Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases

Vineet Sinha
vineet@csail.mit.edu

Rob Miller
rcm@mit.edu

David Karger
karger@mit.edu

MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)
32 Vassar Street, Cambridge, MA 02139

ABSTRACT

As software systems grow in size and use more third-party libraries and frameworks, the need for developers to understand unfamiliar large codebases is rapidly increasing. In this paper, we present a tool, Relo, that supports developers' understanding by allowing interactive exploration of code. As the developer explores relationships found in the code, Relo builds and automatically manages the context in a visualization, thereby helping build the developer's mental representation of the code. Developers can group viewed artifacts or use the viewed items to ask Relo for further exploration suggestions. Relo is built as an Eclipse plug-in integrated into the Java Tooling (JDT), and uses a standard, RDF, based backend allowing for maintaining code relationships and performing inferences about the relationships.

See also the related demonstration & poster at OOPSLA

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User interfaces;
D.2.6 [**Programming Environments**]: Graphical environments, Interactive environments.

General Terms

Management, Design, Human Factors.

Keywords

Program Comprehension, Program Understanding, Software Visualization, Large Software Systems.

1. INTRODUCTION

As software grows in size and complexity, developers face increasing difficulties in comprehending it and maintaining a coherent mental model of the code. Techniques like object-oriented programming and design patterns have helped control complexity in large projects by allowing developers to create and use appropriate abstractions and encapsulate inessential details. Unfortunately, these techniques make certain parts of program comprehension harder, requiring a developer reading the code to follow multiple forms of relationships. For example, following a function call, once a simple task, now also requires keeping track of inheritance and polymorphism. This task of being forced to follow multiple different relationships when trying to understand a portion of the code results in developers losing their context while exploring code.

In this paper, we present a program comprehension tool called Relo that aims to help developers deal with the different types of relationships in the software system. Relo allows developers to direct their exploration through the code while providing support for managing the context of the explored code. This support for explicitly showing the explored code surroundings via incremental visualizations, allows developers to build a consistent mental model (representation) of the code. Relo uses simple techniques to ensure developers' control over the visualization, while leveraging visual constraints to generate diagrammatic representations of the code. Additionally, Relo builds on these control primitives by automatically doing simple searches and adding or removing code artifacts.

Prior work on program comprehension [1, 5, 6, 7] has shown that while developers examine small codebases in a systematic manner, they use an as-needed bottom-up exploration strategy for large codebases. Relo visualizations therefore start with a single code artifact (such as a package, class, or method), from which a developer can browse the different types of relationships, incrementally adding more code artifacts. Relo helps developers in building a consistent representation of the code by showing artifacts in expected positions using visual constraints such as containment and left-to-right ordering in the diagram. As developers manage the visualization by adding, removing, or grouping artifacts, the traversed relationships, artifacts, and chunks are shown to help maintain the developers' context.

Relo visualizations show only a small manageable part of the code and like concern graphs [14] do not include irrelevant details, allowing a developer to focus on the important relationships. Relo visualizations try to be intuitive to developers, showing code artifacts in diagrams similar to UML class diagrams, while at the same time allowing developers to zoom 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 code. Relo further helps maintain developers' understanding of the code by providing explicit support for managing the amount and presentation of information to the developer based on his/her interaction with code elements.

2. PREVIOUS WORK

Relo's strength comes from providing an intuitive interface for an incremental user-directed exploration of large projects. Previous approaches to user-directed exploration have done so by using multiple distinct views each supporting only a single predetermined relationship (like inheritance or method-call hierarchy). 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. JQuery [4] tries bringing the multiple relationships together in a single tree-view. It allows developers to perform queries on nodes in an as needed manner, and then uses query results to populate children of the node. 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

OOPSLA'05 Eclipse Technology eXchange (ETX) Workshop.
Oct. 16-20, 2005, San Diega, California, USA.
Copyright 2005 ACM

children may represent containment, but for other parts, they may represent method calls. Relo also overcomes the loss in context by bringing the different relationships together in a single view, but uses diagrammatic constraints, such as containment or left-to-right ordering, to represent the different relationships.

Visualization approaches have focused on the presentation of code information instead of exploration capabilities. Reiss' FIELD [15] system used graphical widgets but supported user-directed exploration across one relationship only. SHriMP Views [2] supports comprehension by using multiscale graph visualization, fisheye-lens distortion, and zooming in on targeted pieces of code. Its expansion in a user-directed manner happens only along the containment axis – users cannot select a code artifact of interest and choose to expand the visualization across a relation of interest. This approach of showing all siblings even when interested in a different relation tends to overwhelm [3]. In contrast Relo 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 choosing relationships and nodes, it uses global filters and are not designed do not support exploration as part of the user's interaction with the visualization. For example, it is hard to get the system to first focus on inheritance in some parts of the visualization and later on method calls in other parts.

Another visualization approach, the TkSee Visualizer [16] supports users performing queries to build a visualization for graphical exploration and displays relationships using a radial layout. It however limits the user control of the visualization and does not allow zooming or removal of irrelevant items. Items are added by users specifying queries in a dialog box outside the visualization, instead of allowing the users to leverage contextual information to browse as shown to be needed by users [13]. Relo further uses visual constraints to present nodes in expected locations. Relo visualizations are similar to those proposed when studying navigation behaviors of programmers in traditional IDE's [17], but also provide for incremental exploration in building the visualization.

Compared to the previous techniques, Relo takes a hybrid approach: Relo leverages user-directedness in reducing cognitive overhead from viewing multiple elements, and uses a graph-based view with automatic layout, placing nodes and children in predictable locations. Further, Relo uses direct-manipulation browsing techniques, like handles (described later) to implicitly query and build the relevant graph for users. Design tools with reverse-engineering capabilities like Rational Rose [9], TogetherJ [10], and Fujaba [11] provide another kind of support for program comprehension. However, these design tools are aimed at developers who already understand the code, allowing them to create diagrams as documentation. Relo instead focuses on exploration involved in the comprehension process.

3. WALKTHROUGH

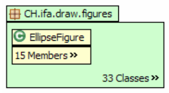
Relo is built with the intent of supporting developers explore the static structure of code, in a UML like visualization. We illustrate how Relo would be used by a developer for typical comprehension task. For this example, we use a task similar to that used by JQuery [4]. The task involves a developer working with the `JHotDraw` [12] project, a GUI framework for building drawing applications consisting of figures like rectangles, triangles, ellipses, etc. A developer needing to add a feature that operates on figures would like to understand how to manipulate them. In attempting this task, the developer will try to understand the code, by likely 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, at step 4, when the developer selects 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 when examining the roles of nodes connected by multiple relationships – in this case: the inheritance, containment, and method calls relationships.

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` would be 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 1a).

Figure 1a shows that the class has 15 members, and the developer clicks on the menu to see a list. Considering the method `basicMoveBy` as interesting, he clicks on the method name in the menu and thereby adds the method to the diagram for future examination. Once added, the developer clicks on the class, and is presented with a handle indicating the class inherits from another class (shown in Figure 1b). The developer clicks on this handle to show superclasses, and continues his exploration to find a relevant base class by clicking upwards (Figure 2).



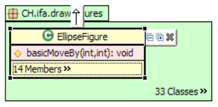


Figure 1a: Relo started by opening `EllipseFigure`
Figure 1b: Adding method and showing the classes handles

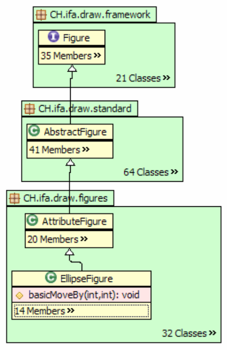
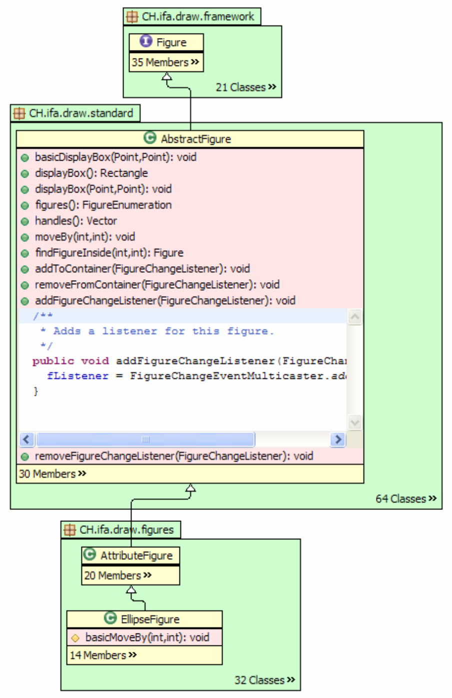


Figure 2: After clicking on the inheritance handles

Once the developer has an idea of the inheritance tree of figures, he chooses to expand the `AbstractFigure` class. After double-clicking to see all public methods, the developer removes methods irrelevant to his task (manipulating figures) by clicking on the 'x' in the corner, and examines the available methods to select one for expansion. Deciding that the `addFigureChange-Listener` method is part of the general framework for manipulating figures, the developer decides to expand it.

The developer is presented with Figure 3, which shows the implementation of the method. After finding the implementation relevant, the developer will want to find a relevant caller of `addFigureChangeListener`. The developer collapses the `AbstractFigure` class and clicks on the caller handle, Relo continues to build the graph (shown in Figure 4), and has begun to act as both a call-hierarchy browser as well as an inheritance-hierarchy browser.


CH.ifa.draw.framework
Figure
35 Members »
21 Classes »
CH.ifa.draw.standard
AbstractFigure
basicDisplayBox(Point,Point): void
displayBox(): Rectangle
displayBox(Point,Point): void
figures(): FigureEnumeration
handles(): Vector
moveBy(int,int): void
findFigureInside(int,int): Figure
addToContainer(FigureChangeListener): void
removeFromContainer(FigureChangeListener): void
addFigureChangeListener(FigureChangeListener): void
removeFigureChangeListener(FigureChangeListener): void
30 Members »
64 Classes »
CH.ifa.draw.figures
AttributeFigure
20 Members »
EllipseFigure
basicMoveBy(int,int): void
14 Members »
32 Classes »

```
/**
 * Adds a listener for this figure.
 */
public void addFigureChangeListener(FigureChai
  fListener = FigureChangeEventMulticaster.ad
}
```

Figure 3: Expanding the class `AbstractFigure` and the method `addFigureChangeListener`

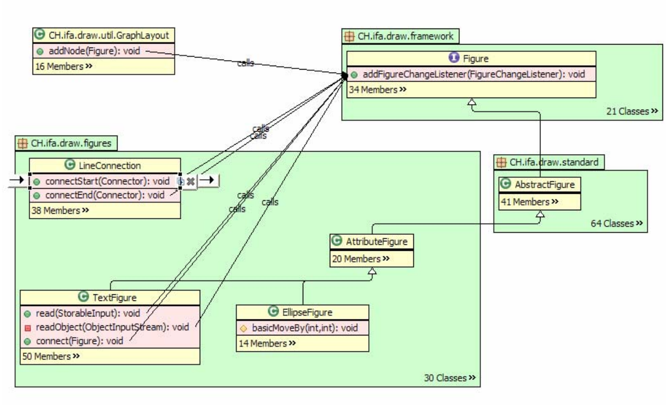


Figure 4: Asking for callers of `addFigureChangeListener`

Once presented with figure 4, the developer can easily select the relevant classes that manipulate figures, and does not have two 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 code base.

4. USER DIRECTED EXPLORATION

Relo provides capabilities to show arbitrary code artifacts together, using browsing handles to help manage the artifacts, and annotations to group artifacts into larger chunks. It further links itself to exploration in Eclipse so that developers can easily switch between Relo and other navigation tools.

4.1. Browsing Handles

Developers browse the code in Relo diagrams by using "handles" to navigate and extend the visualization with simple clicks. Instead of requiring the developer to navigate property dialogs or context menus to configure relationships to be shown or filtered as done by most visualization tools, Relo presents context sensitive buttons on the currently selected code element. For example, as shown in Figure 1b, when a class is selected, it will sprout handles for different relationships that could be followed from the class (extends, extended-by). Clicking on a handle will make the visualization grow by showing more items having the appropriate relationship, i.e. following the selected handles relationship. Handles are only shown on the most common relationships for exploration 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 handle (as in Figure 1b). This also results in handle clicks appearing as 'instantiating' the handles as relations to code artifacts; after a click, all the relationships of the handle's type will be shown and the handle will no longer be shown on the source code element.

4.2. Linked Exploration

Relo automatically synchronizes to explorations made by the developer in other Eclipse views. This allows the developer to work using the standard views, and at any time decide that he has possibly lost context, and would like a Relo visualization to help him. When asked to open such a visualization, Relo uses the developer's exploration history, presents the developer with a dialog to select how far back to go, finds both the code elements viewed and the relationships traversed, and then shows these nodes and relationships while building the diagram. Once linked, Relo continues to track the developer's exploration in the other views and updates the visualization to help provide context to the developer. 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 shown to the developer. With this bootstrapped diagram generated, the developer can switch to using Relo for his exploration and have the benefits of the various automatic and explicitly invoked agents (described in section 5).

4.3. Annotations

As developers understand code, their understanding moves from a structural model to a model consisting of data-flow and functional abstractions [5]. For example, systems using model-view-controller architectures can have the associations between components (model-controller or view-controller) carefully separated into factories in the code, however, a particular view of the code could have such relationships added by the developer. Relo helps users maintain these forms of understanding by providing support for basic types of annotations. It allows developers to interactively create named relations between items being shown, group components into chunks, and add comments to the visualization, to allow the developers to represent formed higher level abstractions when examining the code [1]. These annotated diagrams can then be saved for future reference or for communicating with other developers. Thus, important properties of the system that might not necessarily be in the code can be annotated into a live visualization connected with the code.

5. MANAGING CODE ELEMENTS

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.

5.1. Live Agents

Live agents monitor one of a number of events on the shown code artifacts and make modifications to the generated visualization. One such agent automatically draws the containing class or package when there are multiple artifacts that share the parent. Other agents listen to code artifact selection, creation, or other user-performed actions, and then either add elements or vary properties of the visualization. By adding these obvious code artifacts to the diagram, Relo is able to reduce the information that needs to be understood by the developer even though information is added to the visualization. Similarly, Relo also draws direct inheritance relations between elements shown in the visualization. These simple agents thus work together in providing an intelligent experience to the user. Since these agents only provide simple basic functionality, the developer using Relo has the feeling of still being in control of the interface and the elements presented.

5.2. AutoBrowsing

Relo builds on top of the agent infrastructure to help developers explore by implementing an *Autobrowse* feature. Autobrowse tries to model a simple directed exploration activity by a developer, between two or more selected items. It effectively does a breadth first search finding other hidden artifacts that are relevant to more than one visible item. Since some relationships, like inheritance, are considered more important than others, they are searched first, with the system terminating after an item is added to the view. Developers can repeat autobrowse to add more items. They can also select a subset of items in order to have autobrowse explore only the smaller set.

5.3. User Control

With agents automatically adding elements, the system needs to support developers removing these added items (by using one of the handles). By default, an agent would be triggered again, and could result in the developer-removed item being added again. In such cases, Relo keeps track of all such exceptions to the agent's actions that have been performed, and does not automatically allow agents to create such elements. A developer using autobrowse can take advantage of these user driven exceptions in the system, when on finding added item to not be relevant, the developer can remove the added item and run autobrowse again; since the system tracked the removed item, it is not added automatically. Thus, developers can see how artifacts are related while ignoring noise artifacts like utility artifacts or obvious connecting code artifacts.

6. PRESENTING CODE ELEMENTS

Relo only lays out newly added elements, while making code elements previously moved by the developer have their positions fixed. It further shows the addition, removal, or moving of elements by the system using animation so that the developers do not loose context while working with the system. While maintaining the context, Relo tracks how much detail must be shown (in each code artifact's view), as well as how to constrain the layout of an artifact.

6.1. Levels of Detail

In order to minimize cognitive overhead on developers, every element presented in a Relo visualization, defaults to showing as little information as possible. Developers 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 would mean showing the method implementation in an editor view. Instead of expanding a code artifact to show all public members, developers can also use the *more items* menu to get a list of children and add only the relevant items. Like expanding, developers can also collapse code artifacts by clicking on the '-' handle, and can selectively eliminate artifacts by clicking on the 'x' handle.

6.2. Constrained display of elements

Relo tries to reduce cognitive overhead by using topological constraints to assist in providing a default layout of code elements, so that elements are found at expected locations. Wherever possible, inheritance edges are drawn vertically, method calls horizontally, and containment is shown by visual nesting. Children are shown by default in class-diagram based defaults: package children are shown using a graph layout engine, while class children are shown using a vertical layout. In cases when the containment hierarchy is not important, children are laid out independently of the parent. Relo allows developers to select an element and get it to *break* from its current layout, into one of these three options. In addition, as mentioned previously, to reduce clutter, Relo uses agents that automatically add obvious relationships to items that have not been 'broken', i.e. code artifacts that are not laid out vertically next to each other, such as the default layout of methods in a class.

6.3. Automatic Chunking

In some cases, code artifacts can contain a large number of members. Packages and classes in some codebases can have over 100 members. In order to control the layout of the large number of elements, these items are automatically grouped using one of a few simple heuristics. By default, grouping is based on access (public, protected, private), followed by members being grouped by name similarity. The goal of the automatic chunking is to get the number of elements at any level down to less than 10 elements, and Relo therefore uses different types of the available chunking in turn to reduce the number of shown items.

7. INTEGRATION WITH ECLIPSE

The primary goal of Relo is in supporting developers work with large codebases. To accomplish this, we have built Relo integrated into the Eclipse IDE. In order to enable working rapidly on large codebases, having agents operating continuously on the shown items, and without consuming a large memory footprint, Relo compiles the entire underlying codebase into a database. This database stores all nodes with their relationships as triples of the form <*source, relationshipType, destination*> and uses the W3C standard RDF [19] to store in a database. In order to provide the implementation access to the different representations of java element, Relo builds a mapping engine for converting the four representations: Eclipse's Java Parser `ASTNodes`, Eclipse's Java UI `Elements`, RDF `Resources`, and Relo's visible `Artifacts`. With a mapping engine available, Relo leverages the incremental builder framework of Eclipse and transparently extracts the relevant relationships into the database. Adding support for any Eclipse language, involves building a mapping engine, providing a extractor of relationships for the compiler, and providing any user-interface customizations.

8. EVALUATION AND FUTURE WORK

In order to evaluate the usefulness of Relo, we have started conducting formative evaluations using the tool on projects of over 150,000 lines of code. Developers have found Relo useful in tasks where their exploration strategy typically needs more than 2-3 hops to find their target. Since these developers examine the code using an 'opportunistic strategy', i.e., not examining the code in a systematic manner, one developer doing a task can find Relo to be very useful while another developer doing the same task can find Relo to not be useful. We are targeting to characterize these situations and find means for Relo to help developers be more 'opportunistically successful'. We are also investigating multiple types of views to help developers understand codes at different levels of granularity.

9. CONCLUSION

We have presented a program comprehension tool Relo, which uses software visualization to help manage developers context and support comprehension in large software project. We have conducted a preliminary evaluation of the tool and regardless of bugs, developers have found Relo to be useful and have wanted to use it in their development tasks. Relo is built as an integrated plug-in into the Eclipse environment, and is to be made freely available from: **http://relo.csail.mit.edu**

REFERENCES

[1] B. Shneiderman. "Software Psychology: Human Factors in Computer and Information Systems." Winthrop Publishers Inc., 1980

[2] M.-A. Storey, H. Muller, and K. Wong. "Manipulating and documenting software structures using SHriMP views", ICSM 1995.

[3] M.-A. Storey, H. Muller, and K. Wong. "How Do Program Understanding Tools Affect How Programmers Understand Programs?", WCRE 1997.

[4] Doug Janzen and Kris De Volder. "Navigating and Querying Code Without Getting Lost", AOSD 2003.

[5] N. Pennington. "Stimulus structures and mental representations in expert comprehension of computer programs". Cognitive Psychology, 19:295-341, 1987.

[6] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. "Designing documentation to compensate for delocalized plans". Communications of the ACM, 31(11):1259-1267, 1988.

[7] M.-A. Storey, F. Fracchia, and H. Muller. "Cognitive design elements to support the construction of a mental model during software visualization". IWPC'97.

[8] Sim, S.E. and Holt, R. C. "The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize", ICSE 1998

[9] Rational Rose, IBM, http://www.ibm.com/software/rational/

[10] Together Technologies, Borland Software Corp., http://www.borland.com/together/

[11] Fujaba Tool Suite, Universität Paderborn Software Engg. Group. http://wwwcs.uni-paderborn.de/cs/fujaba/

[12] JHotDraw. http://www.jhotdraw.org/

[13] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. "The perfect search engine is not enough: a study of orienteering behavior in directed search". CHI 2004.

[14] Martin P. Robillard , Gail C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies", In ICSE 2002, Orlando, Florida

[15] Reiss, S. "Visualization for Software Engineering – Programming Environments", Chapter 18, pages 259-276, in "Software Visualization", ed. Stasko et al.

[16] Wang, L. "Animated Exploring of Huge Software Systems", Masters Thesis, School of Information Technology and Engineering, University of Ottawa, 2002

[17] Ko, A. J., Aung, H., and Myers, B. A. (2005). "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks". ICSE 2004.

[18] Lindgaard, G., "Usability Testing and System Evaluation: A Guide for Designing Useful Computer Systems", 1994, Chapman and Hall, London, U.K. ISBN 0-412-46100-5

[19] O. Lassila and R. Swick. "Resource description framework (RDF): Model and syntax specification", http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, February 1999. W3C Recommendation