Helping Developers Understand Code via Diagrammatic Explorations

Thesis, PhD Computer Science

Vineet Sinha, vineet@csail.mit.edu
32 Vassar Street, 32-G598
Cambridge, MA - 02139

Computer Science and Artificial Intelligence Laboratory (CSAIL)
Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology, Cambridge, Massachusetts

COMMITTEE

- [supervisor] Prof. David Karger <karger@mit.edu>
- [supervisor] Prof. Rob Miller <rcm@mit.edu>
- [reader] Prof. Daniel Jackson <dnj@mit.edu>
- [reader] Prof. Howie Shrobe <hes@csail.mit.edu>

File: thesis_022_tx.doc
Helping Developers Understand Code via Diagrammatic Explorations

ABSTRACT

As software systems grow in size and use more third-party libraries and frameworks, the need for developers to understand unfamiliar large code-bases is rapidly increasing. In this thesis I present two tools, Relo and Strata, which support developers’ understanding by creating partial diagrammatic representations of the explored code. These tools help developers explore code and select the important subsets of code artifacts and relationships to display using various visual constraints to ease the comprehension of the targeted code. As the developer explores relationships found in the code, Relo and Strata track and use the explorations to build and automatically manage a diagrammatic representation of the developer’s mental model. While Relo focuses on creating diagrams showing distinct relationships similar to commonly created class diagrams, Strata focuses on diagrams showing aggregated modules and dependencies similar to layered architectural diagrams. This thesis shows that developers’ understanding of code can be supported by a tool that creates visually constrained partial diagrammatic representations of interactively explored code.
CONTENTS

Contents .................................................................................................................... 5
Figures ....................................................................................................................... 7
Tables ....................................................................................................................... 9
1. Introduction ....................................................................................................... 11
   1.1. The Problem .............................................................................................. 11
   1.2. Program Comprehension .......................................................................... 13
   1.3. The Approach ............................................................................................ 13
   1.4. Walkthrough .............................................................................................. 19
       1.4.1 Layered Overviews with Strata............................................................. 19
       1.4.2 Detailed Relationships with Relo........................................................ 22
   1.5. Scenarios ................................................................................................... 26
   1.6. Contributions ............................................................................................. 27
   1.7. Outline ...................................................................................................... 28
2. Previous Work ................................................................................................... 29
   2.1. Developer Behavior .................................................................................. 29
       2.1.1 Understanding Models ........................................................................ 30
       2.1.2 Usage of Sketching ............................................................................... 31
       2.1.3 Working with Concerns ....................................................................... 32
   2.2. Overview Tools ........................................................................................... 33
   2.3. Exploration Tools ..................................................................................... 40
       2.3.1 Tree based approaches ......................................................................... 41
       2.3.2 Visualization based approaches .......................................................... 42
       2.3.3 Discussion ............................................................................................ 45
   2.4. Tools for Large Projects ............................................................................ 46
   2.5. Design Tools ............................................................................................. 48
3. Survey of Software Immigrants ........................................................................ 50
   3.1. Method ...................................................................................................... 50
   3.2. Results ....................................................................................................... 53
   3.3. Task Analysis ............................................................................................. 55
4. Strata User Interface ......................................................................................... 59
   4.1. Appearance ............................................................................................... 60
   4.2. Layout ....................................................................................................... 62
   4.3. Navigation ................................................................................................. 65
       4.3.1 Interface Behavior for Navigating....................................................... 65
       4.3.2 Exploration via Navigation Buds ........................................................ 66
   4.4. Updating Module Definitions ................................................................... 67
FIGURES

Figure 1 – Typical class diagrams ................................................................. 15
Figure 2 – Typical layered architectures diagrams ......................................... 16
Figure 3 – Top level view when a developer opens Strata on the entire JEdit codebase ................................................................. 19
Figure 4 – Strata after the developer removes the modules other than 'org' ....... 21
Figure 5 – Relo started by opening EllipseFigure ......................................... 23
Figure 6 – After adding the method basicMoveBy (by using the more items menu) ................................................................. 24
Figure 7 – Clicking on the class to show its handles ....................................... 24
Figure 8 – After clicking on the inheritance handles ....................................... 24
Figure 9 – Expanding the class AbstractFigure and the method addFigureChangeListener ................................................................. 25
Figure 10 – Asking for callers of addFigureChangeListener .......................... 26
Figure 11 – Visualization of Software Terrain Maps ....................................... 35
Figure 12 – The SeeSoft Visualization ............................................................ 36
Figure 13 – Armin view on loading dependencies ........................................... 37
Figure 14 – Armin view after running a short script ......................................... 37
Figure 15 – Layered layout provided by Deref .................................................. 38
Figure 16 – LDM on an old version of the Ant project ...................................... 39
Figure 17 – A layered view by LDM of the ant project ..................................... 40
Figure 18 – Results of the exploration session shown in the previous section with the Eclipse IDE ................................................................. 41
Figure 19 – Results of the exploration session shown in the previous sections with JQuery .................................................................................. 42
Figure 20 – The call graph viewer of Field ....................................................... 43
Figure 21 – Class hierarchy browser of Field ................................................... 43
Figure 22 – Part of an exploration session with SHriMP ................................. 44
Figure 23 – Part of an exploration session with the TkSee Visualizer ............... 45
Figure 24 – Screenshot of Mylar ..................................................................... 47
Figure 25 – Screenshot of Jazz [39] ................................................................. 48
Figure 26 – Survey Questions ........................................................................ 53
Figure 27 – A Layered Architectural Diagram ............................................... 60
Figure 28 – Strata display of the jEdit project .................................................. 61
Figure 29 – Automatically building a layered diagram without dealing with cycles (for the jEdit project) ......................................................... 63
Figure 30 – A depiction of the steps of the layering algorithm on the ant project. .......................................................... 64
Figure 31 – Results of the layering algorithm. .......................................................... 64
Figure 32 – Merging the results of the generated layers. ........................................ 65
Figure 33 – Breaking a module in Strata. .............................................................. 66
Figure 34 – Navigation Buds in Strata. ................................................................. 66
Figure 35 – Expanding dependencies in Strata. .................................................. 67
Figure 36 – Relo showing part of the JHotDraw project. (same as Figure 8) ... 70
Figure 37 – Basic Relo Architecture. ................................................................. 79
Figure 38 – Architecture with caching support for Relo. ..................................... 80
Figure 39 – Architecture with support for Strata. .............................................. 81
Figure 40 – Diagram width with and without cycle breaking compared to ideal ... 86
Figure 41 – RSSOwl with cycle breaking .......................................................... 86
Figure 42 – Traditional partitioning with RSSOwl ........................................... 87
Figure 43 – One of the higher-level views of System “A”. .................................. 89
Figure 44 – Business domain view of System “A” using Strata. ......................... 90
Figure 45 – Background for the Ant task. .......................................................... 92
Figure 46 – The actual Ant bug. .................................................................... 93
Figure 47 – The test case for the Ant task. ........................................................ 94
Figure 48 – Hints provided for the Ant bug. ....................................................... 94
Figure 49 – The Lapis task. ............................................................................ 95
Figure 50 – Hints for the Lapis task. ............................................................... 95
Figure 51 – Questions Programmers Ask During Software Evolution Tasks – Part I. ................................................................. 96
Figure 52 – Questions Programmers Ask During Software Evolution Tasks – Part II. ................................................................. 97
Figure 53 – Users progress improvement with Relo. ....................................... 100
### TABLES

Table 1: Difficulty of understanding the project ................................................... 53  
Table 2: Users saying documentation technique was used effectively to assist in understanding project ................................................................. 53  
Table 3: Comparing current and wanted techniques for all survey participants. 54  
Table 4: For developers examining internal code .............................................. 55  
Table 5: Tasks done by user when they were examining the code ...................... 56  
Table 6: The Java Relationships Model ......................................................... 74  
Table 7: Projects used in determining Strata usefulness .................................. 83  
Table 8: Results from expanding the projects in Strata ................................... 85  
Table 9: Projects involved in user feedback with Strata ................................... 87  
Table 10: Users performance with Relo on Study Tasks ................................. 99
1. **INTRODUCTION**

As software systems grow in size and use more third-party libraries and frameworks, the need for developers to understand unfamiliar large code-bases is rapidly increasing. In this thesis I present two tools, Relo 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**

As software grows in size and complexity, developers face increasing difficulties in comprehending it 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, when available, the design documentation is often not comprehensive of the software system. Comprehension 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].

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. The complexity brought about by the interaction of multiple types of abstraction mechanisms introduced by modern object-oriented programming forces developers to manually keep track of their context while exploring and trying to understand the code. Developers thus need tool support to keep track of and appropriately show these different relationships.

For a given task, even when the right code elements and relationships are provided to the developer (such as from external sources), they need to be visualized in a manner appropriately emphasizing the right elements and the right relationships. Consider UML Diagrams – users create a Class Diagram to emphasize the classes, attributes, operations, and the relationships between them, while a Sequence Diagram or a Collaboration Diagram are created to emphasize the interaction between the given classes/objects. While UML diagrams were created to support communication of design issues, and available UML tools try to provide reverse engineering capabilities to generate diagrams directly from source code, the generated diagrams have too much information. Users complain that “the amount of information in resulting diagrams is overwhelming” [2] resulting in such tools have had only a 10% adoption [3]. In other words, the user needs a simpler diagram, focused on the elements and relationships relevant to the current task. This focused diagram represents one of the many architectural views needed to understand the different aspects of the project [19][46].

To provide developers with a diagram focused on their current task, this thesis proposes the use a user’s exploration to effectively select or build a diagram for the user. In a study at Microsoft Cherubini et al. [17] observed the use of diagrams created for a software system, and with the exception of new hires found them of little use with a key limitation that the created diagrams contained either too few or too many details in relation to cases when the diagrams might have been useful. Even having a set of systematically created diagrams has the limitation that the diagrams need to be adapted to the current task of the user. There is therefore a need to track the user’s progress in his task and customize any shown diagrams.

Previous tools (described in Chapter 2) attempting to help program comprehension have had scalability limitations resulting in various usability issues. The result of the scalability limitations have ranged from overwhelming users to requiring developers 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 similar results with shown similar scalability limitations, and have suggested the use of analyzing users tasks to customize visualizations to get around the scalability limitations [31][37]. This thesis therefore present Re-lo [77][78][79][80] and Strata, which support common software diagram types and allows the user’s exploration to explicitly represent the user’s current task via generated diagrams.
1.2. Program Comprehension

As described later in section 2.1, program comprehension among developers has been studied for twenty plus years. 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 their 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. With understanding often needing to be done to accomplish their task, there is an additional need for the interface to be intuitive and be tightly integrated with their development environment. Comprehension tools that require the developer to be trained or require the developer to author a script to use the tool will not be of much benefit to developers.

Program understanding studies have found it 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. Developers have been shown to build a number of different types of mental representations or mental models while understanding code. They however build these mental models mostly using an opportunistic strategy focusing only on code elements relevant to the task at hand.

1.3. The Approach

While numerous visualization tools for helping program comprehension do exist today (covered in Chapter 2), their evaluations in program comprehension tasks have shown users getting overwhelmed with the output of the tools themselves. Thus, program comprehension tools need to focus on reducing developers getting overwhelmed by both the comprehension task and using the tools themselves. We have therefore designed Strata and Relo to be built with a few important basic principles to reduce the cognitive overhead. In our discussion, diagrams consist of nodes and relationships – where nodes are boxes in the diagram and represent code elements including methods, fields, and classes, and relations are edges in the diagram and represent dependencies including method calls and inheritance.
Figure 1 – Typical class diagrams

Sources:
http://developers.sun.com/jsenterprise/learning/tutorials/jse8/uml_class_diagram.html and
http://java.freehep.org/freehep1.x/yappi/ClassDiagram.html

The first images returned by Google on searching for “Class Diagrams”
Below are the key principles underlying Strata and Relo.

1. When developers understand code, their mental model of the consisting of the involved code elements and the manner in which they interact. This model changes over time as the developers get a better understanding of both the current code and the problem that they are trying to solve [12][42][56][60]. A program comprehension interface therefore needs to support developers understanding task by managing this changing mental model.

Support for a developer’s mental model can be provided by showing using sketches commonly created by humans when doing such tasks. This thesis therefore focuses on supporting comprehension tasks that can be performed with the aid of two of the most commonly created diagrams types of code [28][18]. 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 are shown in Figure 1 and Figure 2 respectively.

2. During comprehension seeing unrelated items to the task at hand tend to overwhelm users. In order to allow users to focus on the current task diagrams need to be partial. Each diagram emphasizes a certain type of architectural view. Furthermore, it is the partialness of each diagram that implicitly emphasizes the shown relationships and

---

*a 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/aw04/papers/refereed/treloar/paper.html and
http://bartdesmet.net/blogs/bart/archive/2005/09/01/3517.aspx*
thereby a particular view of the code. Thus, a node being added to a

diagram should ideally not require all its siblings to be shown (as is

common traditionally in trees and most tools). For example, it should

be possible for a class and one of its members to be shown without the

rest of the members being visible, an example often found in human-
drawn diagrams of code [18] but not available in today’s UML

tools [9][10].

Again, in order to not overwhelm users diagrammatic changes

need to be incremental. Users should be able to have a fine-
grained control to add and remove nodes in an easy manner. This

manner needs to be presented contextually and close to the node or re-
lationship in question. Relo and Strata use navigation buds which ap-
pear when nodes are selected. Beyond nodes, users also need to access

relationships on the various methods and follow them. This allows a
developer to expand the visualization interactively, using simple con-
trols on the visualization itself. Thus developers can avoid clutter that

would otherwise hinder the comprehension activity. Prior work on

program comprehension [63][71][82][85] has shown that while devel-

opers 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. Based on this observation, Relo and Strata diagrams

allow for developers to browse relationships interactively and to add

or remove code artifacts.

3. To minimize the cognitive overhead of using the tools, the tools needs
to appear and behave as expected. The goal for each generated dia-
gram is to provide a set of defaults so that the diagram usually appears

as if it was drawn for the specific task, i.e. diagram nodes and rela-
tionships should be customized to the diagram type. For ex-

ample, inheritance in class diagrams is often shown as a relationship
directed upwards with the arrow always being closed, Relo therefore
uses these as defaults in its diagrams. In contrast current visualization
tools show all nodes in a similar manner and all relationships in a sim-
ilar manner – with the only customization 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).

Along with appearance the diagram nodes and relationships

need to also have their behavior customized to the diagram

type. Relationships can be displayed via visual constraints. Profes-
sional 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 automatically lay out each diagram, with layout rules

that try to put components in predictable places based on their rela-
tionships. 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.

For nodes, having their behavior be customized to the diagram, means certain nodes are automatically added or deleted. For example, nodes default to showing the bare minimum information and by default do not show their parent. However, when multiple siblings are added to a diagram then it is often expected for them to be shown together with their parent.

4. With developers understanding code so frequently there is a need to support understanding as the primary activity of the developer. Further, since understanding is often done secondarily to some other development task, there is also a need to support understanding as a secondary task, i.e. a need to help developers understand when they might not be explicitly thinking about understanding or directly using the understanding tool.

Supporting comprehension as a primary task typically means supporting a basic program comprehension style. A developer focusing on examining a layered architecture will often want to examine particular details of the components shown in the architecture. This behavior corresponds to those typically done during Top-Down Comprehension [14][81]. In contrast, a developer focusing on examining a class diagram will evolving his understanding of the code as he is able to process parts of the diagram. This behavior corresponds to those typically done during Bottom-up Comprehension [63][71][82][85].

On the other hand, a developer focusing on a code maintenance task and looking at some new code (for him) will benefit from a layered diagram to quickly see how the new code plays a role in the entire project and on the components that he already understands. The same developer can benefit from a class diagram view to know and remember what relationships he has been exploring. Relo and Strata thus can support usage in this secondary manner by automatically add and remove items being explored in the traditional environment. Additionally these explorations in the traditional environment are always tracked in the background. This allows the user at any point in time – when he feels lost – to ask the tool to guess and display the diagram based on the previous explorations.

Thus Relo and Strata are designed to appear like commonly made code diagrams, displaying parts of large codebases, allowing incremental user-directed changes laid out via visual constraints to help reduce developers cognitive overhead.

With such basic support we can further extend the capabilities to allow such diagrams to be used for communication among developers or provide support for building more complex queries based on the viewed items. Such queries can allow users to ask more 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.

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 the 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. A visualization representing the current selection (or the entire project) 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 mechanisms (via explorations) to find the relevant portion of the code that they might be interested in.

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 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 support\(^3\), 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.

\(^3\) Developers not recognizing these modules can explore and discover their functionality as shown with the ‘org’ module.
At this point the developer can select and remove the smaller modules from the view. Doing so results in Strata automatically expanding 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. Again 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’ module, and the ‘jedit’ module is again automatically expanded 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. The ‘util’ module being below the ‘jedit’ module, 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. The presence of the upward arrows indicates dependency cycles among these modules and shows that the involved modules are possibly working closely with each other.

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. Since such small manageable parts of the code do not include irrelevant details, using them in developer tasks can help productivity [13][66]. Relo therefore provides an interactive exploration interface for developers to 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. Relo provides a graphical presentation similar to UML class diagrams. As mentioned earlier, Relo automatically lays out each diagram, with layout rules that try to put components in predictable places 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 relationships, 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 1).

![Figure 5 – Relo started by opening EllipseFigure](image)

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 2). 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 3). The developer clicks on this handle to show superclasses, and therefore continues his exploration to find a relevant base class by clicking upwards (Figure 4).
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 5, 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 6). Relo is now acting as both a call-hierarchy browser and an inheritance-hierarchy browser.
Once presented with figure 6, 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.

Figure 9 – Expanding the class AbstractFigure and the method addFigureChangeListener
1.5. **SCENARIOS**

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

1. **Concrete Context Representation:** As the developer explores the code, a Relo diagram can represent his mental model, supplementing his short-term memory, by automatically creating and updating the relevant visualization of the code. At the same time, a Strata diagram can help the developer be situated in the codebase as he explore it.

2. **Lost in Code:** As the developer explores code in a traditional IDE, Relo can track navigation in the background for use if the developer has the feeling of being lost or if he has forgotten an important relationship that was traversed. The developer can ask Relo to launch a visualization based on the tracked navigation events. The generated diagram can help the developer see how things are 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 in code 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 result in a form of documentation that is linked to the code. Such diagrams can be used as structured documentation [61] 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 [67] 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 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**

In reducing developers' cognitive overhead, Relo and Strata have been designed to appear like commonly made sketches of code, displaying parts of large codebases, allowing incremental user-directed changes laid out via visual constraints to help reduce developers cognitive overhead. By building and evaluating these tools, this thesis shows that developers' understanding of code can be supported by a tool that creates visually constrained partial diagrammatic representations of interactively explored code.

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.
- 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 in-
volved in the current module respectively.

• 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 survey of developers experiences on the effectiveness of various do-
cumentation techniques.

• A survey of amount of dependency cycles in high-level modules of
popular publically available projects.

• A qualitative evaluation of a light-weight tool for showing overview
diagrams of modules by users on their own projects.

• An in-depth evaluation (comments from users in the field and con-
trolled quantitative and qualitative) of a static visualization based pro-
gram comprehension tool providing support for large codebases.

1.7. OUTLINE
This thesis starts with a brief overvi ew  o f  p r e v i o u s  w o r k  t h a t  S t r a t a  a n d
Relo have been informed from in Section 2. Results of a survey of devel-
opers experiences on the effectiveness of various documentation tech-
niques 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 dis-
cuss 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 become known there has only been limited usage of such tools. Below we describe work done in understanding developers’ behavior and cover previous tools helping developers either gain an overview or explore through their codebases. We also describe other tools that have been built to help developers work with large projects.

2.1. DEVELOPER BEHAVIOR

Program comprehension is agreed to be an important and large task, and ethnographic studies of developers’ behaviors have shown this fact. However, partially because comprehension is often a secondary task, studies measuring it vary in 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 [74]. 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 developers 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 [59], 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][86].

Developers typically take one of two strategies either 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 focusing only on the code elements relating to the task at hand at the cost of being more error prone. While most tools support systematic strategies, Relo and Strata also focus on the opportunistic strategies by providing developers multiple navigation steps provided in the context of the currently shown diagram.

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 into higher level abstractions [71][72]. This building of developer’s mental model has been found by Pennington [63] to happen in the 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 verification to form subsidiary hypotheses. High-level views of Strata support overviews of the code and therefore enable such 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 mod-
els (program model and situation model) but also a top-down domain model while understanding [92]. 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. [76] and has been used in evaluating the effectiveness of Relo as described in Chapter 7.

2.1.2 Usage 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] studies 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 upwards).

Relo supports these ideas by allowing developers exploration through box-and-arrow diagrams, and also has diagram types with code elements placed next to each other to minimize arrows. Relo’s approach to supporting diagram types also allows for extensions to support other attributes as found in the study, such as iconic pictures 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.

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][88] and providing schemas for their interactions [87][36]. A concrete instantiation of concerns on code, called Concern Graphs [67] 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][66]. 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 re-
relationships – this approach of aggregation of dependencies has been shown effective for high-level views [70].

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.
1) **Software Terrain Maps**

Figure 11 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) 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 12). 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.
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 descriptions. 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 13 shows the Armin display on initial extraction of dependencies, with Figure 14 showing the view after a short script is run.
Figure 13 – Armin view on loading dependencies

Figure 14 – Armin view after running a short script
The DEREF environment [90] takes things a step further (as shown in Figure 15) 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.

![Layered layout provided by Deref](image)

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.

4) **LDM**

An approach that has been effective in being automated has not been visualization related. LDM [70] consists of a tool that extracts dependencies and shows them without user-intervention in a matrix form. This approach (shown in Figure 16) while scalable requires significant 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 17), their diagrammatic view has a number of significant 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.

![Figure 16 – LDM on an old version of the Ant project](image-url)
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 18). 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 18. 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.
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 19, 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 19 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 code base 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.
2.3.2 Visualization based approaches

Visualization approaches have typically focused on presentation at the expense of exploration capabilities. While Reiss’ FIELD [64] system (shown in Figure 20 and Figure 21) supported user-directed exploration using graphical widgets its benefits were limited as exploration would only be on a single relationship.
Figure 20 – The call graph viewer of Field

Figure 21 – Class hierarchy browser of Field
SHriMP Views [64] supports comprehension by using fisheye-lens distortion for zooming in on targeted pieces of code (see Figure 22). 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 [83]. 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.

Figure 22 – Part of an exploration session with SHriMP
Another visualization approach, the TkSee Visualizer [93] provides support for users to perform queries to build a visualization for graphical exploration, and displays relationships using a radial layout (see Figure 23). 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 [87]. 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 23 – Part of an exploration session with the TkSee Visualizer](image)

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.

1) *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 artifacts shown in Relo and Strata to provide more useful directions to users.

2) *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.
3) Mylar

Mylar [44] is built as an integration into the Eclipse IDE (shown in Figure 24) 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 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 24 – Screenshot of Mylar
4) Jazz

The Jazz project [39] (shown in Figure 25) 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 25 – Screenshot of Jazz [39]](image)

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**

Our goal is to support users of comprehension tools who would likely be experts in programming but novices to the project or part of the code that they are trying to understand. These *software immigrants* have been previously studied by Sim and Holt on aspects other than the learning of the software system [73].

In order to inform the design of Relo and Strata, we conducted a survey of software developers focusing on their comprehension experiences. Previous research has focused on functionalities found in program comprehension tools [11], but we wanted our survey to focus on the documentation techniques used and wanted by developers when they looked at the code. Our attempt is closer to the study of Robillard et al. [65], though instead of focusing on comprehension strategies, we wanted to understand the effectiveness and needs of developers using various documentation techniques available to them. We also wanted to gain an insight on developer tasks when they looked at code, 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).

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 chose 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\(^4\) 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.

\(^4\) Survey questions available in the appendix.
Understanding Software: [Project Name]

1. [Required] What is the depth of your experience with this Project:
   - Core Developers
   - Examined Internal Code
   - API Client Only
   - Never saw the Source Code (this survey does not apply to you)

2. How would you rate the difficulty on understanding the code?
   - Can't really say
   - Very Hard
   - Hard
   - Like other projects
   - Easy
   - Very Easy

3. Are there any particular things that the code did which made it easy to understand?
   - Good Javadoc
   - Good Examples
   - Good Diagrams of code components
   - Good Articles describing using code
   - Good Articles describing code architecture
   - Other: [_________________________________

4. What would be the most beneficial things that you would suggest to add and make the code easy to understand?
   - More Javadoc
   - More Examples
   - More Diagrams of code components
   - More Articles describing using code
   - More Articles describing code architecture
   - 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?

[_________________________________
[_________________________________
[_________________________________
[_________________________________
[_________________________________
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 code. Table 1 shows the number of developers with each level of experience, and their estimated difficulty of understanding the code in their project, it is not surprising to note that most of the experienced (core) developers thought that their project was easy to understand, while other developers examining the code found it harder.

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

<table>
<thead>
<tr>
<th>Developer Experience with Project</th>
<th>Very Easy</th>
<th>Easy</th>
<th>Like other projects</th>
<th>Hard</th>
<th>Very Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Client</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Examined Code</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>Core Developer</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Effective Documentation Technique</th>
<th>% saying used effectively</th>
</tr>
</thead>
</table>

---

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

6 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.

7 Percentage of 78 users. The sum is more than 100% since each user could select multiple techniques.
Effective Documentation Technique | % saying used effectively
---|---
Javadoc | 44.8%
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 | 
Test cases | 

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>
<thead>
<tr>
<th>Documentation Techniques</th>
<th>Current</th>
<th>Want</th>
</tr>
</thead>
<tbody>
<tr>
<td>Articles describing code architecture</td>
<td>14</td>
<td>35</td>
</tr>
<tr>
<td>Articles describing using code</td>
<td>29</td>
<td>26</td>
</tr>
<tr>
<td>Diagrams of code components</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>Examples</td>
<td>30</td>
<td>23</td>
</tr>
<tr>
<td>Javadoc</td>
<td>35</td>
<td>19</td>
</tr>
</tbody>
</table>
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, is 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>
<thead>
<tr>
<th>Documentation Techniques</th>
<th>Current</th>
<th>Want</th>
</tr>
</thead>
<tbody>
<tr>
<td>Articles describing code architecture</td>
<td>9</td>
<td>23</td>
</tr>
<tr>
<td>Articles describing using code</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>Diagrams of code components</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>Examples</td>
<td>22</td>
<td>13</td>
</tr>
<tr>
<td>Javadoc</td>
<td>22</td>
<td>14</td>
</tr>
</tbody>
</table>

### 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. Since developers were drawing on their memory, probably some time after the event, these tasks are unlikely to be typical but rather memorable; a task 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. Tasks that required a static analysis of the code are indicated in that column, while those requiring a run-time analysis are indicated in another column. Items in the ‘other’ column refer to tasks that were too vague to determine whether they were static or dynamic.
Table 5: Tasks done by user when they were examining the code

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

From the table, it seems that support for over 66% of the tasks can be provided by helping users explore static relationships in the code.
4. **Strata User Interface**

The Strata interface provides for understanding code using commonly drawn 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 so that a developer can add or modify functionality.

The parts of the project under consideration by the developer are referred to here as *modules* or *components*. Modules can consist of multiple sub-modules to further organize the code, and form a key basic building part of a layered architecture interface. Ideally these modules have sharp, obvious boundaries with layered dependencies so that code can be clearly and easily be indicated to belong to the modules. This is needed since developers need to increase predictability of changes in the code and therefore minimize the impact of the changes. This attempt at lowering the coupling among modules, together with the goal of reducing dependency cycles and managing a potentially large number of modules results in the modules being ideally organized in a layered architecture. Given that a developer understanding an underlying codebase might be done in order to achieve some other primary task, Strata is designed to easily build layered diagrams.

However, without active intention to maintain the architecture these code boundaries decay. This decay can happen because of developer error, rapidly approaching code delivery deadlines, or the presence of cross-cutting concerns. While having such architectural diagram being created efficiently from the code is expected to promote effective architecture reviews and evaluations [32], having the architecture available and synchronized with the application implementation has been difficult and costly [47]. 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 dia-
grams from spaghetti code. It is therefore important to provide good defaults to generate layered architectural diagrams even with such decayed boundaries. It is also important to provide easy interaction to modify the generated diagram when the defaults are not correct, as opposed to existing tools which require expertise is using the tool to obtain a valid architectural diagram.

Thus Strata diagrams not only provide high-level overviews to help developers in understanding the current codebase, but this understanding of the current state of the codebase as different from the intended ideal version also provides an opportunity to maintain module interfaces. Below we describe the Strata interface and behavior and show how a user would explore using it.

4.1. Appearance

Layered architectural diagrams examples can be seen in Figure 2 and Figure 27. The main features of such diagrams are used in shaping Strata's appearance: The architectural diagrams consist of modules which are rectangular, as opposed to radial layouts such as those shown by some of today's tools (for example in Figure 14). Members of a module are laid out in a single row or column, and if there are too many items they are shown

---

8 Source: A typical architectural diagram used from one of Accenture's Project Documentation
using a grid like layout. Modules that build on other modules are shown in
a layer above it. As can be seen in Figure 28, Strata uses these points in
providing users an ‘expected’ diagram.

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 ex-
tact them in an efficient manner and they further add an obstacle to
building the diagrams. These runtime dependencies can be easily added to
the underlying dependencies for displaying to the user.

DEALING WITH IMPERFECT CODE

Beyond showing traditional layered diagrams, Strata also needs to help
developers with the real world constraints – helping in cases when the
module boundaries have decayed and when it is hard to detect the impor-

Figure 28 – Strata display of the jEdit project
(same as Figure 4)

RCM: Why? Not explained in text –
DRK asked in intro but here makes
more sense
tant modules. In such cases Strata attempts to make good guesses, based on those shown successful in more powerful reverse-engineering tools [70]. For example, in the absence of a provided module organization, the directory structure (or package structure in Java) is used as the default.

Using the directory structure, however, can result in the root of the code-base having some branches with very little code with other branches having most of the code. For example, in Figure 4 and Figure 28 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. The module with the largest size (measured in the number of classes) 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.

Similarly, the presence of dependency cycles makes it is hard to detect which dependencies are the correct/important ones for building the layers. Strata uses the number of dependencies between two modules as a proxy for the importance of the dependency between the modules. Dependencies between any two modules are the sum of any strong dependencies, i.e. class and member references, method invocations, and inheritance, as opposed to including weak dependencies, such as method parameters that are not actually referenced in the code.

While Strata does provide a good set of defaults it also allows for user interaction to correct its guesses. Modules can be easily moved to more appropriate layers, and can either be changed or removed from the diagram.

4.2. Layout

Strata focuses on building layered architectural diagrams. This layout is active and is applied by default whenever new items are added to a diagram as explained layer, such as by a developer dragging-and-dropping a module on the diagram. When a module is removed from the diagram the diagram is collapsed. The automatic layout rules are overridden when a user explicitly moves a module.

For the layout, basic (partitioning) algorithms are followed in splitting modules into layers so that the modules in higher layers have dependencies on the modules in lower layers [15][30][94]. These algorithms work by taking the given modules, and iteratively removing modules. The modules that don’t have any modules depending on them are removed and added to the top layer, while the modules that don’t depend on other modules are removed and added to the bottom layer.

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

The largest challenge in building the layers is however in dealing with the presence of unexpected and unplanned dependencies between modules. 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 28 generated by Strata and Figure 29 generated by LDM [70]. This requires Strata to have an appropriate strategy for dealing with cycles. Since inferring the correct dependencies is likely impossible Strata needs to provide good guesses, and further provide an easy approach for users to change the wrong guesses – the modules in incorrect layers guessed based on importance to the wrong dependencies.

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. 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 set problem, is known to be NP-complete [34] Strata uses a simple algorithm to approximate it.

\[9\] 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.
The algorithm works iteratively by first finding the smallest number of dependencies between any two modules, then removing those dependencies, and then applying the basic partitioning algorithm described earlier. This continues until all the cycles have been removed. 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]. We represent each of the nine modules using a number in the 0..8 range. 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 30, can be said to be a partitioning with a filter of 0. The algorithm now looks at the remaining cycle applying a stronger filter (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 atmost a cycle of size 3. The algorithm chooses not to break all the cycles since it would otherwise result in a very large number of layers – with diagrams that are too tall or too wide not considered as useful.

Once we have finished partitioning the modules, we can combine the results of the partitioning as shown in Figure 31.
Breaking the cycles and applying filters can mean that modules in two adjacent layers have no dependencies on one another. For example, in Figure 32 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.

4.3. **NAVIGATION**

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.

In one mode a developer could be intending to understand a project, or a subcomponent. 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 often wants 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 double-clicks on modules to expand them, and automatically expanding the largest module when there are 3 or fewer items shown.

In a secondary mode, a developer could be focusing on doing other feature addition or maintenance task, and would be curious wanting to situate himself. He would typically have an idea of the overall architecture and would be looking to Strata to place himself in the architecture. Strata supports this behavior by providing a linked mode, when developers’ explorations in the IDE are shown in Strata together with the modules that depend on the currently shown code and the modules that the currently shown code 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 with Strata, users are provided with button on selecting modules (as shown in Figure 34). These are for common exploratory actions like removing modules, expanding them to show their children, and showing which modules depend on the current module. While some of these have keyboard shortcuts, these navigation buds help developers to realize the available options and provide a means to easily explore the code.

![Figure 33 – Breaking a module in Strata.](image)

As in Figure 33, Strata allows users to 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.

![Figure 34 – Navigation Buds in Strata.](image)

When users select a module (org in this case) and click on the ‘show depending’ navigation bud, Strata shows the modules that depend on org. This however raises the question as to the granularity of the modules that should be shown on clicking the navigation bud. For example, consider a class inside the org package and a class java.lang.Foo which depends on it. The question is that when the Show Depending navigation bud is clicked whether the class Foo, the package java.lang, or the package java is added. Strata resolves this by adding items at the same level as the module clicked on – asking for dependencies of classes will show classes, and dependencies of package will show other packages that are nested to the same level (or higher).
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 we worked with, 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

When developers examine unfamiliar codebases they have problems keeping oriented. Studies have shown that developers think of code as laid-out spatially and want views of code to maintain a consistent look [26]. In an attempt to support bottom-up comprehension by showing details of methods and the different relationships Relo is based on UML class diagrams. Since previous tools designed to help in program comprehension have been shown to overwhelm developers, Relo takes the approach of showing the bare minimum information by partial diagrams – any items that are not explicitly indicated to be relevant are not shown to the developer.

5.1. APPEARANCE

In contrast to traditional software visualization tools which show code in basic box-and-arrow diagrams (such as can be seen in Figure 22), Relo’s appearance is tailored to be similar to UML class diagrams (as can be seen in Figure 36). 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 class from a package. Further, class members are shown using a stacked/vertical layout with no space between members, while the various classes are laid out in a less constrained layout. Relationships are also customized with inheritance relationships not being drawn as straight lines from the start to the end, but as stepped lines with closed arrows.
Since minimizing cognitive overhead is important, Relo shows as little information as possible such that every code element shown is easily identifiable. A method being displayed will therefore not automatically include the containing package, i.e. when code elements are shown will their container elements are not added to the diagram until explicitly requested. If an element is displayed without its parent, the element’s name is shown fully qualified – a Java method appearing by itself is prefixed by its containing package and class.

The approach of showing nodes only when explicitly requested, often results in cases when only some of the siblings of a node being shown. In order to indicate to the user that more nodes exist and to provide a mechanism to add them, the user is shown a ‘more items’ menu which when clicked lists the remaining sibling nodes. In contrast to commercial UML tools or other software visualization tools, this approach allows the current diagram to not include irrelevant details.

In showing the minimal amount of information, Relo uses automatically-triggered navigation services to show the obvious relationships to the graph via visual grouping present the diagram to have lesser information. One service automatically adds inheritance relationships. Another tries to reduce the amount of information shown by adding the container element when there are multiple siblings shown. These added relationships improve visual grouping and are therefore expected to ease comprehension tasks.
5.2. 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.

The first algorithm used by Relo, and the most popular one available in most graph tools, was based on the orthogonal graph layout algorithm [69][75]. These algorithms work by targeting the layout of edges with respect to horizontal and vertical constraints and then iterating to minimize edges crossovers. Such algorithms work hard on providing a globally optimal layout. However, this approach causes adding a new node to a diagram to possibly cause all nodes to be repositioned. This lack of a consistent diagram being built-up distracts 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, and forces to support directed constraints and graph containment [91]. The presence of such forces results in the layout engine having a number of local minima, which result in graphs not having the perfect layout but behaving in more predictable ways. Newly added nodes do not cause the rest of the graph to re-layout since doing so will cause nodes to move on top of other nodes and the repulsive forces will prevent this from happening. However, such force-directed approaches are limited in their need to model all constraints as forces, and perform poorly for complex conditions. Simple things like aligning either disconnected or vertically-connected nodes in a column are hard – and made impossible when one of these nodes has a relationship associated with it. Additionally, while force-directed layouts do support incremental layout, adding nodes does cause nearby nodes to unexpectedly move.

Relo therefore have 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 get triggered on the commonly occurring scenarios and are therefore laid out well. 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.

Give Rules and Figures showing how they work.
5.3. SUPPORTING INCREMENTAL 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.

5.3.1 Navigation Buds

When used in a primary mode, a developer browses 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 3, when a class is selected, it will sprout buds for different relationships that could be followed from the class (extends, extended-by). Clicking on a bud will make the visualization grow by showing more items having the appropriate relationship, i.e. following the selected buds relationship. Buds are only shown when they will result in a modification of the view, i.e. a class that is not extended by other classes will not show the extended-by bud (as in Figure 3). Buds provide support for a browsing behavior commonly observed in users trying to home-in on information based on the surrounding contextual information [87]. They are provided instead of navigation property dialogues or context menus for the configuration of shown or filtered relationships as required by most visualization tools.

At every code artifact, the most common relationship explorations that the developer will want to perform are made available as buds, to make them easily accessible. These buds are listed in Table 6. Complex relationship explorations, such as showing the entire subclass tree of a class, are available through a context-sensitive menu.

Relo also tries to limit the number of items added during exploration, by providing developers with a more items menu to get a list of children and add only relevant items. The navigation buds, while primarily designed for exploring, also provide features in 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 [27]. Further, just like the more items menu on code elements containing multiple children, right-clicking on a navigation bud lists the names of all code elements that will be added in a menu allowing the developer to choose to add a single item to the visualizations.

5.3.2 Levels of Detail

In order to minimize cognitive overhead, every element presented in a Relo visualization, defaults to showing as little information as possible. The user can semantically zoom-in by double clicking on an element or selecting the expand handle (‘+’) 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. 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. Additionally, Relo tries helping developers using code elements that contain a large number of children, such as packages and classes having over 100 classes and methods respectively. These items are automatically grouped based on access (public, protected, private), and a developer expanding a class will first only see public methods, followed by protected methods, and finally private methods. This keeps the diagram from growing too fast and at the same time provides the user with fine-grained controls.

The user can also collapse code artifacts by clicking on the ‘-’ handle. Alternatively, developers can selectively remove artifacts from the diagram by clicking on the ‘x’ handle. To ensure that developers think such an item is hidden and not deleted, when items are only removed from the view, 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 removal navigation bud.

5.3.3 Autobrowse

Developers trying to understand code can easily get lost in it. In particular, when they want to know how two code elements are connected, the high branching factor of the code can get them lost easily. Autobrowse is a feature of Relo designed to help in these situations, and tries to automatically browse through the code and nodes to the diagram. Autobrowse requires some starting points given by a developer, such as by selecting and right clicking two code-element under consideration for the current task. It then locates and shows short paths of relationships between the visible elements, and shows how the selected elements are connected. It does a simple breadth first search for hidden artifacts that are connected (and therefore relevant) to at least two of the selected items. Since some relationships, like inheritance, are considered more important than others, for any given path length they are searched first. The search terminates as soon as at least one path is found, displaying all found paths of the shortest length between the selected nodes. Developers can repeat autobrowse to add longer paths.

Relo adds items that have been removed from the diagram as not being relevant to the current task. Since only paths with relevant nodes are shown, this feature can be helpful to users. A developer can select two items, choose autobrowse and if he finds the added item not relevant, he can remove it and run autobrowse again – since the system tracked the removed item, it will not be added automatically, thereby allowing the developer to see how the elements are related ignoring utility or obvious connecting code elements.

Approaches such as those shown by Holmes and Murphy [38] can be used to provide improve the quality of results to users.
5.4. Relationship Model

Elements of Relo visualizations have been chosen heuristically based on formative evaluations with users. Relo elements include packages, classes (including nested classes, but not anonymous classes), fields, and methods (including constructors). The relationships used by Relo are shown in Table 6. Some relationships are rendered as navigation buds, but others are available only on a context menu. This relationships model is similar to that of concern graphs [66], but instead of focusing on describing 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. Further, we have chosen not to show relationships between methods and local variables, or methods and the classes inside them. These relationships can be easily extended to provide exploration of an object model by performing lightweight analysis [41], however, we chose to focus on the static relationships in Relo.

<table>
<thead>
<tr>
<th>Nav Bud</th>
<th>Ctx Mnu</th>
<th>Relationship</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Inheritance</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Method override</td>
<td>Method</td>
<td>Method</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Field access</td>
<td>Method</td>
<td>Field</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Field modify</td>
<td>Method</td>
<td>Field</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Field type</td>
<td>Field</td>
<td>Class</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Containment</td>
<td>Package</td>
<td>Class</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>Method</td>
<td>Method</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>Field</td>
<td>Field</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Method param.</td>
<td>Method</td>
<td>Class</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>Method calls</td>
<td>Method</td>
<td>Method</td>
</tr>
</tbody>
</table>

Relo allows the resulting diagram from an exploration session to be saved to a file. These files include the visualized nodes, relationships and user comment. The files can be used for future retrieval, or be used as a form of communication among project members.

10 For inheritance in the context menu we also allow users to open the inheritance hierarchy, i.e. the transitive closure of the relation.
5.5. **AUTOMATED DIAGRAM MANAGEMENT**

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

### 5.5.1 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 an 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.

### 5.5.2 Automated Removal of Items

Working on this sub section

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]. The approach keeps a selection count for any item being shown, increasing it when the item is selected while reducing it over time. Items with low values can be treated as not being relevant and are the first to be removed when the visualization takes a large amount of space.
Relo also provides an auto-remove mode, which automatically removes elements when more space is needed to display new items. Elements are removed based on a score, which is its number of visible edges minus the time since the user last selected it (measured by counting selection events). Auto-remove mode is automatically enabled during linked exploration. The developer can also manually enable or disable auto-remove at any time.

5.6. SUPPORTING COMMUNICATIONS USING DIAGRAMS

By providing a lightweight method for experts to provide various forms of annotations, developers can not only use Relo to understand an underlying codebase, but can also describe this understood knowledge along with domain knowledge for Relo to use and enhance the default visualization.

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 sessions/diagrams can then be saved for future reference or for communicating with other developers.

Once the developer starts exploring the code, he/she will mentally form a model of the portion of the code and groups components into chunks. Relo supports this process by allowing developers to describe the grouping to the system. Developers can interactively ‘draw boxes’ around siblings and group them into chunks. These chunks are represented by an icon based on the chunked items, and behave similar to other code artifacts, allowing users to zoom in/out, hide them, or can show relationships to them.

Add figure showing comment and named relationship, and add howto in text.
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 be processed: text selection, structured
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. Support for good response 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 [64].

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 so 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 37 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 37 – Basic Relo Architecture.](image)

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 \(<source, relationshipType, destination>\) 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 38 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 38 – Architecture with caching support for Relo.](image)

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 39, 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.
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 added 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

With developers often understanding code as a secondary task, 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. 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 it was noticed that most tools only provided limited support in 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 first evaluated Strata on an externally created collection of projects and measured the quality of the cycle breaking on these projects.

7.1.1 Methodology

The collection used for this evaluation was an externally provided collection of java projects, which were partially 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 below in Table 7, along with descriptions, project sizes, and information on extracting the collection again.

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Hosted</th>
<th>Col</th>
<th>Size</th>
</tr>
</thead>
</table>

Table 7: Projects used in determining Strata usefulness
<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Repository</th>
<th>Month</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azuereus</td>
<td>Popular GUI and Implementation of the BitTorrent protocol</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>434,676</td>
</tr>
<tr>
<td>Buddi</td>
<td>Program to manage personal finances and budgets</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>31,186</td>
</tr>
<tr>
<td>Cargo</td>
<td>API for managing J2EE containers</td>
<td>codehaus</td>
<td>Feb '07</td>
<td>64,726</td>
</tr>
<tr>
<td>Carol</td>
<td>Library for abstracting away different RMI (Remote Method Invocation) implenements.</td>
<td>objectweb</td>
<td>Feb '07</td>
<td>20,664</td>
</tr>
<tr>
<td>Sphinx</td>
<td>Speech recognition system</td>
<td>sourceforge/cmu</td>
<td>Nov '06</td>
<td>98,553</td>
</tr>
<tr>
<td>Commons-Codec</td>
<td>Implementation of common encoders and decoders</td>
<td>apache-jakarta</td>
<td>Nov '06</td>
<td>10,988</td>
</tr>
<tr>
<td>Dnsjava</td>
<td>Implementation of the DNS protocol</td>
<td>sourceforge</td>
<td>Feb '07</td>
<td>33,634</td>
</tr>
<tr>
<td>jEdit</td>
<td>Configurable text editor for programmers</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>159,908</td>
</tr>
<tr>
<td>jMemorize</td>
<td>Tool involving simulated flashcards to help memorize facts</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>17,187</td>
</tr>
<tr>
<td>Jmol</td>
<td>Tool for viewing chemical structures in 3D</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>100,131</td>
</tr>
<tr>
<td>JRuby</td>
<td>Implementation of the Ruby programming language</td>
<td>codehaus</td>
<td>Nov '06</td>
<td>82,081</td>
</tr>
<tr>
<td>JTrac</td>
<td>Web Application for issues-tracking</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>10,288</td>
</tr>
<tr>
<td>Radeox</td>
<td>API for rendering wiki markup</td>
<td>codehaus</td>
<td>Nov '06</td>
<td>11,811</td>
</tr>
<tr>
<td>Rssowl</td>
<td>Newsreader supporting RSS</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>81,647</td>
</tr>
<tr>
<td>TvBrowser</td>
<td>Extensible TV-guide program</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>149,564</td>
</tr>
<tr>
<td>Zimbra</td>
<td>A set of tools involving instant messaging</td>
<td>sourceforge</td>
<td>Nov '06</td>
<td>579,288</td>
</tr>
</tbody>
</table>

Each project was opened in Strata both with cycle breaking enabled and with it disabled. When opening a project, we realized that at some level most projects have cycles and therefore wanted an objective method for selection the modules shown. Since having too many modules can result in developer’s getting overwhelmed we Strata provides a top-down expansion interface thereby allowing users to only see the items that they are focusing in on. We therefore chose to have approximately 10-12 modules shown – this was done by expanding the largest module if either (a) there where less than 6 modules, or (b) the largest module was more than twice

---

11 All projects were extracted on from the head of source repository at the given months.
the second largest module. These 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

Data collected from expanding the various projects is shown in Table 8. 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.

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

Any diagram that has too few layers will not be helpful. We therefore plotted the number of columns as a percentage of the total number of modules in the diagram (Figure 40). 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.
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 that the ideal width. Figure 41 and Figure 42 show the RSSOwl project with cycle breaking and with the traditional partitioning algorithm being used.
Among the various projects only 4 (25%) had no cycles at the level we expanded. And only 1 of these had no cycles at all as we further expanded. The project that had no cycles was the apache commons codec project, which is a library of independent codes and can therefore be argued to have no cycles. The presence of cycles in most of the project, shows the high-tendency of cycles being added to codebases, and therefore a need for light-weight and easy to use tools to point the cycles out.

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 41). Similarly the algorithm often put the `gui` modules in higher layers. In the Buddi project the `view` and the `controller` depended on the `model`. 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).

### 7.2. **USER-STUDY WITH STRATA**

With support for cycle breaking built and tested with various projects we wanted user-feedback of developers actually 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. These projects are listed above in Table 9. 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).

<table>
<thead>
<tr>
<th>Project</th>
<th>Project Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longwell [8]</td>
<td>Open source research system for refining heterogeneous datasets</td>
<td>18k 127</td>
</tr>
<tr>
<td>System A</td>
<td>In production, patient management J2EE application for a large health care firm.</td>
<td>127k 553</td>
</tr>
<tr>
<td>System B</td>
<td>In use, research system for data mining.</td>
<td>13K 61</td>
</tr>
<tr>
<td>Lapis [7]</td>
<td>Open source research system for building programming by demonstration based editor</td>
<td>142k 475</td>
</tr>
<tr>
<td>Project</td>
<td>Project Description</td>
<td>Size</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>??? - ALG</td>
<td>???</td>
<td>??? - ???</td>
</tr>
<tr>
<td>??? – DR</td>
<td>???</td>
<td>??? - ???</td>
</tr>
<tr>
<td>Babel</td>
<td>Open source system to convert data to/from various semistructured formats (including Bibtext, RDF/XML, N3, etc.)</td>
<td>??? - ???</td>
</tr>
</tbody>
</table>

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

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 would need 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. Figure 43 shows the display after expanding the largest module for System A.
From the figure it can be seen that some of the modules use the exceptions provided by the project. Also 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. 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 once. 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 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 43 a limitation of the default views shown in Strata was that they sometimes showed the modules along technology boundaries – a manner that is not the default way of thinking about the project. While the technical separations were useful, the business domain diagrams where expected to be the default. The developer did appreciate being able to quickly create their own module descriptions and use it with Strata (as shown in Figure 44).

![Diagram of System "A" in Strata](image)

*Figure 44 – 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 code\(^{12}\). 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 ??? lines of code.

In order to best represent experienced software developers, study participants were required to have at least a year of worth of Java development experience. Participants had further not 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 where 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.

\(^{12}\) Size of code base was measured using the command ‘wc’.
Participants used a think-aloud protocol [52][62], and their actions were recorded by screen capture software and event logging. The study concluded with a questionnaire and a short semi-structured interview.

1) The Study Tasks

Header on both tasks: “Perform the below task, and then click the continue button shown at the bottom (hints are provided after the task description).”

Footer on both tasks: “Click the button when you know exactly what needs to be changed to fix the bug.” Followed by a button having the text “Done Task. Proceed to Questions >”

The Ant task started by: “What is the cause of the below bug? What needs to be modified to fix it?” and included a copy of the Ant bug description 37426\(^{13}\) and a test-case to reproduce the bug. Through preliminary studies questions we realized and added a small background as well.

We added hints to both tasks to make them solvable in the time, which were independent.

---

**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
<?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 45 – Background for the Ant task.

---

\(^{13}\) [http://issues.apache.org/bugzilla/show_bug.cgi?id=37426 on Ant version 1.6.5](http://issues.apache.org/bugzilla/show_bug.cgi?id=37426 on Ant version 1.6.5)
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
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'

```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}"
           classpath refid="ant-test.classpath"/>
  </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" />
      </junit>
    </junit>
    </junit>
    <target>
    <project>
```

Figure 47 – 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 "T
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 48 – 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.

You can launch Lapis by running Main.main method.

**Figure 49 – The Lapis task.**

<table>
<thead>
<tr>
<th>Hints (try answering these to figure out the bug):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. What code is responsible for the &quot;Named Patterns&quot; list, i.e. where was it created?</td>
</tr>
<tr>
<td>2. Where should you be adding the fix, i.e. what gets called when a node is selected in the Named Patterns list?</td>
</tr>
<tr>
<td>3. What already implements the functionality that needs to be added, i.e. what gets called when the node is expanded?</td>
</tr>
</tbody>
</table>

**Figure 50 – Hints for the Lapis task.**

2) **Comprehension Questions**

Measuring understanding of code can be a very difficult. We chose to use ask users a set of questions about the codebase after the fixed the bug. To ensure that these questions were not biased, the questions were selected based on a study done by Sillito et al. [?] on the questions programmers ask when doing such coding tasks. There are 8 such types of questions and are listed in Figure 51 and Figure 52. 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 51 – 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?
7.3.2 Results

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 [83]. By contrast, participants did not find Relo to be overwhelming (2.6 on a 7-point Likert scale 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 only asked those users that finished a significant portion of each task to answer the questions related to it. Table 10 contains each users progress and the number of questions answered. Progress was encoded based on the below list. Each task had three hints and the corresponded to the main points 4, 5, and 6 below:

---

14 In the Likert scale, 1 represented ‘strong disagreement’, 3 represented ‘disagreement’ 4 represented ‘neutral’, 5 represented ‘agreement’ and 7 represented ‘strong agreement’.
1. Main method
2. Oriented (breadth-first)
3. Oriented (depth)
4. Found Point 1/src
5. Found Point 2/dst
6. Found Point 3/conn
7. Suggested an Answer (typically when cut before verifying that the solution was as designed)
8. Were comfortable with an answer

Table 10: Users performance with Relo on Study Tasks

<table>
<thead>
<tr>
<th>User</th>
<th>Proj</th>
<th>Relo</th>
<th>Progress</th>
<th>Questions</th>
<th>Proj</th>
<th>Relo</th>
<th>Progress</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ant</td>
<td>Yes</td>
<td>5</td>
<td>Lapis</td>
<td>No</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Lapis</td>
<td>Yes</td>
<td>6</td>
<td>Ant</td>
<td>No</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ant</td>
<td>No</td>
<td>8</td>
<td>Lapis</td>
<td>Yes</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Lapis</td>
<td>No</td>
<td>6</td>
<td>0</td>
<td>Ant</td>
<td>Yes</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Ant</td>
<td>Yes</td>
<td>5</td>
<td>Lapis</td>
<td>No</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ant</td>
<td>No</td>
<td>4</td>
<td>Lapis</td>
<td>Yes</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Lapis</td>
<td>Yes</td>
<td>4</td>
<td>Ant</td>
<td>No</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Ant</td>
<td>No</td>
<td>5</td>
<td>4</td>
<td>Lapis</td>
<td>Yes</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>Lapis</td>
<td>No</td>
<td>6</td>
<td>1</td>
<td>Ant</td>
<td>Yes</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Ant</td>
<td>Yes</td>
<td>5</td>
<td>6</td>
<td>Lapis</td>
<td>No</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>Lapis</td>
<td>No</td>
<td>7</td>
<td>4</td>
<td>Ant</td>
<td>Yes</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>Lapis</td>
<td>Yes</td>
<td>8</td>
<td>4</td>
<td>Ant</td>
<td>No</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Ant</td>
<td>Yes</td>
<td>6</td>
<td>6</td>
<td>Lapis</td>
<td>No</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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 53) we see that of the three users doing worse with Relo, in two of the cases (4 & 5) the users were 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.

![Progress Improvement with Relo](image)

Figure 53 – Users progress improvement with Relo.

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 ap-
proaches 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 8.5/10\(^{15}\)). 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.

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:

“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).”

\(^{15}\) As of October 12th: http://www.eclipseplugincentral.com/Web_Links-index-req-viewcatlink-cid-19-orderby-rating.html
8. Future Work

Ideal program comprehension support needs a number of components. This thesis will in particular not make contributions in three important components. The primary reasoning for this decision is that a comprehensive consideration of any of the topics will take potentially 3+ years and the benefits of including such support are not clear. That is not to say that the thesis will not provide support for them, it is just that the support will be minimal.

8.1. Design Patterns

Experienced developers have a good understanding of design patterns, and are often able to use them effectively in understanding parts of codebases. While design patterns seem to form an essential part of understanding code, using them in building effective program comprehension tools, has obstacles in:

- Effective recognition of patterns.
- Visualization of roles performed by classes in the design pattern (especially in cases where a class performs more than one role, such as in different patterns).

8.2. Static Program Analyses

Static program analyses can extract a number of useful properties from code that can help in program comprehension. However, Relo is taking the approach of providing a lightweight interaction for the user so that his interactions explicitly provide the results of such analyses. Automatic extraction of all important properties will require:

- Formal enumeration of all program comprehension tasks;
• Definition of the important properties needed for the above tasks;
• Analyses to extract such properties from the code.

Effective implementation of such analyses can take significant computation time that would result in the lack of effectiveness of such techniques.

8.3. Runtime Information

Depending on the task, values taken up by variables at runtime can be helpful in program comprehension. Relo, however, does not leverage this approach since developers are often reluctant to use support already provided by debuggers and profilers due to issues with computational performance, distributed execution, application interactivity, etc.

8.4. Different Diagram Types

Message Sequence Diagrams

Strata: Colors and Size of boxes for recency, current task and playing with Global Landscape.
REFERENCES


