A Tool for Visual Understanding of Source Code Dependencies

Martin Pinzger, Katja Gräfenhain, Patrick Knab, and Harald C. Gall
Department of Informatics, University of Zurich, Switzerland
{pinzger,graefenhain,knab,gall}@ifi.uzh.ch

Abstract

Many program comprehension tools use graphs to visualize and analyze source code. The main issue is that existing approaches create graphs overloaded with too much information. Graphs contain hundreds of nodes and even more edges that cross each other. Understanding these graphs and using them for a given program comprehension task is tedious, and in the worst case developers stop using the tools. In this paper we present `DA4Java`*, a graph-based approach for visualizing and analyzing static dependencies between Java source code entities. The main contribution of* `DA4Java` *is a set of features to incrementally compose graphs and remove irrelevant nodes and edges from graphs. This leads to graphs that contain significantly fewer nodes and edges and need less effort to understand.*

1. Introduction

Program comprehension is a necessary step in achieving objectives, such as fixing errors, changing or adding features, or improving code and design [13]. Program comprehension is a major cost factor in developing and maintaining software systems. Vendors of integrated development environments, *e.g.*,, Eclipse and Microsoft Visual Studio, have built in search functionality, class and call hierarchy browsing, cross-reference browsing, etc. which help in basic program comprehension tasks. However, they lack adequate visual support for more advanced program comprehension tasks. Providing such support has been a key objective of research and tool vendors.

Several approaches and tools have been developed, for example, Rigi [7], Creole[1], CodeCrawler [4], or `Imagix-4D`[2]. Most of these tools use graph-based visualizations where nodes represent source code entities, such as packages, classes, methods, and fields. Edges denote dependency relationships between them, such as class inheritance/subtyping, method calls, and field accesses. Typically, these tools follow the extract-abstract-view metaphor as described by Ebert *et al.* [2]. They first load all the information into the graph which then is queried, filtered, and edited by the user. For instance, `Creole` starts with an overview-graph on the package level whose package nodes can then be expanded to analyze source code details. This approach follows the mantra presented by [11] which is useful to get an overview of the implementation, however, it bears the problem that graphs get cluttered with irrelevant details.

[1]`http://www.thechiselgroup.org/creole`
[2]`http://www.imagix.com`

In this paper we focus on Java source code and present `DA4Java` (Dependency Analyzer for Java), a graph-based visualization approach for understanding static dependencies between source code entities. The main objective of `DA4Java` is to reduce the *cognitive effort* to understand dependency graphs. Large dependency graphs with many nodes and edges that overlap each other are usually not aesthetic and require more effort to understand. Our approach supports the creation of condensed, aesthetic graphs by showing only the information *relevant* to solve a given program comprehension task. For this, `DA4Java` uses nested graphs and a set of features to add and filter nodes and edges. The adding features allow the user to incrementally compose the dependency graph. For instance, the 'Add callers' feature adds methods that call a selected entity and only the corresponding method calls to the graph. Features for filtering are used to remove irrelevant information and stay focussed on the program comprehension task.

We demonstrate these features of `DA4Java` with a number of examples from the source code of the Eclipse plugin `JDT Debug`. The examples show, that our approach allows the user to create condensed views on Java source code. They enable the understanding of the big picture by hiding details and the understanding of details by hiding the irrelevant parts of the system.

The remainder of the paper is structured as follows: In the next section we present related work. In Section 3 we motivate our approach with an example. The `DA4Java` approach is presented in Section 4. In Section 5 we draw the conclusions and outline future work.

2. Related Work

Most of the existing program comprehension tools are geared to a top-down approach, and lack bottom-up exploration. Von Mayrhauser *et al.* found that program understanding is not unidirectional, *i.e.*, top-down or bottom-up exclusively [16]. They present a meta model that integrates both, the top-down model of Soloway *et al.* [12] and the bottom-up model of Pennington [9]. This calls for a better integration of the two approaches. In this regard, `DA4Java` in comparison with other tools, provides a significant improvement.

In [11] Shneiderman *et al.* discuss the visual information seeking mantra: "Overview first, zoom and filter, then details-on-demand." In this paper we demonstrate that this mantra is not always the best way to go for several general program comprehension tasks. For example in tasks, in which the analysis concerns a particular method or class, it is more efficient to use a bottom-up approach and stepwise add more information to the graph than to start top-down and filter all the details. A combination of both directions is preferred and supported by `DA4Java`.

The following source code visualization tools and techniques are most related to our approach. `Rigi` is a tool that concentrates on the mastery of structural complexity of large systems with graph-based visualizations [7]. It follows mainly a top-down analysis approach and uses Simple Hierarchical Multiperspective views (SHriMPs). They reduce clutter while preserving the big picture with multiple views. `Rigi` provides a set of filters via edge and node types, or incoming and outgoing dependency relationships. `Shrimp` [14] is a further development of `Rigi`. It introduces the concept of nested interchangeable views to allow a user to explore multiple perspectives of information at different levels of abstraction. `Creole` is an Eclipse plugin based on `Shrimp`. The main difference to `DA4Java` is that these tools lack features for the incremental composition of graphs.

IBM's `Structural Analysis for Java`^{TM} (SA4J) is a tool to analyze structural dependencies of Java applications and detect "anti-patterns."[3] SA4J provides similar features in the exploration view as `DA4Java`. Sophisticated composition and filtering features such as the filter for node internal dependencies are missing. Similar to `Imagix-4D` it uses flat graphs and does provide only limited support for a combined top-down and bottom-up program comprehension approach.

Lanza introduced `CodeCrawler`, a tool that uses 'Polymetric Views' to display various aspects of object-oriented software systems [4]. Its focus is on visualizing the overall structure of a system, to asses, for example, design violations. The focus on the big picture has the tendency to lead to complex views when trying to get information for smaller units. In [5] Lungu and Lanza presented `Softwarenaut`. This is a tool used for top-down exploration of large software systems. Its combination of a detail view and overview view is interesting because the overview view limits cluttering substantially without compromising the big picture. Features for the incremental composition of graphs and filtering are not provided by this approach.

`Source Viewer 3D` [6] is a tool that uses a 3D representation to visualize source code. It is a further development of the `SeeSoft` [3] metaphor. They also improved the `SeeSoft` metaphor with regard to the optimization of simultaneously presenting as much information as possible while avoiding information overload. The `SeeSoft` metaphor is different from our approach, but `Source Viewer 3D` shows, that there are other possibilities to improve the expressiveness of visualizations.

3. Motivating example

Consider the following program comprehension scenario: the developers of the `JDT Debug` plugin want to refactor the package `breakpoints`. A first step towards this refactoring is to find out which other packages, classes, and methods will be affected by these modifications. One way to answer this question is to analyze the incoming method calls of package `breakpoints`.

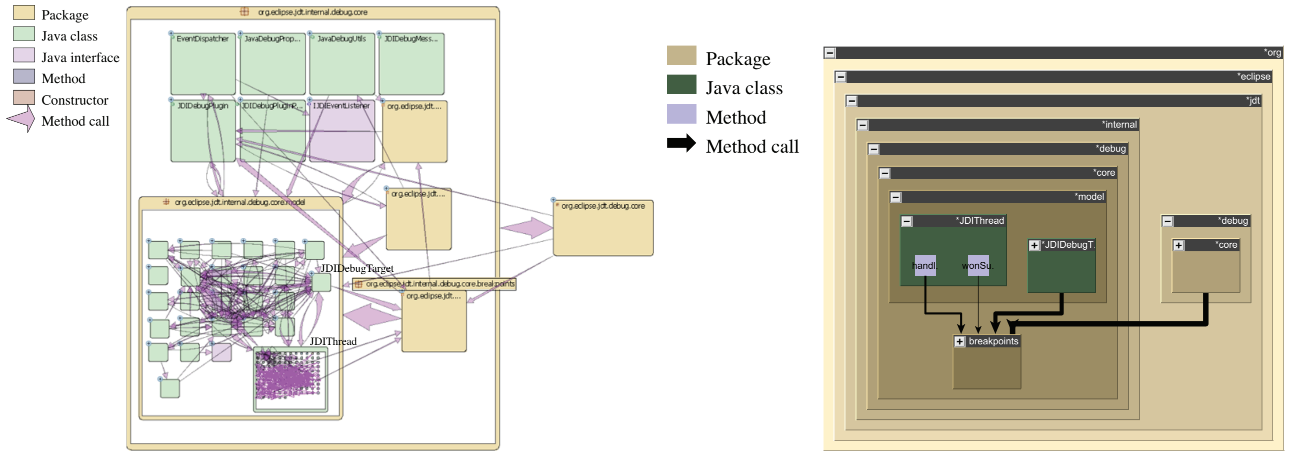
Visualizing the dependencies with `Creole` we get the graph depicted in Figure 1a. The graph is cluttered with nodes and edges and the user-effort to understand the graph and find the answer to the question is high.

The graph depicted in Figure 1b was created with `DA4Java`. It shows the same level of detail as the graph created with `Creole`. In contrast to the `Creole` graph it contains *only* the nodes and edges that are needed to answer the question, namely, the entities that call methods of package `breakpoints`. The number of nodes is reduced from 41 to 14 (not considering the nodes representing the members of class `JDIThread`). The number of edges is even smaller and there are no edge-crossings in the `DA4Java` graph. The effort to understand this graph and to answer the question is reduced significantly.

We performed this and other general program comprehension tasks, such as presented by Pacione *et al.* [8] (reduced to static source code analysis), with `Creole`, `Imagix-4D`, and `Rigi`. The main problem with these three tools turned out to be the lack of features to:

- incrementally add entities and relationships to the graph that are relevant for solving program comprehension tasks.
- remove nodes and edges from the graph that are irrelevant in context of a program comprehension task.

[3]`http://www.alphaworks.ibm.com/tech/sa4j`



(a) Creole (b) DA4Java

Figure 1. Packages, classes, and methods using the package `breakpoints` visualized with `Creole` and `DA4Java`.

In summary, resulting graphs were cluttered with irrelevant nodes and edges that were more effort to understand than the graphs created with `DA4Java`. In the following section we introduce `DA4Java` and present its features to incrementally compose graphs and remove irrelevant information.

4. Dependency Analyzer for Java

In [15] von Mayrhauser and Vans stated "tools must quickly and succinctly answer programmer questions, extract information without extraneous clutter, and represent the information at the level at which the programmer currently thinks." These are the key requirements according to which we developed `DA4Java`. For the description of our approach, we first present the basic visualization technique of `DA4Java`. After this follows the presentation of the main contribution of this paper, *i.e.*, the set of features to compose and filter graphs.

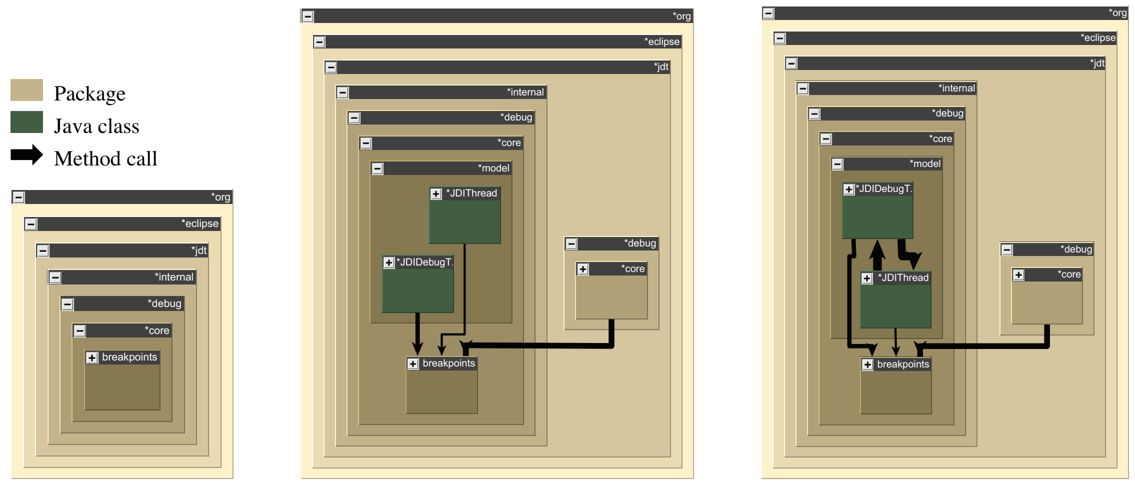
4.1. Visualization technique

`DA4Java` uses nested graphs to represent source code information at the various levels of abstraction. The model of the graph corresponds to the FAMIX meta model [1]. Nodes in the graph represent source code entities which are packages, classes, methods, and fields. Edges in the graph represent static dependencies between source code entities which are class inheritance/subtyping, method calls, and field accesses. In the remainder of the paper we focus on method calls to demonstrate our approach.

Nested graphs reflect the hierarchic structure of Java programs. Packages contain sub-packages and classes which contain the class members (*e.g.*, inner classes, methods, and fields). Nested graphs allow the user to combine top-down and bottom-up source code analysis. `DA4Java` supports existing cognition models for program comprehension: top-down building of mental models [12] and bottom-up building of program and situational models [9]. The user expands package or class nodes to view more implementation details and collapse nodes to remove details.

Figure 1b depicts an example graph showing the hierarchic structure of classes and packages using package `breakpoints`. The button in the top-left corner of nodes is used to expand or collapse nodes. Instead of visualizing each single dependency relationship, `DA4Java` aggregates edges between nodes. The width of an edge represents the number of aggregated low-level edges (*e.g.*, method calls). For example, all method calls in Figure 1b from methods of package `debug.core` to methods of package `break-points` are shown as one strong arrow. This reduces the number of edges in a graph and leads a more aesthetic layout.

The cognitive effort to understand graphs needs to be reasonable to quickly and succinctly answer programmer questions. In other words, the visualization needs to present the nodes and edges for solving the program comprehension task without unnecessary noise. Nested graphs and edge aggregation are two basic techniques to filter noise in hierarchic data, such as source code. They have been used already by related tools, such as `Creole`. Nevertheless, graphs can get complex, when analyzing large systems and digging into source code details as demonstrated by the example in Figure 1a. To keep the amount of information in graphs manageable, `DA4Java` provides a set of features to incrementally compose graphs, and filter irrelevant nodes



(a) Added package `breakpoints`

(b) Added callers of `breakpoints`

(c) Added method calls between the classes `JDIDebugTarget` and `JDIThread`

Figure 2. Example of analyzing the incoming method calls of package `breakpoints`.

and edges. Next, we describe these features and demonstrate them with the `breakpoints` example.

4.2. Features to add information to graphs

`DA4Java` supports two ways of adding information to a graph. The first way is to select the entities in an Eclipse view such as the Package Explorer and add them to the graph. The second way is to select nodes in the graph and add entities and relationships via their incoming or outgoing dependency relationships. For the explanation of these features we use the examples depicted in Figure 2. The features are:

Add entities: Adds selected entities, their parents, and descendants to the graph. The selection is done in the Eclipse Package Explorer or similar views. Method calls between added entities as well as between added entities and methods that are already contained in the graph are included as well. In the example, we selected the package `breakpoints` from the Package Explorer and added it to the graph. The resulting graph is depicted in Figure 2a. It shows the package `breakpoints` and all its parent packages.

Add callers: Adds methods to the graph that *call* the selected node. The corresponding method calls, parent packages, and classes of methods are also added to the graph. Nodes of different types can be selected in the `DA4Java` graph. If a package node is selected, methods that call any method of the selected package are added. If callee methods of the selected package are not present in the graph, they are added. In the example, we selected the node representing package `breakpoints` and added its callers. Figure 2b depicts the result. Two packages `debug.core` and `model` contain methods that call methods of package `breakpoints`.

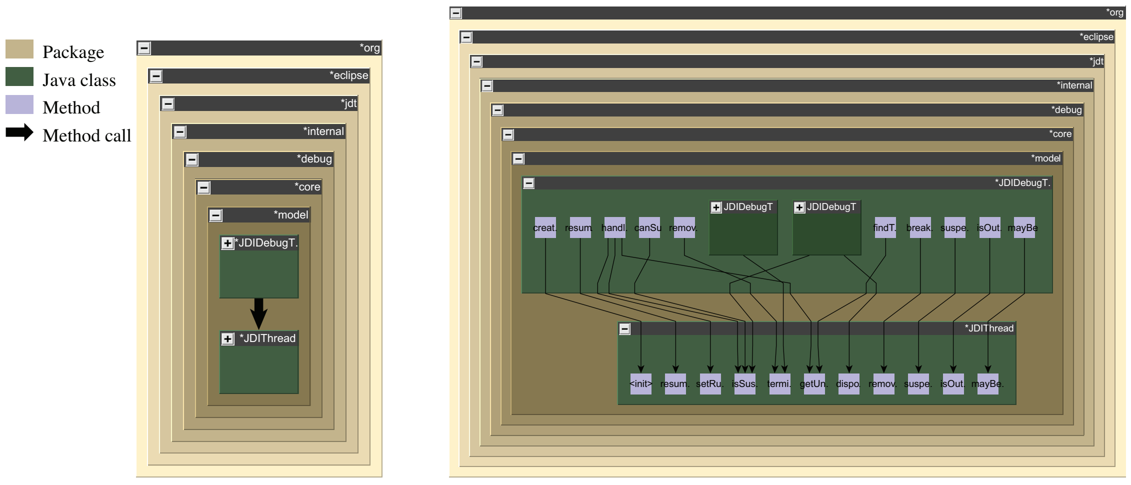
Add callees: Adds methods that are *called by* methods of the selected node to the graph. The corresponding method calls, parent packages, and classes of methods are added to the graph as well. Nodes of different types can be selected in the `DA4Java` graph. If a package node is selected, methods that are called by any method of the selected package are added. If caller methods of the selected package are not present in the graph, they are added.

Add calls between selected nodes: Given at least two selected nodes in the graph, this feature adds the method calls between these nodes. Nodes of different types can be selected in the `DA4Java` graph. For example, if two classes are selected the incoming and outgoing method calls between the methods of both classes are added. Methods that are not present in the graph but involved in method calls are added to the graph as well. In Figure 2c we expanded the package `model` and added the method calls between the two classes `JDIDebugTarget` and `JDIThread`. They are represented by the two edges between the class nodes.

4.3. Features to filter information from graphs

While adding information to the graph the number of nodes and edges in the graph increases until the graph becomes too complex and can hardly be grasped by the user. To re-focus on relevant source code entities and dependency relationships, `DA4Java` provides a number of features to filter nodes and edges from the graph. These are:

Keep callers and remove other nodes: Removes nodes that do *not call* a method of the selected node. The corresponding method calls are removed from the graph as well.



(a) Kept the callers of class `JDIThread` and removed other nodes and edges.

(b) Removed node internal dependencies from classes `JDIThread` and `JDIDebugTarget`.

Figure 3. Example of analyzing the method calls from `JDIDebugTarget` to `JDIThread`.

If a package or class is selected, `DA4Java` takes into account calls *to* methods of the selected package/class that are present in the graph. This feature allows the user to re-focus on the incoming method calls and involved entities of the selected node. For example, we applied this filter to the node `JDIThread` of the previous example graph depicted in Figure 2c. The result is depicted in Figure 3a. The only entity that calls methods of class `JDIThread` is the class `JDIDebugTarget`. All other nodes and edges were removed.

Keep callees and remove other nodes: Removes nodes that are *not called by* a method of the selected node. The corresponding method calls are also removed from the graph. If a package or class is selected, `DA4Java` takes into account calls *from* methods of the package/class that are present in the graph. This function allows the user to re-focus her analysis on outgoing method calls and involved entities of the selected node.

Remove non-dependent nodes: This feature combines the two previous features. It removes nodes that neither call nor are called by methods of the selected node. This feature allows the user to focus on the incoming and outgoing method calls and involved entities of the selected node.

Remove node internal dependencies: Removes internal method calls of a selected package or class node. Furthermore, child nodes with no calls to methods outside the selected package/class are also removed from the graph. This feature is used to focus the analysis on inter-node dependencies such as method calls between packages or classes. Applying this filter to the two classes `JDIDebugTarget` and `JDIThread` of our example, creates the graph depicted in Figure 3b. It shows only the nodes and edges of entities and dependencies that are involved in method calls from `JDIDebugTarget` to `JDIThread`. Other methods and internal method calls were removed.

Remove selected nodes/edges: Removes selected nodes and edges from the graph. If a package or class node is selected their descendant nodes, and incoming and out-going method calls are also filtered from the graph. If aggregated edges are selected underlying method calls are removed from the graph but not their source and target nodes.

Remove non-selected nodes/edges: Removes the non-selected nodes and edges from the graph. If nodes are selected only the edges between these nodes are kept. If edges are selected, only nodes that are a source or target node of the selected edges are kept. With this filter the user is able to focus the analysis on certain nodes/dependency relationships in the graph.

The same set of features, that `DA4Java` provides for method calls, are also provided for class inheritance/subtyping and field access dependencies.

4.4. Features to handle incomplete graphs

The main advantage of our approach is that the user is able to control the complexity of graphs and speed up program comprehension tasks. There is, however, also a drawback: the graph composed by the user may not represent all information and may give a wrong impression of the current implementation. For example, to simplify the graph a user filters a number of dependency relationships of package `breakpoints`. Because of the missing relationships an overall analysis of the dependency relationships of this package is not possible anymore. To alleviate this problem `DA4Java` provides two features:

Not all descendant nodes are present: A node label beginning with a '*' signals the user that not all descendant nodes of the corresponding node are present in the graph (*e.g.*, see Figure 3a). The 'Add entities' feature is used to add the missing descendants of this node and their dependency relationships.

Graph edit history: `DA4Java` keeps a history of executed add and filter features. For each history-entry the set of nodes and edges that were added or respectively removed from the graph are stored. Each executed add and filter feature can be undone in backwards order. In the other direction support for redo is also provided.

4.5. First Experiment with DA4Java

In an experiment with the source code of `JDT Debug` plugin of Eclipse we compared our approach with `Creole` and `Imagix-4D`. We performed two typical program comprehension tasks that were concerned with analyzing the dependencies of class `JavaWatchpoint`. For each tool, we assessed the user effort to create the graphs and we measured the size and complexity of resulting graphs. In both tasks, the creation of the graphs with `DA4Java` was straight forward. Furthermore, the graphs created with `DA4Java` contained significantly fewer nodes and edges and needed less effort to understand than the graphs created with `Creole` and `Imagix-4D`. For more details on the comparison we refer the reader to our technical report [10].

5. Conclusions

Visualizations generated by program comprehension tools still consist of graphs that contain hundreds of nodes and even more edges that cross each other. Understanding these graphs and using them for a given program comprehension task is tedious. In this paper, we proposed a graph-based approach called `DA4Java` for visualizing and analyzing Java source code. It consists of features to incrementally enrich graphs such as adding entities, callers, callees, and their call relationships. It further provides effective filtering features to keep only the interesting nodes and edges in the graph. Complex graphs are simplified by effective filter algorithms to provide a user with a quick comprehension path in large software systems.

As future work we foresee a controlled user experiment to do a detailed analysis of the features of our tool. That will give us input for fine-tuning our graph algorithms and the user interface.

References

[1] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, Software Composition Group, University of Berne, August 1999.

[2] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. Gupro - generic understanding of programs. *Electronic Notes in Theoretical Computer Science*, 72(2):59–68, 2002.

[3] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[4] M. Lanza. Codecrawler - polymetric views in action. In *Proceedings of the International Conference on Automated Software Engineering*, pages 394–395, Linz, Austria, 2004. IEEE Computer Society Press.

[5] M. Lungu and M. Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 351–354, Washington, DC, USA, 2006. IEEE Computer Society Press.

[6] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software visualization*, pages 27–36, New York, NY, USA, 2003. ACM Press.

[7] H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the International Conference on Software Engineering*, pages 80–86, Singapore, April 1988. IEEE Computer Society Press.

[8] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proceedings of the Working Conference on Reverse Engineering*, pages 70–79, Washington, DC, USA, 2004. IEEE Computer Society Press.

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

[10] M. Pinzger, K. Gräfenhain, P. Knab, and H. C. Gall. Incremental visual understanding of java source code. Technical report, University of Zurich, 2008.

[11] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, Washington, DC, USA, 1996. IEEE Computer Society Press.

[12] E. M. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[13] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Control*, 14(3):187–208, 2006.

[14] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: an interactive environment for information visualization and navigation. In *Extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM Press.

[15] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tools capabilities. In *Proceedings of the International Conference on Computer-Aided Software Engineering*, pages 230–239, Singapore, July 1993. IEEE Computer Society Press.

[16] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.