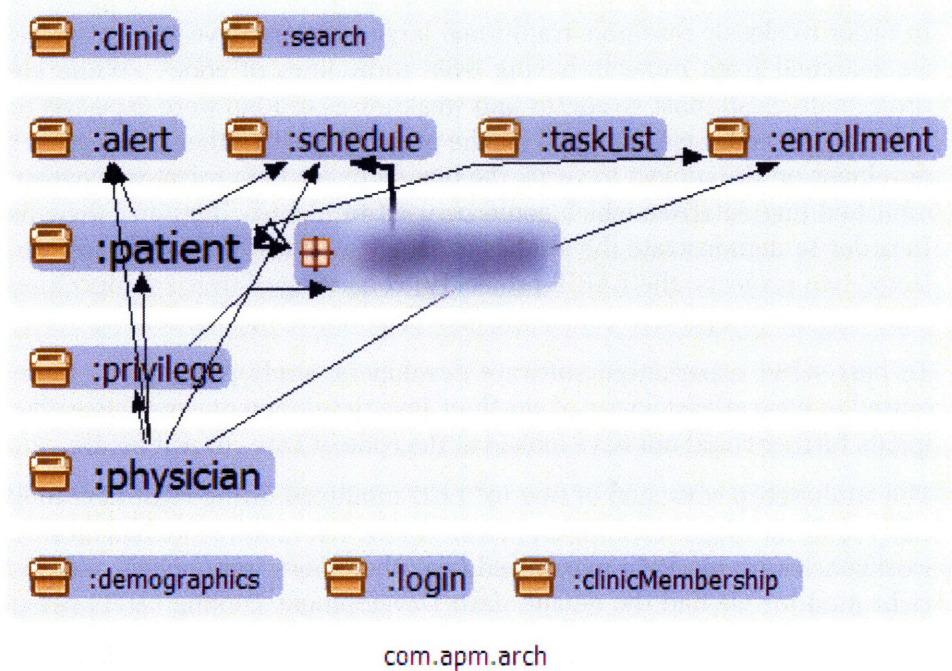


Figure 56 – Business domain view of System “A” using Strata.

Developers wanted to use Strata in different ways. While some developers were interested in why the diagrams did not appear as expected, other developers seemed to focus more on the diagrams as the reality and wanted to understand the impact of their changes. Developers also wanted Strata to help in situating themselves, by providing an indication in the diagrams of the code elements involved in their current task. They felt that Strata would be useful for both a new project that they were joining as well as the current project that they were working on.

Two limitations of Strata were felt by user. The first one was that developers would have liked to have the architectural diagrams be based on runtime data as well. The other need was for Strata to connect to currently available design documents and provide verification capabilities.

7.3. USER-STUDY WITH RELO

In order to evaluate the strengths and weaknesses of Relo, we conducted a controlled user-study of Relo on bug fixing tasks. Over a period of 6 months we iterated through 15 users while refining the study and Relo. We then ran a controlled study on 13 users. The goal of the study was to demonstrate the validity of approach used in Relo. Secondary goals were to detect usability issues in Relo, and gain more insight into developers’ decision processes when using tools to help in the comprehension of large projects.

7.3.1 Method

In order to closely resemble traditional large software development projects, we searched from projects having over 100K lines of code¹². While larger projects do exist, most strengths and weaknesses of Relo were expected to be found in the codebase. The goal for the projects was to either have access to a developer on the project to verify the bug fix or for the project to have an established bug database which could be used to identify bugs and their fixes. In order to demonstrate the studies accuracy over varying codebases, we selected two projects: the LAPIS project [7] consisting of over 150,000 lines of code and the Ant project consisting of over 200,000 lines of code.

To best select experienced software developers, study participants were required to have atleast a year of worth of Java development experience. Participants further could not have looked at the code of Lapis or Ant before.

The study setup consisted of two 19" LCD monitors sitting right next to each other and running at 1280x1024 pixels. On the left monitor we had an Eclipse workbench running Relo maximized (no other views were open), and on the right monitor we had the default Java Development Tooling (JDT) perspective of Eclipse. Study participants were informed that they could change this configuration, but no one did so.

After a short description and demo of Relo, study participants were given three tasks. The first task was a warm-up task during which the study facilitator helped them with both the task and the Relo features. Users were then given a bug to find the solution to from Ant and Lapis. The bugs were chosen from the projects bug database, and the first bugs that were deterministic, reproducible, and not OS-specific were selected for the task. In the tasks the developers did not have to write the needed code, but had to mention the exact location/cause of the bug or implementation location of the feature addition. After the tasks users were asked to answer questions that measured their understanding of the system.

Participants used a think-aloud protocol [52][63], and their actions were recorded by screen capture software and event logging. The study concluded with a questionnaire and a short semi-structured interview.

THE STUDY TASKS

Both study tasks included a description of the task, and a set of short hints. The hints were designed based on the information provided in the tasks questions, as opposed to using any information or understanding from the codebase. Users were told to do the task and continue when they knew exactly what needed to be changed to fix the bug.

¹² Size of code base was measured using the command 'wc'.

The Ant task (Figure 57, Figure 58, Figure 59 and Figure 60) required fixing an old bug that we had selected from the Ant bug database. We chose an old bug so that we could have a verified solution to the bug. The chosen bug had id 37426¹³ and also included a test-case to reproduce the bug attached originally to the bug descriptions. Through preliminary studies questions we realized and added a small background on Ant as well.

The Lapis task (Figure 61 and Figure 62) required adding a small feature which had components already implemented to the codebase.

Background

This bug is from the Ant project when providing support for JUnit tasks. Below is an example of a simple build file for ant.

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="build" name="ant-test">
  <path id="ant-test.classpath">
    <pathelement location="bin"/>
  </path>
  <target name="build">
    <javac debug="true" debuglevel="${debuglevel}"
destdir="bin" source="${source}" target="${target}">
      <src path="src"/>
      <classpath refid="ant-test.classpath"/>
    </javac>
  </target>
</project>
```

Figure 57 – Background for the Ant task.

¹³ http://issues.apache.org/bugzilla/show_bug.cgi?id=37426 on Ant version 1.6.5

7. EVALUATIONS

What is the cause of the below bug? What needs to be modified to fix it?

Reported Bug

Summary: task doesn't print all the Test names when using forkmode='once'

Description:

Hi,

if you use forkmode='once' for your junit task, the names of tests are not printed to the console. Only 1 test-name is printed!!!

for example:

```
junit:
[junit] Running org.example.MyTest3
[junit] Tests run: 1, Failures: 0, Errors: 0, Time
elapsed: 0,05 sec
[junit] Tests run: 4, Failures: 0, Errors: 0, Time
elapsed: 0,01 sec
[junit] Tests run: 4, Failures: 0, Errors: 0, Time
elapsed: 0 sec
```

However,
the expected output is (which is the same output as you get when using forkmode='perTest'):

```
junit:
[junit] Running org.example.MyTest1
[junit] Tests run: 1, Failures: 0, Errors: 0, Time
elapsed: 0,05 sec
[junit] Running org.example.MyTest2
[junit] Tests run: 4, Failures: 0, Errors: 0, Time
elapsed: 0,01 sec
[junit] Running org.example.MyTest3
[junit] Tests run: 4, Failures: 0, Errors: 0, Time
elapsed: 0 sec
```

Figure 58 – The Ant task.

Test Case:

The lines below show a junit test case, and provide the options to reproduce the bug. Running the test case should require running the Main.main method in the project with the arguments '-buildfile text.xml test' -- test.xml --

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="build" name="ant-test">
  <path id="ant-test.classpath">
    <pathelement location="bin"/>
  </path>
  <target name="build">
    <javac debug="true" debuglevel="${debuglevel}"
destdir="bin" source="${source}" target="${target}">
      <src path="src"/>
      <classpath refid="ant-test.classpath"/>
    </javac>
  </target>
  <target name="test">
    <!--
    <junit fork="on" forkmode="perTest" printsummary="on">
    <junit fork="on" forkmode="once" printsummary="on">
    -->
    <junit fork="on" forkmode="once" printsummary="on">
      <test name="tstPckg.HelloTest" />
      <test name="tstPckg.AnotherTest" />
      <classpath refid="ant-test.classpath"/>
    </junit>
  </target>
</project>
```

Figure 59 – The test case for the Ant task.

Hints (try answering these to figure out the bug):

1. Where is the correct code being outputted called from, i.e. what code displays "Tests run"?
2. Where is the code that is only being printed in some cases called from, i.e. what code displays "Running"?
3. How does the execution of the unit tests call the two different outputs? What code needs to be changed to have the desired behavior as needed by the bug

Figure 60 – Hints provided for the Ant bug.

The Named Patterns list has a count in parentheses after each pattern. This count doesn't update often enough. At the very least, it should update when the user clicks on it. For example, suppose you start with a blank document (File/New File). Then search for English\Word in the named patterns list; you'll see that the Word entry in Named Patterns has (0) after it. Now edit the document to add some words: "some text here". Word still has (0) after it, which is updated if you collapse and expands the "English" node. Fix it so that clicking on the Word entry in the Named Patterns list updates it.

Figure 61 – The Lapis task.

Hints (try answering these to figure out the bug):

1. What code is responsible for the "Named Patterns" list, i.e. where was it created?
2. Where should you be adding the fix, i.e. what gets called when a node is selected in the Named Patterns list?
3. What already implements the functionality that needs to be added, i.e. what gets called when the node is expanded?

Figure 62 – Hints for the Lapis task.

COMPREHENSION QUESTIONS

To measure users understanding during the task we chose to use ask users a set of questions about the codebase after the fixed the bug. In order to ensure that these questions were not biased, the questions were selected based on a study done by Sillito et al. [78] on the questions programmers ask when doing such coding tasks. There are 8 such types of questions and are listed in Figure 63 and Figure 64. Before the study we asked 4 users to do the bug-fixing tasks and then told them to generate a question based on each type. These 4 users then voted on the questions generated by the other 3 users, and we picked 1 question of each type.

1. FINDING INITIAL FOCUS POINTS

- Which type represents this domain concept or this UI element or action?
- Where in the code is the text in this error message or UI element?
- Where is there any code involved in the implementation of this behavior?
- Is there a precedent or exemplar for this?
- Is there an entity named something like this in that unit (for example in a project, package or class)?

2. BUILDING ON POINTS - QUESTIONS ABOUT TYPES

- What are the parts of this type?
- Which types is this type a part of?
- Where does this type fit in the type hierarchy?
- Does this type have any siblings in the type hierarchy?
- Where is this field declared in the type hierarchy?
- Who implements this interface or these abstract methods?

3. BUILDING ON POINTS - ENTITIES AND RELATIONSHIPS FOR INCOMING CONNECTIONS

- Where is this method called or type referenced?
- When during the execution is this method called?
- Where are instances of this class created?
- Where is this variable or data structure being accessed?
- What data can we access from this object?

4. BUILDING ON POINTS - ENTITIES AND RELATIONSHIPS FOR OUTGOING CONNECTIONS

- What does the declaration or definition of this look like?
- What are the arguments to this function?
- What are the values of these arguments at runtime?
- What data is being modified in this code?

Figure 63 – Questions Programmers Ask During Software Evolution Tasks – Part I.

5. UNDERSTANDING A GRAPH - BEHAVIOR AND LOGIC

- How are instances of these types created and assembled?
- How are these types or objects related? (whole-part)
- How is this feature or concern (object ownership, UI control, etc) implemented?
- What in this structure distinguishes these cases?
- What is the behavior these types provide together and how is it distributed over the types?
- What is the 'correct' way to use or access this data structure?
- How does this data structure look at runtime?

6. UNDERSTANDING A GRAPH - DATA AND CONTROL-FLOW

- How can data be passed to (or accessed at) this point in the code?
- How is control getting (from here to) here?
- Why isn't control reaching this point in the code?
- Which execution path is being taken in this case?
- Under what circumstances is this method called or exception thrown?
- What parts of this data structure are accessed in this code?

7. GROUPS OF GRAPHS - RELATIONSHIPS BETWEEN MULTIPLE GRAPHS

- How does the system behavior vary over these types or cases?
- What are the differences between these files or types?
- What is the difference between these similar parts of the code (e.g., between sets of methods)?
- What is the mapping between these UI types and these model types?

8. GROUPS OF GRAPHS - CHANGING GRAPHS AND THEIR IMPACT ON THE SYSTEM

- Where should this branch be inserted or how should this case be handled?
- Where in the UI should this functionality be added?
- To move this feature into this code what else needs to be moved?
- How can we know this object has been created and initialized correctly?
- What will be (or has been) the direct impact of this change?
- What will be the total impact of this change?
- Will this completely solve the problem or provide the enhancement?

Figure 64 – Questions Programmers Ask During Software Evolution Tasks – Part II.

7.3.2 Results & Discussion

The exploration capabilities are an important aspect of Relo, and we found that participants liked the ability provided by navigation buds to understand the surroundings fast, especially the ability to click on the buds and rapidly see the different code artifacts connected by the various relationships. Four of 12 participants mentioned during the study (without prompting) that they liked the ability to examine the code quickly. Participants found that the navigation buds gave them good control of the generated diagrams. They did feel that they were able to access code that they wanted (4.6 on a 7-point Likert scale). Users seemed to be able to use to get more easily situated in the codebase, allowing them to explore around faster. One missing feature mentioned was an integrated search mechanism for the entire codebase; while Relo does support exploring from a given starting point, searches need to be performed in a separate tool and then brought into Relo. Participants also felt slowed down by Relo when the navigation buds would result in around six items or more – in the JDT, these developers were able to use the keyboard quickly to navigate such lists faster than a mouse could be used to manage such code artifacts. Better keyboard access to Relo might help in such cases.

Developers using previous visualization tools have found them to be overwhelming [85]. By contrast, participants did not find Relo to be overwhelming (2.6 on a 7-point Likert scale¹⁴ to the assertion of the tool being overwhelming). Five participants indicated strong disagreement, while the remaining participants indicated during the interview that their sense of being overwhelmed was really caused by the task and the large codebase given to them, while Relo was very helpful for the task.

Participants also understood the code artifacts that were being automatically added (like parent class/packages and inheritance relationships). They did not feel helpless with manipulation of the interface. Even given the large tasks 32.96% of items added to Relo were added by the user explicitly, with the rest being added automatically via selection. They found the tool-tips useful in explaining the various navigation buds, and liked the capability of being able to organize the artifacts and their relationships into “something relevant”. Five of the participants mentioned wishing that they had the tool earlier for their previous projects, and three other participants mentioned the tool would be helpful in understanding code written by others.

We measured understanding by asking those users that made significant progress with each task to answer the questions related to it. To measure progress we created a set of 8 milestones that each user would hit in completing the task. They were:

¹⁴ In the Likert scale, 1 represented ‘strong disagreement’, 3 represented ‘disagreement’ 4 represented ‘neutral’, 5 represented ‘agreement’ and 7 represented ‘strong agreement’.

7. EVALUATIONS

1. This happened when the participants started looking at the code. Looking at the main method in a project would qualify here.
2. **Breadth First:** This represented the stage if a user would systematically try to understand the entire codebase. While rare in experienced developers, some of our participants had not worked with such large projects before and did try to understand the codebase completely.
3. **Depth First:** This represented the stage when a user would start at some basic point in the code and then follow references to various fields in order to understand the codebase. While not purely a depth first exploration, this strategy is relatively a lot more depth-first than the breadth first strategy. Users that hit this milestone often made good progress.
4. **Found Hint 1:** Users that were able to find the relevant part of the code relevant to the first hint hit this milestone.
5. **Found Hint 2:** Users that were able to find the relevant part of the code relevant to the second hint hit this milestone.
6. **Found Hint 3:** Users that were able to find the relevant part of the code relevant to the third hint hit this milestone.
7. Users hit this milestone when they were able to suggest an answer, but were not yet comfortable enough to say with any certainty that it was the solution.
8. Users hit this milestone when users were comfortable with their answer.

Some users did make progress by using other avenues than the provided hints. However, in all such cases users did need to hit two main distinct code locations specific to their strategy (milestone 4 and 5), and did need to pull together the information to solve the task (milestone 6).

For example, consider a user doing the Lapis task with Relo. On hitting milestone 4, he would have found the code responsible for creating the "Named Patterns" list and associated fields. Doing so would give Figure 65. Tracking what code gets called when nodes in the "Named Patterns" list nodes are expanded, results in milestone 5 and Figure 66 – implying that the `grayOut()` method needs to be called by the selection listeners as shown in Figure 67 and part of milestone 6.

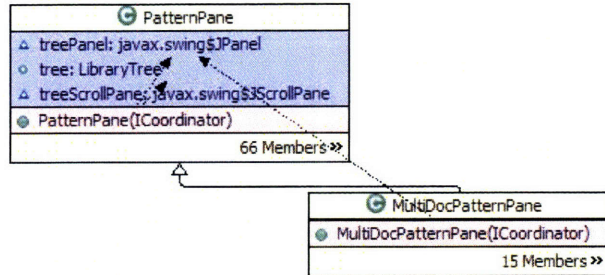


Figure 65 – Milestone 4 with Lapis using Relo Searching for the creator of the Named Patterns list in Lapis.

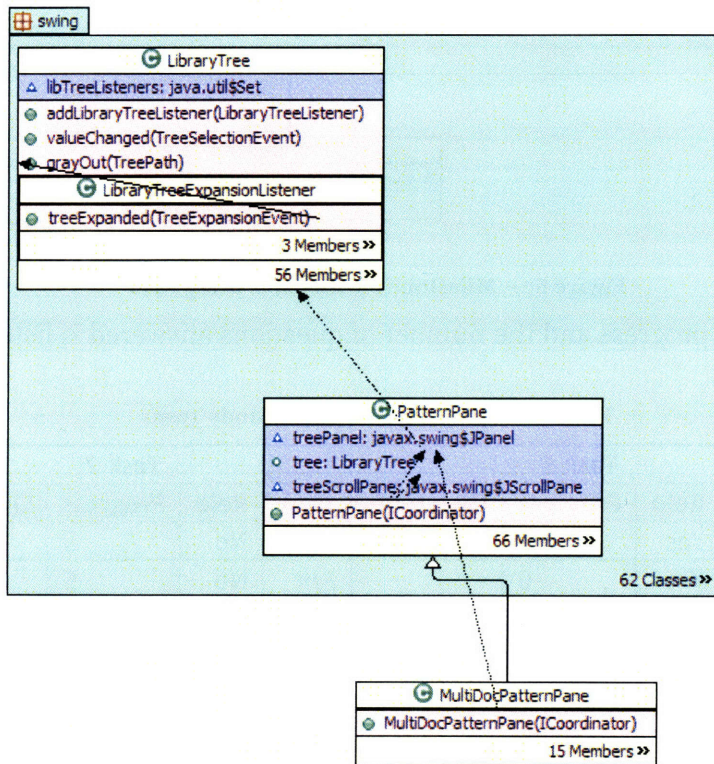


Figure 66 – Milestone 5 with Lapis using Relo

7. EVALUATIONS

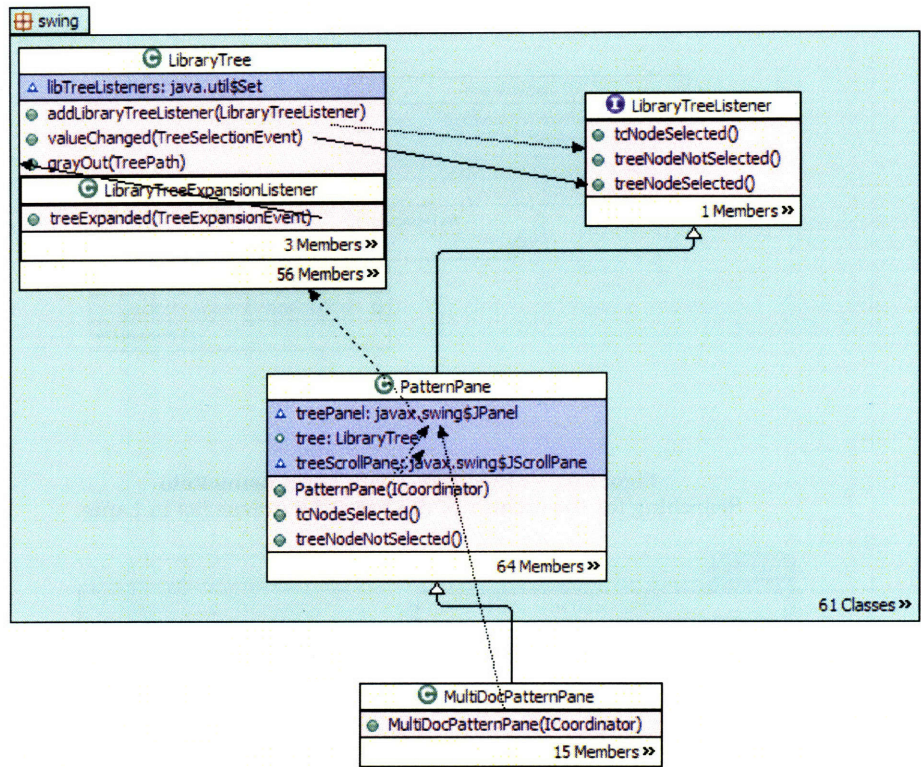


Figure 67 – Milestone 6 with Lapis using Relo

Each users progress and the number of questions answered is listed below in Table 9:

Table 9: Users performance on Study Tasks

User	Task 1				Task 2			
	Proj	Relo	Progress	Questions	Proj	Relo	Progress	Questions
1	Ant	Yes	5		Lapis	No	5	
2	Lapis	Yes	6		Ant	No	5	
3	Ant	No	8		Lapis	Yes	8	3
4	Lapis	No	6	0	Ant	Yes	5	0
5	Ant	Yes	5	4	Lapis	No	8	7
6	Ant	No	4		Lapis	Yes	5	
7	Lapis	Yes	4		Ant	No	7	5
8	Ant	No	5	4	Lapis	Yes	8	5
9	Lapis	No	6	1	Ant	Yes	6	2
10	Ant	Yes	5	6	Lapis	No	4	2
11	Lapis	No	7	4	Ant	Yes	7	4
12	Lapis	Yes	8	4	Ant	No	5	3

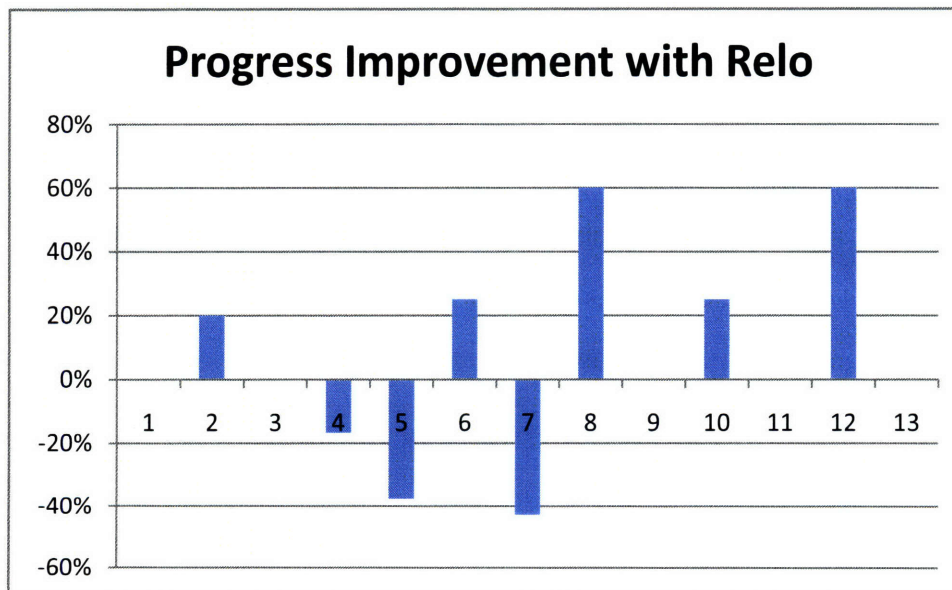
User	Task 1				Task 2			
	Proj	Relo	Progress	Questions	Proj	Relo	Progress	Questions
13	Ant	Yes	6	6	Lapis	No	6	7

From the table it can be seen that on average users did make more progress (+8%) and answered more questions correctly (+43%) when using Relo. However, in both cases there is a large amount of noise and the numbers are not statistically significant. The three possible hypothesis that could be in this data are:

- I. Improvement because of using Relo.
- II. One task is harder than the other.
- III. There is a significant learning effect.

Analyzing the data shows that for Hypothesis I is supported by 5 user's data and contradicted by 3 user's data, i.e. 3 users performed worse when they used Relo. For Hypothesis II the Ant task being harder than Lapis is supported by 6 user's data and contradicted by 2 user's data. Hypothesis III is supported by 4 user's data and contradicted by 4 user's data. Thus, the Ant task is likely harder than the Lapis task. Getting the average performance different for Ant users also shows this to be true, but again the values are not significant.

Plotting the progress improvement (Figure 68) we see that of the three users doing worse with Relo, in two of the cases (4 & 5) the users where using Relo while doing the harder Ant task. Collecting more data should allow us to quantify the amount of the difficulty of the ant task, and help in reducing noise for the Relo improvements.



7. EVALUATIONS

Figure 68 – Users progress improvement with Relo.

In further examining Relo's effect on progress improvements, we can look at the actions done by users during the study, by plotting the number of items automatically added to Relo, their number of selections of items in Relo, and the number of moves of nodes in Relo. Figure 69 and Figure 70 show these plots for user 5 (who did worse with Relo) and user 8 (who did better with Relo) respectively.

Examining user 5's actions it can be seen that while he did continue exploring in the IDE and therefore the auto-adds keep increasing, he just tried looking at and using Relo at two times during the study – once near the beginning and once near the end. Thus, Relo likely did not slow his progress during the study. Looking at user 8's actions it can be seen that his selections in Relo keep gradually increasing and he therefore did use Relo throughout the study. It is also worth noting that the number of times that user 8 moved a node is much less than the number of selections he made in Relo, i.e. Relo's layout engine was effective in placing nodes.

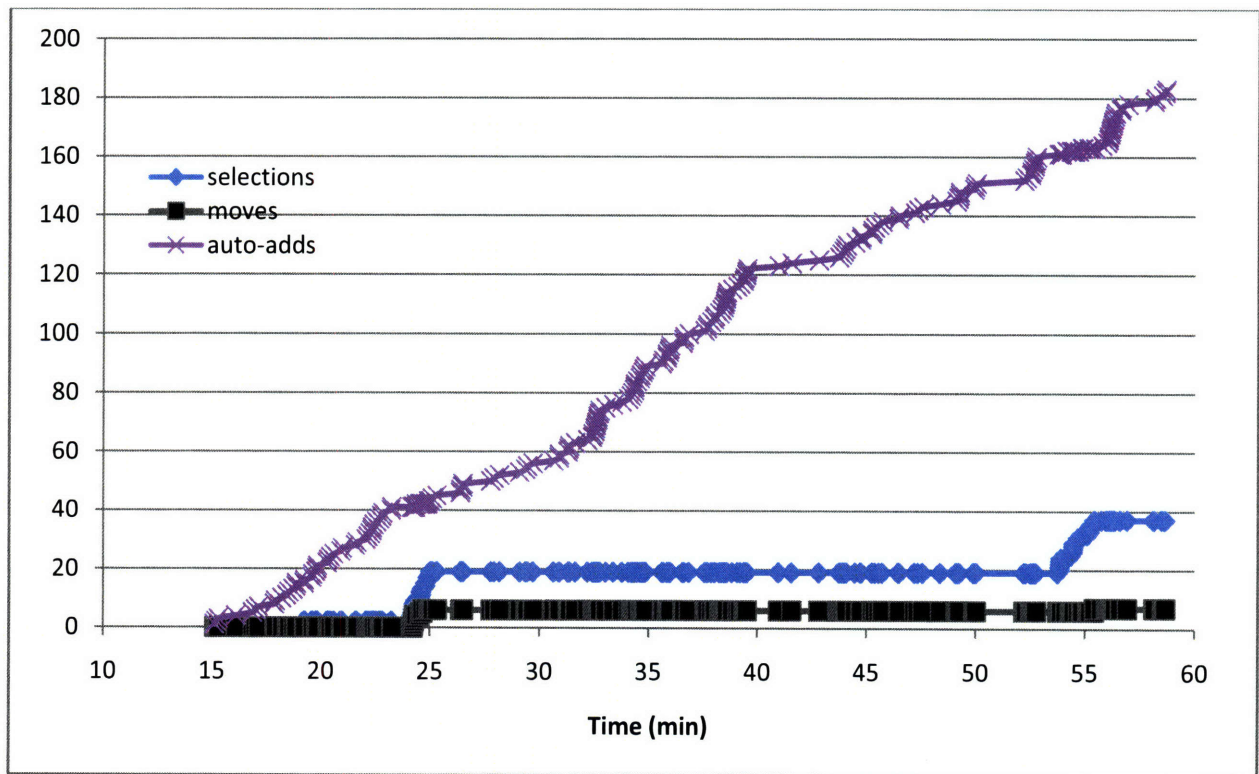


Figure 69 – User 5's actions during the study.

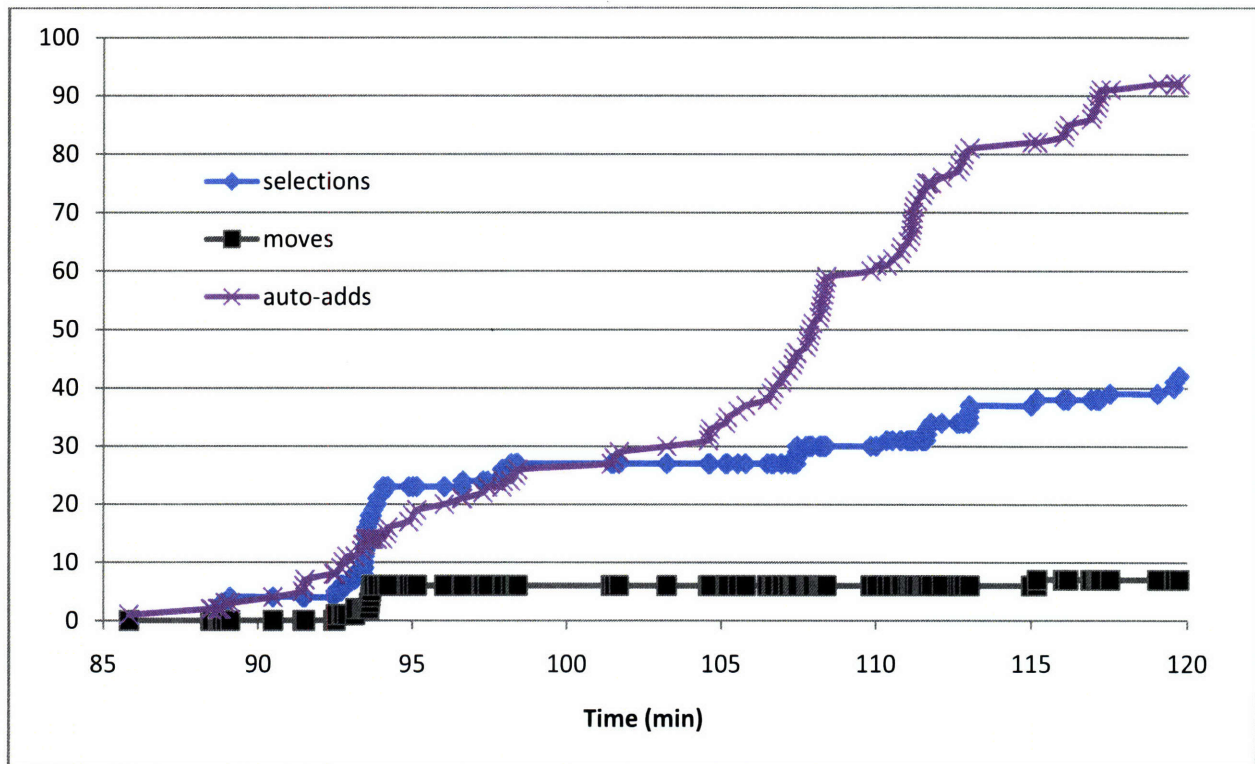


Figure 70 – User 8’s actions during the study.

During the interview, all participants admitted to starting each task in the Eclipse Java Tooling (JDT) because that was what they were used to. While we had expected a bias towards the familiar tools and had encouraged participants to use Relo, the size of the codebase and tasks made them initially ignore the tool. However, as each task progressed, the study participants mostly drifted towards Relo. This usage of Relo would happen as the complexity of the task increased: Relo would first be used as a contextual map, and then the participants would work directly with it. Even though minor bugs in Relo (with layout) would sometimes drive developers back to the JDT, they would keep drifting back into Relo to mitigate the task complexity.

One obstacle to using Relo was in dealing with dynamic/runtime code structure. For one task, a participant used exception traces to detect bugs to get runtime information and then tried to understand the relationships in the code. However, since Relo currently only infers static relations from the code, it was not able to draw relationships between such method calls. Related to this is the representation of control flow in the visualization which was a larger part of the Ant task. Relo diagrams are closer to UML class diagrams than other interaction diagrams like sequence diagrams. In some cases, this results in Relo visualizations not being very helpful.

The most interesting result of the study was the understanding of the cases in which Relo was most helpful to developers. In large codebases since developers use an as-needed/opportunistic approach they often take approaches to the task that only result in dealing with 2-3 classes/methods. In such cases, they would find navigating with Relo to be more of a hindrance since mouse-based navigations do not have as many shortcuts available as keyboard based navigations. However, in cases where more than 3 code artifacts are interacting, participants found Relo very useful.

There were a number of challenges to getting good results. While inexperienced programmers would either aimlessly make progress or would take a breadth-first strategy in understanding the code, more experienced programmers typically take a more opportunistic strategy in understanding the code. The problem is that taking this approach requires the developer to situate in the workspace, which for a large project can take time. Another challenge is that some experienced programmers don't necessarily have Eclipse experience and end up spending time getting comfortable with the IDE. Further providing the study framework does reduce the stability of Eclipse and sometimes unexpectedly causes some of the Eclipse base functionality to stop working. Similarly Eclipse has the debugging functionality as part of a separate perspective, switching perspectives causes Relo to be unexpectedly hidden.

7.4. FEEDBACK FROM THE FIELD WITH RELO

Relo has received a positive response from the Eclipse plugin community. The plugin has been rated as the top Eclipse Plugin in the UML category with one of the largest number of community votes (139 people have voted an average rating of 8.5/10¹⁵). Users have mostly appreciated the abilities to only show parts of the code in the diagram and the ease to explore and expand a diagram. Users' comments also indicate fairly sophisticated usage of Relo, for example, them trying to explore the codebase and building up a diagram, then saving it, and opening it later on to continue exploration.

All negative comments from the field have been for fixing bugs. The single largest has been in dealing with the various platform configurations for deploying the plugin, i.e. in supporting different versions of Java VM, different versions of Java in the users project, in supporting different versions of Eclipse, and in supporting different plugin combinations that might be available.

Users have provided helpful comments such as:

¹⁵ As of October 12th: http://www.eclipseplugincentral.com/Web_Links-index-req-viewcatlink-cid-19-orderby-rating.html

“This is cool!! Last year I was exploring JHotDraw and it was a lot of work to grasp the essential principles (even with all the available papers and pattern descriptions). Your tool helps a lot. Go on like that. I am looking forward to the first major release (with less bugs).”

8. CONCLUSIONS

This concluding chapter summarizes the contributions of the dissertation, and presents both questions directly raised by this work and future directions of taking this approach.

8.1. SUMMARY OF CONTRIBUTIONS

For developers, understanding code is both a large and an important component of their work. Given the complexity and size of the underlying codebase, developers can easily get overwhelmed using tools in assisting the comprehension process. By building Relo and Strata this thesis demonstrates and evaluates a number of ideas to help in code understanding tools:

- The use of diagrams familiar to developers, so that developers can understand code without additional training and can benefit from relationships emphasized by the diagram. In particular, the building of tools to support the usage of UML class diagrams and layered architecture diagrams. And further, the exploration of implementation issues in using building such tools, such as providing effective layout support.
- The use of an interactive diagrammatic exploration environment for developers to select and find code elements relevant to their task.
- The tracking, use of, and linking of developers' explorations in the IDE editors, to assist in comprehension while the developer focuses on non-comprehension aspects of the task.

Beyond the ideas built into Relo and Strata, this thesis provides a number of related contributions. It consists of a survey of developers' experiences on the effectiveness of various documentation techniques when used for program

comprehension. Further this thesis contains qualitative and quantitative feedback from both controlled lab studies and field deployments of Relo and Strata.

8.2. RAISED QUESTIONS

Building Relo and Strata has raised a number of questions, many of which have been examined in this dissertation. However, an in-depth examination of some of the below will be helpful in improving the tools quality

- **Improved Layout Engines:** While both Relo and Strata have had multiple iteration in the development of their layout engine, there is room for further improving the engine. Simply put, providing interactive layout support for diagrams with visual constraints has a number of challenges. Not only is there a need for an algorithm that provides a good guess for code semantics that the algorithm doesn't really understand, but such an algorithm also needs to deal with decayed code, and further provide support for users interaction with the diagram.
- **Improved Interactivity:** Both Relo and Strata have support for basic direct interaction with the shown code elements, to either show more code element, to zoom into details of the shown code, or for the removal of the shown code elements. In fact, support has been provided for complex queries such as shown with autobrowse. However, such provided support include very simple graph queries, and presents an opportunity for an intelligent algorithm which users team knowledge towards quickly building relevant and useful diagrams.

8.3. LOOKING AHEAD

This work highlights the need for improved tool support for program comprehension by supporting interactive exploration of code in familiar diagrams. Further help can be provided in understanding code or other non-software domains by providing support for some of the below.

8.3.1 Design Patterns

Experienced developers have a good understanding of design patterns. They are often able to use them effectively in understanding parts of codebases by recognizing the patterns in the underlying codebase, i.e. in a top-down manner. While design patterns seem to form an essential part of understanding code using them in building effective program comprehension tools has significant obstacles.

A key challenge has been the effective recognition of design patterns. Design patterns are high-level concepts and have a great degree of variability in their instantiations. To take into account such variability design pattern detection

engines can be tried to be implemented on a fuzzy scale. Additionally, opportunities exist for interfaces for a programmer to interactively use a detection engine for easing the detection of design patterns.

The other challenge in providing support for design patterns is that it is not clear as to the form of an appropriate visualization of the design patterns. In particular, members of design patterns play different roles inside the pattern – it is not clear as to how these roles should be visualized, especially in cases where a member performs more than one role as in different patterns.

8.3.2 Program Knowledge Extraction

Various types of annotations can be extracted automatically from a system for use in helping developers explore code. These approaches do however have a number of challenges in successfully adopting them.

Static program analyses build a model of the program which is then simulated to extract one of a possibly large number of useful properties from code. Effective usage of such analyses for program comprehension, however, requires the appropriate selection and definition of extractable properties that are important for understanding code. Additionally, such analyses can take significant computation time that would affect the usefulness of such techniques. Such analyses can be made practical by splitting them into multiple parallelizable chunks and caching results to perform the analyses only on changed portions of the codebase.

Runtime analyses can also be helpful for program comprehension. These involve analyses based on values taken up by variables at runtime. Again, leveraging this approach requires dealing with the involved performance overhead. In particular, the overhead of logging data during the traced program's run can be significant. Other challenges include providing support for logging in distributed environments and dealing with interactive application.

Finally, program knowledge can be extracted using heuristics. While heuristics cannot make guarantees on the detection of any particular property, they have been shown to be effective in program comprehension tasks [41]. Simple approaches such as using TFIDF on tokenized identifiers in the code can be used to provide good starting points for developers exploring a codebase.

8.3.3 Different Diagram Types

While Relo supports exploration in UML like class-diagrams and Strata provides support for layered architectural diagrams, support for other popular diagram types can potentially help developers with diagrams emphasizing other properties of the codebase. For example, Message Sequence Diagrams emphasize interactions between objects. Supporting such diagrams have challenges in their having a stronger demand for runtime data and also in some of these diagrams taking large space to show the important properties.

8. CONCLUSIONS

Beyond providing support for different diagram types, there is also an opportunity for the usage of colors and sizes to emphasize different properties of components like recency of edits or browsing, and amount of edits or reuse in the code base.

Additionally with the presence of these different diagrams is the need for consideration between switching from one diagram type to another, and the support for having more than one diagram type be shown in a single visualization. These can become a challenge especially in the cases when the definition of familiar for one diagram type conflicts with those of another.

8.3.4 Collaboration and Communication

While Relo and Strata are helpful for developers understanding code, they can be used by an expert programmer to store annotations about the system for use by a new programmer later on. These annotations can be of many types. Bookmarking support for diagrams can implicitly define the implementation of a feature. Diagrams can be enforced as build rules and can be injected into views inside the text editor. Comments in diagrams can be used for suggesting more code elements of interest to a user. More structured annotations such as numbers can show control or data flow in a diagram for future viewers of a diagram, and powerful annotations can even be used by a design pattern engine to capture and leverage design patterns used in a project.

8.3.5 Support for non-software domains

Relo and Strata have been built on a general framework, which can be expanded to other domains. Information-rich domains have both significant amounts of structured data readily available and common graphical representations to be used to show the data. Beyond expanding the underlying framework for diagrammatic exploration in other domains, support can be provided for working out-of-the-box using graph style sheets [59] as a basis for the used diagrams.

REFERENCES

- [1] Apache Ant, ver 1.6.5, <http://ant.apache.org>, 2006.
- [2] A. Bien, "UML Reverse Engineering ... not so important in real world projects?", http://www.adam-bien.com/roller/page/abien?entry=uml_reverse_engineering_not_so, 7th Aug 2007.
- [3] ComputerWorld, "Waiting for UML 2.0" interview with Grady Booch and Bran Selic, Mar 2004, <http://www.computerworld.com/developmenttopics/development/story/0,10801,91325,00.html>, 2007
- [4] Fujaba Tool Suite, <http://wwwcs.uni-paderborn.de/cs/fujaba/>
- [5] jEdit, ver 4.3, <http://jedit.org>, 2006.
- [6] JHotDraw. <http://www.jhotdraw.org/>
- [7] The LAPIS Project, User Interface Design Group, MIT Computer Science and Artificial Intelligence Laboratory, ver 1.2., <http://groups.csail.mit.edu/uid/lapis/>, 2007
- [8] Longwell – Simile, ver 2.0, <http://simile.mit.edu/longwell/>, 2007
- [9] Rational Rose, IBM, <http://www.ibm.com/software/rational/>
- [10] Together Technologies, Borland, <http://www.borland.com/together/>
- [11] S. Bassil, and R.K. Keller, "Software visualization tools: survey and analysis". IWPC 2001.
- [12] M.J. Bates, "The design of browsing and berrypicking techniques for the on-line search interface". Online Review, 13(5), October 1989, pp. 407-424.

REFERENCES

- [13] L. O'Brien and C. Stoermer, "Developerure Reconstruction Case Study", Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-2003-TN-008
- [14] R. Brooks, "Towards a theory of the comprehension of computer programs" *International Journal of Man-Machine Studies*, 1983
- [15] Browning, T. "Applying the Design Structure Matrix to System Decomposition and Integration problems: A Review and New Directions". *IEEE Transactions on Engineering management*, Vol. 48, No. 3, August 2001.
- [16] S. K. Card and D. Nation. "Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface". *Advanced Visual Interfaces Conference*, 2002.
- [17] M. Cherubini, G. Venolia, and R. DeLine. "Building an Ecologically-valid, Large-scale Diagram to Help Developers Stay Oriented in Their Code". In *VL/HCC 2007*.
- [18] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. "Let's go to the whiteboard: how and why software developers use drawings". In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (San Jose, California, USA, April 28 - May 03, 2007)*. CHI '07. ACM Press, New York, NY, 557-566.
- [19] P. Clements, F. Bachmann, and L. Bass. "Documenting Software Developers – Views and Beyond", Addison-Wesley, 2003.
- [20] T. A. Corbi. "Program understanding: Challenge for the 1990s". *IBM Systems Journal* 28 (2), pp. 294–306, 1989.
- [21] D. Čubranić and G. C. Murphy. "Hipikat: recommending pertinent software development artifacts," *ICSE 2003*
- [22] J. Davison, D. Mancl, and W. F. Opdyke, "Understanding and Addressing the Essential Costs of Evolving Systems," *Bell Labs Technical Journal*, 5(2), April-June 2000, pp. 44--54.
- [23] F. D'etienne. "Software Design—Cognitive Aspects". *Springer Practitioner Series*. 2001
- [24] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. "LaSSIE: a knowledge-based software information system", *ICSE 1990*
- [25] R. DeLine, "Staying oriented with Software Terrain Maps", In *Proceedings of the Workshop on Visual Languages and Computation*, Sept 2005.
- [26] R. DeLine, A. Khella, M. Czerwinski, G. Robertson, "Towards understanding programs through wear-based filtering". *ACM SoftVis 2005*.
- [27] K. Doan, C. Plaisant, and B. Shneiderman, "Query Previews in Networked Information Systems". *IEEE ADL 1996*.

- [28] B. Dobing, and J. Parsons, "How UML is used". *Communications of the ACM*, 49(5), May 2006, pp. 109-113.
- [29] S. G. Eick, J. L. Steffen, and E. E. Summner Jr. "Seesoft – a tool for visualizing line oriented software statistics" *IEEE Trans. On Software. Engineering* 18(11), pp957-968.
- [30] D. A. Gebala and S. D. Eppinger. "Methods for Analyzing Design Procedures", *Proceedings of the ASME Third International Conference on Design Theory and Methodology*, pp. 227-233, 1991.
- [31] N. Gershon and S.G. Eick, "Guest Editors' Introduction: Scaling to New Heights", *IEEE Comp. Graphics and Applications*, 18(4):16-17, 1998.
- [32] I. Gorton and L. Zhu. "Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: An Experience Report", *Proceedings of the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, 15-21 May, 2005 (ICSE'05)*.
- [33] J. Gosling, B. Joy, G. Steele, and G. Bracha. "The Java™ Language Specification Third Edition", Addison-Wesley, 2005.
- [34] J. Guo, F. Hüffner, and H. Moser, "Feedback arc set in bipartite tournaments is NP-complete". *Information Processing Letters*, 102(2-3), pp 62-65, Apr. 2007.
- [35] W. Harrison and H. Ossher. "Subject-oriented programming (a critique of pure objects)." *OOPSLA 1993*.
- [36] W. Harrison, H. Ossher, S. M. Sutton Jr., and Peri Tarr, "Concern Modeling in the Concern Manipulation Environment." *IBM Research Report RC23344*, September 2004.
- [37] I. Herman, G. Melancon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey", *IEEE Trans. on Visualization and Computer Graphics*, 6(1):24-43, 2000.
- [38] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples", *ICSE'05*.
- [39] S. Hupfer, L. -T. Cheng, S. Ross, and J. Patterson. "Introducing collaboration into an application development environment" In *ACM 2004 CSCW*, 2004, pp 444-454.
- [40] D. Janzen and K. D. Volder. "Navigating and Querying Code Without Getting Lost", *AOSD 2003*.
- [41] D. Jackson and A. Waingold. "Lightweight extraction of object models from bytecode", *ICSE 1999*
- [42] S. Jul, and G.W. Furnas, "Navigation in Electronic Worlds: Workshop Report", *SIGCHI Bulletin*, 29(4), October 1997, pp. 44-49.

REFERENCES

- [43] R. Kazman and S. Jeromy Carrier, "View Extraction and View Fusion in Developerural Understanding", In Proceedings of the 5th International Conference on Software Reuse, 1998.
- [44] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs", AOSD 2005
- [45] A. J. Ko, H. Aung, and B. A. Myers. "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks". ICSE 2004.
- [46] P. Kruchten "The 4+1 View Model of Developerure". IEEE Software. 12, 6 (Nov. 1995)
- [47] P. Krutchen, R. Hilliard, R. Kazman, W. Kozaczynski, H. Obbink, and A. Ran. The Software Developerure Review and Assessment (SARA) Report, 2002.
- [48] O. Lassila and R. Swick, "Resource description framework (RDF): Model and syntax specification", <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222>, February 1999. W3C Recommendation.
- [49] T. D. LaToza, G. Venolia, and R. DeLine. "Maintaining mental models: a study of developer work habits". In Proceeding of the 28th international Conference on Software Engineering (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM Press, New York, NY, 492-501.
- [50] S. Letovsky. "Cognitive processes in program comprehension", In Proceedings of Empirical Studies of Programmers, pp. 58-79. 1986.
- [51] M. Levesque. "Fundamental issues with open source software development", First Monday , 9(4) April 2004, http://www.firstmonday.org/issues/issue9_4/levesque/
- [52] ? - G. Lindgaard. "Usability Testing and System Evaluation: A Guide for Designing Useful Computer Systems", 1994, Chapman and Hall, London, U.K. ISBN 0-412-46100-5
- [53] G. Little, and R.C. Miller. "Keyword Command Completion in Java", Proceedings of the 22th IEEE/ACM international Conference on Automated software engineering, ASE '07
- [54] D. C. Littman, J. Pinto, S. Letovsky, and Soloway E. "Mental models and software maintenance", In Empirical Studies of Programmers, Washington, DC, pp. 80-98, 1986
- [55] I. Majid and M. P. Robillard. "NaCIN - An Eclipse Plug-In for Program Navigation-based Concern Inference". Proceedings of the Eclipse Technology Exchange at OOPSLA, October 2005.
- [56] G. Marchionini, "Information seeking in electronic environments", 1995, Cambridge University Press.

- [57] G. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *Psychological Review*, 1956, vol. 63 pp. 81-97
- [58] G. C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study", *IEEE Software*, 1997.
- [59] E. Pietriga, "Graph Stylesheets (GSS) in IsaViz".
<http://www.w3.org/2001/11/IsaViz/gss/gssmanual.html>, November 2001
- [60] D. Perry, N. Staudenmayer, and L. G. Votta. "People, Organizations, and Process Improvement". *IEEE Software*, 11(4), 36-45, 1994
- [61] P. Pirolli, and S. Card, "Information foraging in information access environments", *Conference proceedings on Human factors in computing systems*, 1995, pp. 51-58.
- [62] P. O'Shea and C. Exton, "An Investigation of Java Abstraction Usage for Program Modifications." *IWPC 2005*.
- [63] J. Nielsen, "Usability Engineering", pp 195-198, Academic Press, 1993.
- [64] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs". *Cognitive Psychology*, 19:295-341, 1987.
- [65] S. Reiss. "Visualization for Software Engineering – Programming Environments", Chapter 18, pages 259-276, in "Software Visualization", ed. Stasko et al.
- [66] Robillard, M.P.; Coelho, W. and Murphy, G.C. "How effective developers investigate source code: an exploratory study", *IEEE Transactions on Software Engineering*, 30(12), Dec. 2004, Pages: 889- 903
- [67] M. P. Robillard and G. C. Murphy. "Automatically Inferring Concern Code from Program Investigation Activities". In *ASE 2003*.
- [68] M. P. Robillard and G. C. Murphy. "Concern graphs: finding and describing concerns using structural program dependencies", In *ICSE 2002*.
- [69] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang, "Using Multiple Views to Model and Analyze Software Developerure: An Experience Report", University of Southern California, Center for Software Engineering, Technical Report USC-CSE-2003-508
- [70] G. Sander. "Layout of compound directed graphs". Technical Report A/03/96, Universit at des Saarlandes, June 1996.
- [71] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. "Using Dependency Models to Manage Complex Software Developerure". 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems (OOPSLA 2005).

REFERENCES

- [72] M. Shaw and D. Garlan. "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall, 1996.
- [73] B. Shneiderman and R. Mayer. "Syntactic/semantic interactions in programmer behavior: A model and experimental results". *International Journal of Computer and Information Sciences*, 8(3): 219–238.
- [74] B. Shneiderman, "Software Psychology: Human Factors in Computer and Information Systems." Winthrop Publishers Inc., 1980.
- [75] S. E. Sim and R. C. Holt. "The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize", ICSE 1998
- [76] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. "An Examination of Software Engineering Work Practices" In *Proceedings of CASCON '97*, 209–223, 1997.
- [77] K. Sugiyama and K. Misue. *Visualization of Structural Information: Automatic Drawing of Compound Digraphs*. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):867--892, July 1991.
- [78] J. Sillito, G.C. Murphy, and K.D. Volder. "Questions Programmers Ask During Software Evolution Tasks", SIGSOFT'06, November 5–11, 2006.
- [79] V. Sinha, R. Miller, and D. R. Karger. "Incremental Exploratory Visualization of Relationships in Large Codebases for Program Comprehension", Poster, OOPSLA 2005.
- [80] V. Sinha, R. Miller, and D. R. Karger. "Incremental Exploratory Visualization of Relationships in Large Codebases for Program Comprehension", Demonstration, OOPSLA 2005.
- [81] V. Sinha, R. Miller, and D. R. Karger. "Relo: Helping users manage context during interactive exploratory visualization of large codebases", ETX 2005.
- [82] V. Sinha, D. R. Karger, and R. Miller, "Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases". VL/HCC 2006.
- [83] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge", *IEEE Transactions on Software Engineering*, 10(5), pp. 595-609, September, 1984.
- [84] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, "Designing documentation to compensate for delocalized plans". *CACM*, 31(11):1259-1267, 1988.
- [85] M.-A. Storey, H. Muller, and K. Wong, "Manipulating and documenting software structures using SHriMP views", ICSM 1995.
- [86] M.-A. Storey, H. Muller, and K. Wong, "How Do Program Understanding Tools Affect How Programmers Understand Programs?", WCRE 1997.

- [87] M.-A. Storey, F. Fracchia, and H. Muller. "Cognitive design elements to support the construction of a mental model during software visualization". IWPC 1997.
- [88] M.-A. Storey. "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future", *Software Quality Journal*, Springer, 2006.
- [89] S. M. Sutton, and I. Rouvellou. "Modeling of software concerns in Cosmos". AOSD 2002.
- [90] C. Szyperski, "Component Software - Beyond Object-Oriented Programming", ACM Press/Addison- Wesley, 1997.
- [91] P. Tarr, H. Ossher, W. Harrison, and Stanley M. Sutton, Jr., "N Degrees of Separation: Multi-dimensional Separation of Concerns." ICSE 1999.
- [92] J. Teevan, C. Alvarado, M. S. Ackerman, and D. R. Karger. "The perfect search engine is not enough: a study of orienteering behavior in directed search". CHI 2004.
- [93] Q. Teng, X. Chen, X. Zhao, W. Zhu and L. Zhang, "Extraction and Visualization of Architectural Structure Based on Cross References among Object Files". COMPSAC 2004.
- [94] D. Tunkelang. "A Numerical Optimization Approach to General Graph Drawing", Ph.D. Thesis, Carnegie Mellon University, 1999.
- [95] A. von Mayrhauser and A.M. Vans. "From code understanding needs to reverse engineering tool capabilities". In Proceedings of CASE'93, pp. 230-239.
- [96] L. Wang. "Animated Exploring of Huge Software Systems", MS Thesis, School of Info. Tech. and Engg., University of Ottawa, 2002.
- [97] J. N. Warfield. "Binary Matrices in System Modeling" IEEE Transactions on Systems, Man, and Cybernetics, vol. 3, pp. 441-449, 1973.