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

Vineet Sinha *vineet@csail.mit.edu*
David Karger *karger@mit.edu*
Rob Miller *rcm@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, with Relo providing features to limit the growth of the diagram. To ensure developers don't get overwhelmed, Relo has been built with a user-centered approach, and preliminary evaluations with developers exploring new code have shown them to find the tool intuitive and helpful.*

## 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. Modern tools and programming languages have coped with this by allowing the encapsulation of inessential details and the creation of appropriate abstractions. However, such techniques, like object-oriented programming and design patterns, make program comprehension harder for developers joining a project, since they require a developer reading the code to keep track of many different relationships and roles. For example, following a function call to its definition, once a simple task, now also requires keeping track of inheritance and polymorphism. This complexity brought about by the interaction of multiple types of abstraction mechanisms introduced by modern object-oriented programming, forces a developer to explicitly keep track of their context while exploring and trying to understand the code.

In this paper, we present a program comprehension tool called Relo, which is designed to help developers explore and understand small focused parts of large codebases. Such small manageable parts of the code do not include irrelevant details, and using them in developer tasks can help productivity [6, 13]. 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.

Rather than producing a fixed or automatically selected visualization, Relo provides for **interactive exploration** of the currently selected code in its interface. It allows a developer to expand the visualization interactively, using simple controls on the visualization itself. Thus developers can avoid clutter that would otherwise hinder the comprehension activity. Prior work on program comprehension [11, 14, 15, 18] has shown that while developers often examine *small* programs systematically and exhaustively, *large* codebases are not explored that way; instead, developers follow a bottom-up exploration strategy, examining only the artifacts they think they need. Based on this observation, 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 further supports comprehension of this focused code by providing a **graphical presentation**, similar to UML class diagrams. Professional developers examining an unfamiliar implementation have problems keeping oriented and maintaining the layout of visible code elements [1]. Relo therefore 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.

The use of visualizations to help navigate and comprehend large information spaces typically results in usability issues [3, 4]. Relo has therefore been designed to support a number of common use cases that involve program understanding and also manages the size of the diagram being shown. Relo can automatically do simple searches, through the code graph, adding or removing code artifacts. A Relo visualization can also be *linked* to other tools in the development environment, so that exploration through a text editor is reflected automatically in the visualization, helping to preserve the user's context even when the user isn't working directly with Relo.

## 2. Previous Work

The strengths of Relo come from providing an intuitive interface for an incremental exploration of large projects, in a manner directed by the user. 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 [5] brings multiple relationships, like inheritance and method calls, into a single tree-view. It lets developers perform queries on nodes in an as-needed manner, and then uses query results to populate children of the node. As a result, JQuery's tree view represents 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. This merging of relationships puts an extra burden on developers to check the validity of relationships, such as verifying whether a class is inside a parent package. Relo provides support for multiple relationships by using diagrammatic constraints, such as containment or left-to-right ordering, to represent the different relationships.

A number of visualization approaches to program comprehension have focused on display capabilities and therefore limit the user's exploration needs. Reiss' FIELD system [12] used graphical widgets but each view supported user-directed exploration across one relationship only. SHriMP Views [16] provides fish-eye-lens distortion, graph visualization, and zooming in on targeted pieces of code. Nodes are expanded and contracted only along the containment axis, requiring all siblings to be shown when a single node is requested to be shown. Its approach of only expanding nodes along the containment axis even when the user is interested in a different relation tends to overwhelm [17]. 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 which relationships and nodes to show, it uses global filters instead of allowing users to browse interactively [19]. Since the global filters do not support the addition/removal of items as part of the user's interaction with the visualization, it becomes difficult for developers to focus on inheritance in some parts of the visualization and method calls in other parts. Another visualization approach, the TkSee Visualizer [20] supports users performing queries to build a visualization for graphical exploration and displays relationships using a radial layout. The system lets users perform a global query to specify items they want to see, but does not let them add, remove, or zoom-in/out of specific items, and thefore like SHriMP it does not allow the user to focus on different parts of the visualization in different places. Relo not only uses visual constraints to present nodes in expected locations, but also allows users to focus on relevant items.

Compared to these techniques, Relo takes a hybrid approach: It leverages user-directedness in reducing cognitive overhead from viewing multiple elements, and uses a graph-based view with automatic layout for placing nodes and children in predictable locations. Relo uses direct-manipulation browsing techniques, like navigation buds (described later) for users to implicitly query and build the relevant graphs. Further, Relo allows users to have fine-grained control in adding or removing single items. Relo visualizations are similar to those proposed by researchers studying navigation behaviors of programmers in traditional IDE's [6], but Relo also provides for incremental exploration in building the visualization.

A number of commercial tools, such as Rational Rose, TogetherJ, and Fujaba, can be used for building graphical diagrams of code, but they do not provide exploration capabilities. Instead of supporting code understanding, these tools support communication of already-understood code. In other words, the tools are aimed at developers who already understand the code, allowing them to create diagrams as documentation.

## 3. Walkthrough

We illustrate how Relo would be used by a developer for a typical comprehension task. For this example, we use a task similar to that used by JQuery [5]. The task involves a developer working with the `JHotDraw` 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 under-

stand how to manipulate them. In attempting this task, he will likely take a few steps:

1. Find a class implementing figures.
2. Understand it by examining 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 it to duplicate relevant functionality.

A developer following the above steps using a traditional IDE 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. At Step 4, the developer would need to examine the callers of a method which manipulates figures, and would at this point have difficulties in keeping track of the various examined code artifacts and the multiple relationships connecting them – 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. Selecting the class, and opening it in Relo, would give Figure 1.

Figure 1 shows that the class has 15 members, and the developer clicking on the menu sees a list of the members. Considering the method `basicMoveBy` as interesting, he clicks on the method name in the menu and thereby adds the method to the diagram. Once it has been added, the developer clicks on `EllipseFigure`, and is presented with a navigation bud indicating the class inherits from another class (shown in Figure 2). The developer clicks on this button to continue his exploration upwards showing superclasses (as in Figure 3), to find a relevant base class.

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. He then selects `addFigureChangeListener` method for expansion as part of the general framework for manipulating figures.

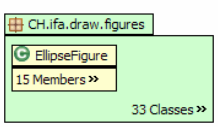


**Figure 1. Relo started by opening `EllipseFigure`**

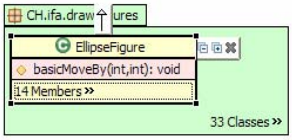


**Figure 2. Developers adds a method and clicking on class to show navigation buds**

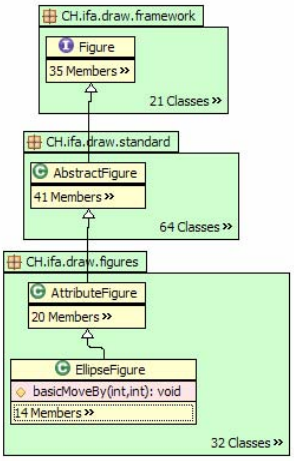


**Figure 3. After clicking on the inheritance navigation buds**

The developer is presented with Figure 4, 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 navigation bud. Relo continues to build the graph (shown in Figure 5), and has begun to act as both a call-hierarchy browser and an inheritance-hierarchy browser.

Once presented with Figure 5, the developer can select classes that manipulate figures, and does not have to remember the inheritance, containment, and method call relationships. The developer can now build a larger visualization or choose to refine the diagram, so that the visualization helps in his understanding of the underlying code base.

## 4. Presentation

Relo has been designed to support comprehension of a set of nodes by minimizing cognitive overhead. It


**Figure 4. Expanding the class `AbstractFigure` and the method `addFigureChangeListener`**

does this by managing the graphical presentation of nodes to minimize the amount of information displayed and to show nodes in expected locations.

### 4.1. Minimizing details

In order to minimize overhead on developers, Relo visualizations default to showing as little information as possible for every element presented. A particular element being displayed will therefore not display its container elements until explicitly requested. If an element is displayed without its parent, the element's name is shown fully qualified, i.e. a Java method appearing by itself is prefixed by its containing package and class.

Relo uses automatically triggered navigation services to add obvious relationships to the graph. One service tries to reduce the amount of information shown by adding the container element when there are multiple siblings shown. Another service automatically adds inheritance relationships so as to allow for easier visual grouping and therefore comprehension.

### 4.2. Directional Contraints

Relo uses topological constraints wherever possible to show nodes in predictable 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.

The automated layout of elements only happens for new elements in order for developers to have the nodes at expected locations. Selected nodes have their position fixed, while any elements that are added, removed, or moved by the system are performed using animation so that the developers do not lose context while working. Relo uses a force-directed approach for laying out code elements, in order to have a fluid automatic layout of the graph. Beyond basic node attraction and repulsion forces, support for directed constraints and graph containment are added to the layout engine. Previously positioned nodes are only moved if other nodes force them to be moved.

## 5. Incremental Exploration

Relo provides support for interactively exploring and effectively focusing on code elements by providing navigation buds to following relationships. At the same time, Relo also builds on these capabilities to support linking to the other tools in the development environment and provides for simple graph searches.

### 5.1. Navigation Buds

Developers browse the code in Relo diagrams by using "navigation buds" to navigate and extend the visualization with simple clicks. Navigation buds are context-sensitive buttons and are visible on the currently selected code element. Clicking on them makes the visualization grow by showing more items having the associated relationship.They provide the primary means of navigating through Relo (instead of the property dialogs or context menus used by most visualization tools). For example, when a class is selected (Figure 2), it sprouts navigation buds for different relationships that can be followed from the class (extends, extended-by). Navigation buds are only shown on the most common relationships (listed in Table 1), and only when clicking on them 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 navigation bud (as in Figure 2).

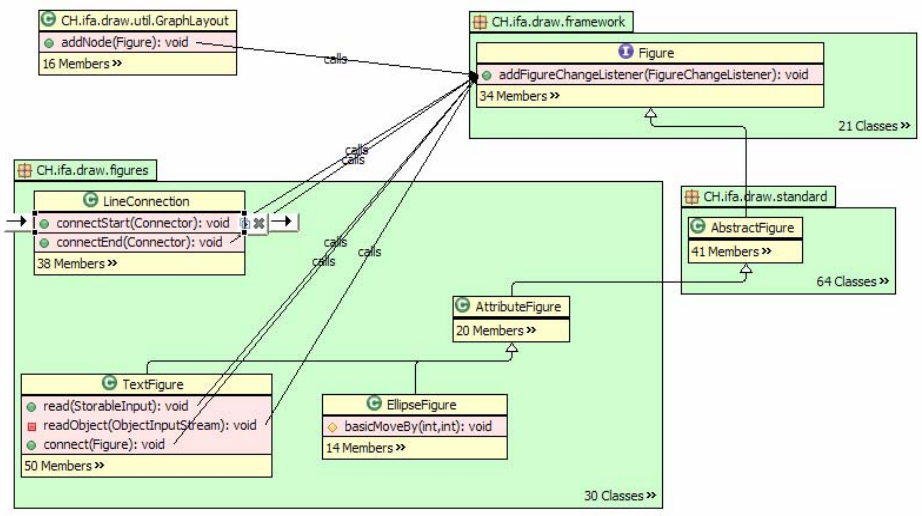


**Figure 5. Asking for callers of `addFigureChangeListener`**

Developers can semantically zoom-in by clicking on a special type of non-relationship navigation bud that expands ('+') and show more details. Expanding a class, first shows public members; expanding it again shows all members. Expanding a method shows the method body in a text editor. Developers can also collapse code artifacts by clicking on the '-' navigation bud, and can selectively eliminate artifacts by clicking on the 'x' navigation bud.

Another navigation bud allows the developer to reduce clutter by grouping items. For sibling nodes, this means first adding a containing element, and then packing the elements vertically next to each other (such as the default layout of methods in a class). A navigation bud present on grouped nodes also allows them to be 'broken' and undo the groupings.

### 5.2. Linked exploration

Relo provides support for developers wanting to use the tools provided by the traditional environment (IDE), but still get the benefit of an incremental visual exploratory environment. To do this, 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.

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.

After generating a diagram from the history, the developer can either use Relo to explore further, or continue with the traditional IDE views, in which case Relo actively mirrors the navigation. Relo will continue in a linked mode and will update the visualization to help provide context to the developer.

### 5.3. AutoBrowsing

Relo includes an *autobrowse* feature which helps a developer wanting to understand the relationships between different items in his visualization. Autobrowse locates and shows short paths of relationships between 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 paths of the same length between the selected nodes. Developers can repeat autobrowse to add longer paths.

## 6. Managing diagram growth

To ensure scalability of the interface Relo provides developers with simple interface features to manage the diagram growth. 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, as described in section 4.1, these items are automatically grouped using one of a few simple heuristics built as services. By default, grouping is based on access (public, protected, private), followed by members being grouped by name similarity. The goal of the automatic grouping is to get the number of ele-

ments at any level down to less than 10 elements, and Relo therefore uses different types of the available grouping in turn to reduce the number of shown items.

Beyond grouping, Relo also tries to limit the number of items added during exploration. 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 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 on clicking the button [2]. 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.

Developers can rapidly remove multiple code elements from the visualization by using the mouse to select multiple items and clicking on the removal navigation bud. 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 (section 5.2). The developer can also manually enable or disable auto-remove at any time.

## 7. Implementation

Since professional developers are familiar with the tools they already use, we built Relo on top of Eclipse, a widely-used Java IDE. 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 translates the underlying codebase into a triples database. This triples database is based on the W3C standard RDF [8] and therefore allows for easier extension of Relo to other languages and domains. All relationships are represented in the form <*source, relationshipType, destination*>. and a mapping engine connects the Eclipse builder framework, the Eclipse user-interface, and the relevant relationships in the database. Thus supporting any language requires adding the relationships to the database and providing the mappings to and from the various Eclipse objects.

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 1. 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 [13], 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.

**Table 1: Java Relationship model.**

| Nav Bud | Ctx Mnu | Relationship | From | To |
|---|---|---|---|---|
| ✔ | ✔ [1] | Inheritance | Class | Class |
| ✔ | ✔ | Method override | Method | Method |
| | ✔ | Field access | Method | Field |
| | ✔ | Field modify | Method | Field |
| ✔ | ✔ | Field type | Field | Class |
| ✔ | ✔ | Containment | Package | Class |
| | | | Class | Class |
| | | | | Method |
| | | | | Field |
| | ✔ | Method param. | Method | Class |
| ✔ | ✔ | Method calls | Method | Method |

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.

## 8. Evaluation

In order to evaluate the strengths and weaknesses of Relo, we conducted a preliminary study of Relo. The primary goal was to get preliminary data about 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.

### 8.1. Method

In order to closely resemble traditional large software development projects, we selected the LAPIS project [7], consisting of over 150,000 lines of code[2]. While larger projects do exist, most strengths and weaknesses of Relo were expected to be found in the

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

[2] Size of code base was measured using the command 'wc'.

codebase. We recruited 9 developers with over a year of experience using the IDE. Study participants had an average of 6.75 years experience with Java. These study participants had not used Relo, or looked at the code of LAPIS 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. The second task was fixing a bug (a failed unit test), and the third task was a minor user-interface feature addition. The tasks were for the LAPIS[3] project, and were selected from the primary developer's project-notes. They were then filtered to remove platform specific tasks and bugs requiring multiple executions of LAPIS to reproduce. In all three 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.

Participants used a think-aloud protocol [9, 10], and their actions were recorded by screen capture software and event logging. The study concluded with a questionnaire and a short semi-structured interview.

## 8.2. 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 9 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). 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 [17]. By contrast, participants did not find Relo to be overwhelming (2.6 on a 7-point Likert scale[4] 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), and did not feel helpless with manipulation of the interface. 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.

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 and lack of undo) would sometimes drive developers back to the JDT, they would keep drifting back into Relo to mitigate the task complexity. Participants spent an average of 56% of their time using the Relo interface.

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 runtime concepts in the visualization. Relo diagrams are closer to UML class

[3] The primary developer of LAPIS is also an author of this paper.

[4] In the Likert scale, 1 represented 'strong disagreement', 3 represented 'disagreement' 4 represented 'neutral', 5 represented 'agreement' and 7 represented 'strong agreement'.

diagrams than other interaction diagrams like sequence diagrams. In some cases, this results in Relo visualizations not being very helpful.

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

## 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. Relo provides this support by providing an incremental interactive exploration environment, and displays the shown nodes using visual constraints to show nodes in expected locations. Relo has been designed to provide support for a number of common use-cases and also provides support to ensure that the diagram size is kept under control. 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 freely available from **http://relo.csail.mit.edu**

## Acknowledgements

We would like to thank Daniel Jackson, Mike Ernst, David Huynh, Derek Rayside, and the study participant for comments and help with this research. This work was supported by the MIT Oxygen project.

## References

[1] R. DeLine, A. Khella, M. Czerwinski, G. Robertson, "Towards understanding programs through wear-based filtering". ACM SoftVis 2005.

[2] K. Doan, C. Plaisant, and B. Shneiderman, "Query Previews in Networked Information Systems". IEEE ADL 1996.

[3] N. Gershon, S.G. Eick, "Guest Editors' Introduction: Scaling to New Heights", IEEE Comp. Graphics and Applications, 18(4):16-17, 1998.

[4] I. Herman, G. Melancon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey", IEEE Trans. on Visualization and Computer Graphics, 6(1):24-43, 2000.

[5] D. Janzen and K. De Volder, "Navigating and Querying Code Without Getting Lost", AOSD 2003.

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

[7] The LAPIS Project, User Interface Design Group, MIT Computer Science and Artificial Intelligence Laboratory. http://groups.csail.mit.edu/uid/lapis/.

[8] 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.

[9] G. Lindgaard, "Usability Testing and System Evaluation: A Guide for Designing Useful Computer Systems", Chapman and Hall, 1994.

[10] J. Nielsen, "Usability Engineering", pp 195-198, Academic Press, 1993.

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

[12] S. Reiss, "Visualization for Software Engineering – Programming Environments", Chapter 18, pages 259-276, in "Software Visualization", ed. Stasko et al., MIT Press, 1998.

[13] M.P. Robillard , G.C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies", ICSE 2002.

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

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

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

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

[18] M.-A. Storey, F. Fracchia, and H. Muller, "Cognitive design elements to support the construction of a mental model during software visualization". IWPC 1997.

[19] J. Teevan, C. Alvarado, M.S. Ackerman, and D.R. Karger, "The perfect search engine is not enough: a study of orienteering behavior in directed search". CHI 2004.

[20] L. Wang."Animated Exploring of Huge Software Systems", MS Thesis, School of Info. Tech. and Engg., University of Ottawa, 2002.