Incremental Visual Understanding of Java Source Code

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 Java source code. `DA4Java` provides features to stepwise add information to graphs as well as features to remove irrelevant nodes and edges from graphs.

In a case study with the `JDT Debug` plugin of Eclipse we compared our approach with `Creole` and `Imagix-4D`. We performed two typical program comprehension tasks and evaluated user effort to create the graphs for the comprehension tasks as well as size and complexity of the resulting graph. In both tasks, our approach outperformed `Creole` and `Imagix-4D`. Graphs created with `DA4Java` contained significantly fewer nodes and edges and needed less effort to understand.

1. Introduction

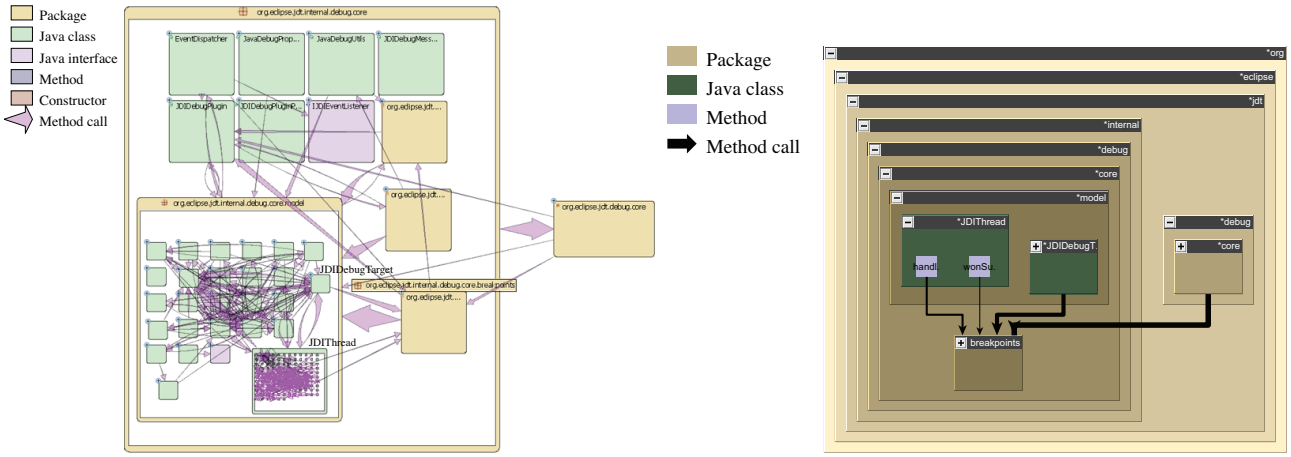
Program comprehension is a necessary step in achieving objectives such as fixing errors, changing or adding features, or improving code and design [13]. Program comprehension is a major cost factor in developing and maintaining software systems. Vendors of integrated development environments, for example, Eclipse and Microsoft Visual Studio, have built in features for speeding up program comprehension. They provide code search functionality, class and call hierarchy browsing, cross-reference browsing, etc. Though modern development environments have been improved towards program comprehension 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 [6], Creole[1], CodeCrawler [3], 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.* [1]. 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 then can 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. But it bears the problem that graphs get cluttered with irrelevant details that need to be filtered out by the user.

In this paper we focus on Java source code and present `DA4Java` (Dependency Analyzer for Java), a graph-based visualization approach for program comprehension. The main objective of `DA4Java` is to reduce the cognitive effort to understand graphs. Large 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 program comprehension task. To keep the size and complexity of graphs minimal, `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 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 at hand. Our approach enables the understanding of the big picture by hiding details as well as the understanding of details by hiding the irrelevant parts of the system.

We integrated `DA4Java` into Eclipse and evaluated it

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



(a) `Creole` (b) `DA4Java`

Figure 1. Packages, classes, and methods using the package `breakpoints`.

in a case study with the Eclipse plugin `JDT Debug`. The evaluation of our approach is based on the comparison of `DA4Java` with two existing program comprehension tools, `Creole` and `Imagix-4D`. We used the tools in two general program comprehension tasks concerning the analysis of static dependency relationships between Java packages and classes. The main criteria for the comparison were the user-effort to create and analyze the graph, and the size and complexity of the graph. In both tasks, our approach created graphs that were significantly less complex and easier to understand than the graphs created with `Creole` and `Imagix-4D`.

The remainder of the paper is structured as follows: In the next section we motivate our approach with an example. The `DA4Java` approach with its features to add and filter information to/from graphs is presented in Section 3. In Section 4 we compare our approach with `Creole` and `Imagix-4D`. Related work is presented in Section 5. In Section 6 we draw the conclusions and outline future work.

2. 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 got 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.

The main problem of existing program comprehension tools is the lack of features to control the amount of information visualized in graphs. This lowers their applicability for several program comprehension tasks such as presented by Pacione *et al.* [7]. This concerns:

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

3. Dependency Analyzer for Java

In [16] 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.

DA4Java uses nested graphs to represent source code information at the various levels of abstraction. 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.

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). According to this hierarchy package nodes contain sub-package and class nodes, which contain the nodes representing class members. These are the different abstraction levels that DA4Java supports.

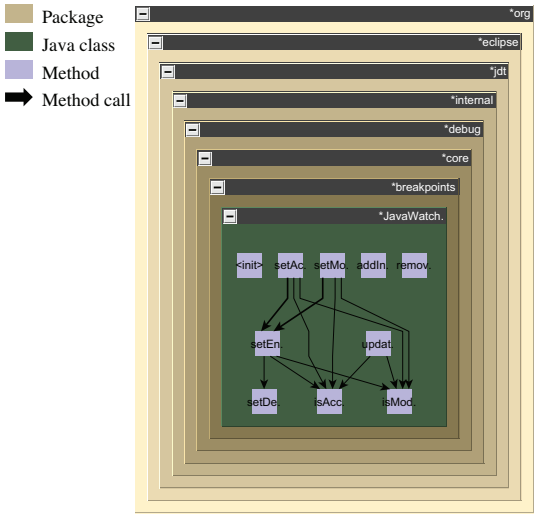
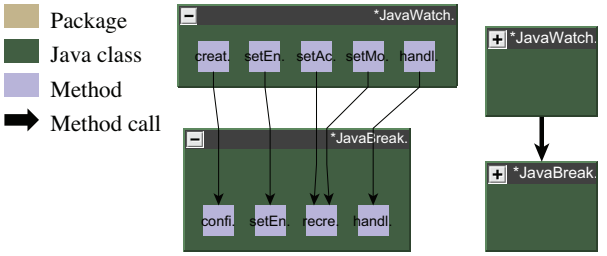


Figure 2. Example graph visualizing the parent packages and some methods of class JavaWatchpoint.

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 the building of mental models [12] and bottom-up the building of program and situational models [8]. The user expands package or class nodes to view more implementation details and collapses nodes to remove details. Figure 2 depicts an example graph showing the parent packages of class JavaWatchpoint and an excerpt of its methods. 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 to one edge. The width of an edge represents the number of aggregated low-level edges (*e.g.*, method calls). This reduces the number of edges in a graph and facilitates a more aesthetic layout. Figure 3 depicts an example of aggregating method calls between the two classes JavaWatchpoint and JavaBreakpoint.



(a) expanded method calls (b) aggregated

Figure 3. Example of aggregating method calls between JavaWatchpoint and JavaBreakpoints.

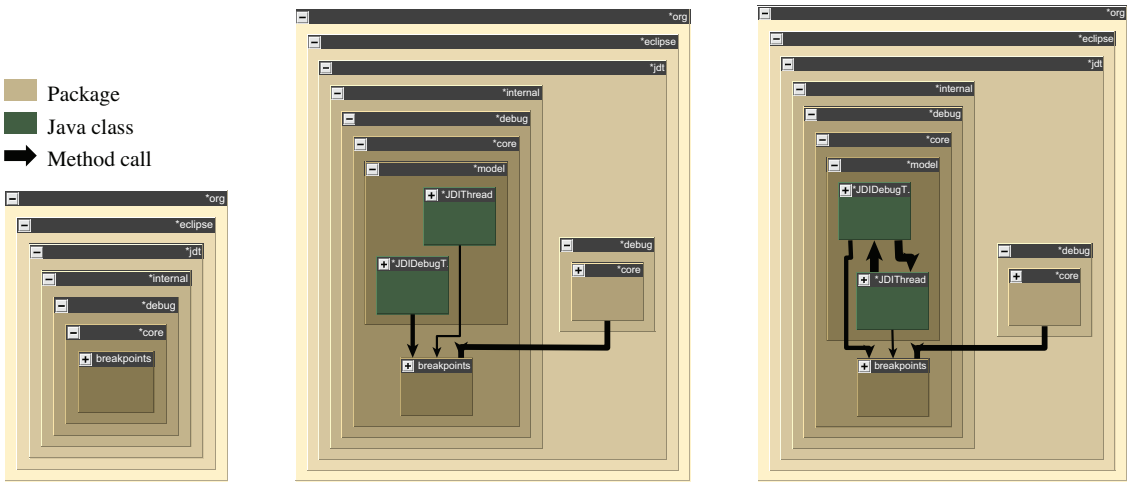
In Figure 3a both class nodes are expanded and five call edges are drawn between the methods. In Figure 3b both class nodes are collapsed and the five edges are aggregated to a single edge. When expanding the nodes it works the other way round, *i.e.*, the single edge is expanded to five edges. The edge aggregation for packages works in the same way.

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. DA4Java allows the user to create such graphs by providing a set of features to incrementally compose graphs, and filter irrelevant nodes and edges. The features to add information to the graph are presented in the following subsection.

3.1. 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 4. 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



(a) Added package `breakpoints`

(b) Added callers of `breakpoints`

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

Figure 4. Example of analyzing the incoming method calls of package `breakpoints`.

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 4a. 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 4b 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.

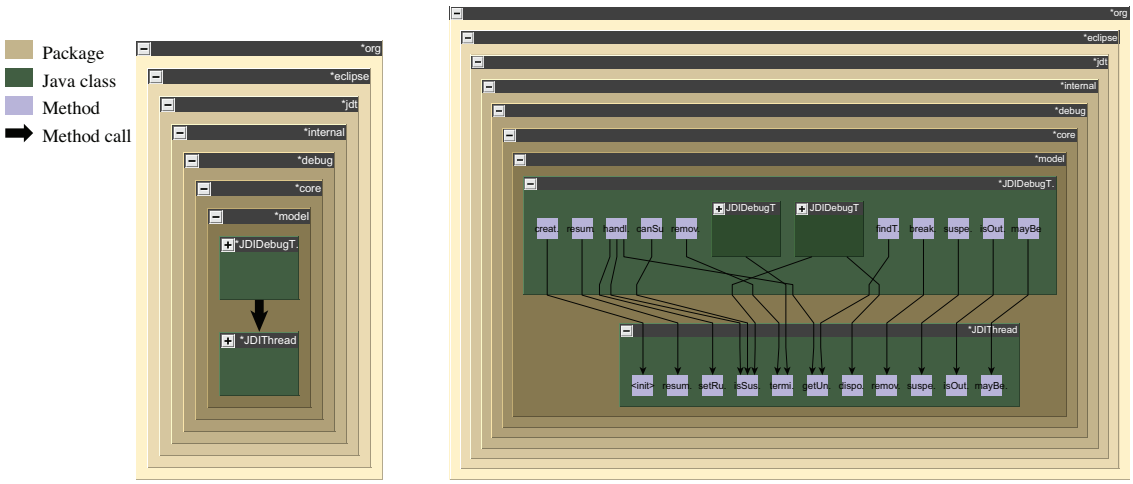
Add calls between selected nodes Given at least two selected nodes in the graph, this feature adds the method calls between these nodes. Nodes of different types can be selected in the `DA4Java` graph. For example, if two classes are selected the incoming and outgoing method calls between the methods 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 4c 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.

3.2. 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. 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 4c. The result is depicted in Figure 5a. 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.



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

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

Figure 5. Example of analyzing the method calls from `JDIDebugTarget` to `JDIThread`.

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 5b. 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.

3.3. 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. 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.

In the following we compare our approach with two existing source code analysis tools.

4. Comparison with `Creole` and `Imagix-4D`

`Creole` and `Imagix-4D` represent two cutting edge source code analysis tools that use graph-based visualizations. `Creole` (version 1.6.1) also uses nested graphs and edge aggregation. In contrast, `Imagix-4D` (version 6.2.0)

uses flat 2D and 3D graphs. Both tools can handle Java source code and we have several years of experience in using these tools for program comprehension tasks.

As case study we selected the `JDT Debug` plugin of the Eclipse project. The plugin provides the core functionality for managing breakpoints, inspecting variables, and evaluating expressions. Version 3.2.2 of the plugin comprises more than 37k lines of code. They are contained in 345 Java classes and 131 interface classes which are organized in 23 packages. For `Imagix-4D` we used the tool-internal parser to process the source code to a fact-repository. This repository is used by `Imagix-4D` for all its analysis features. `DA4Java` and `Creole` both are Eclipse plugins that use the Eclipse java development tools to parse the source code. `DA4Java` stores extracted facts into a Hibernate[3] mapped MySQL database. `Creole` keeps a fact repository in the main memory.

The comparison of the tools is based on the graphs that were created with each tool. We chose a general program comprehension activity which consisted of two subsequent tasks: 1) visualize packages and classes that depend on class `JavaWatchpoint`; 2) visualize the methods of class `JavaWatchpoint` that call methods of other packages and classes.

The criteria for the comparison are:

1. User-effort to create and analyze the graph. For each tool we describe the steps that we needed to create an adequate graph. With adequate we mean a graph that shows the mandatory information in an understandable way so that we were able to solve the given tasks. A detailed evaluation of the effort with a user study is subject to future work.

2. Size and complexity of graphs measured with the number of nodes and edges, and the number of edge crossings [10]. Graphs with a small number of nodes and edges are usually easier to understand than graphs with many nodes and edges. In an experiment Purchase *et al.* compared different graph layout algorithms considering several aesthetic criteria. Results showed that users preferred graphs with few edge crossings over graphs with many edge crossings [9].

4.1. Packages and classes depending on `JavaWatchpoint`

For solving this task, we visualized the packages and classes that depend on class `JavaWatchpoint` via incoming and outgoing method calls. With this task we demonstrate the features of `DA4Java` to incrementally compose a graph.

[3] `http://www.hibernate.org`

Effort estimation In the following we describe the steps to create and analyze the graph with each tool:

- `DA4Java`: We selected the class `JavaWatchpoint` in the Eclipse Package Explorer and added the class to the graph window. Next, we selected the node representing the class `JavaWatchpoint` and via 'Add callers' and 'Add callees' added dependent entities to the graph. No additional post-processing or layout of the graph was needed. The result is depicted in Figure 6a. The view clearly shows that `JavaWatchpoint` is a consumer of functionality of classes provided by the external package `com` and the internal package `model`. Within the package `breakpoints` `JavaWatchpoint` calls methods of the two classes `JavaLineBreakpoint` and `JavaBreakpoint`. The single caller of the methods of `JavaWatchpoint` are methods of package `debug`.

- `Creole`: We used the Package Explorer to select the two source code directories `jdi interfaces` and `model`. We dropped the selection on the button 'Package Dependencies via Method Calls'. We next expanded the node of package `breakpoints` and its incoming and outgoing edges that denote method calls. Each edge needed to be double clicked. Furthermore, we had to manually adjust the layout of the inner graphs of packages `debug.core` and `breakpoints` to find the entities that depend on class `JavaWatchpoint`. Zooming out of package nodes caused an automatic relayout of the inner graphs so that we had to redo the manual layout several times. The final graph is depicted in Figure 6b. The manual expansion of edges and the layout of inner graphs was tedious and time-consuming.

- `Imagix-4D`: We selected the 'Structure' mode and configured the view to show Java classes, interfaces, and method calls. For the layout we configured 'Compact' and 'FromRoots'. Next, we selected the class `JavaWatchpoint` from the Class Index window and added it to the graph. In the graph window we selected the node and added all classes via incoming and outgoing method call relationships. To reduce the complexity of the graph, we omitted package nodes and containment relationships between packages and classes. The `Imagix-4D` graph is depicted in Figure 6c. Similar to `DA4Java`, using the graph compose and filter functions of `Imagix-4D` such views can be created within a short time.

With `DA4Java` and `Imagix-4D` less effort was needed to create the graph. The main problem of `Creole` is the lack of facilities to *incrementally* build the graph. The user

needs to start his analysis top-down and dig into the details. Adequate filters to focus on the entities of interest and remove irrelevant entities and relationships from graphs are not provided by Creole. This leads to more complex graphs that need more effort to handle and understand.

Graph size and aesthetic The following table represents the measures for each graph.

	#nodes	#edges	#edge crossings
DA4Java	13	5	0
Creole	28	> 50	> 200
Imagix-4D	14	≈ 38	≈ 100

DA4Java outperformed both tools, Creole and Imagix-4D. The graph created with Creole contains several packages and classes that neither call nor are called by methods of class JavaWatchpoint. Furthermore, while DA4Java draws only edges that represent incoming and outgoing methods calls of class JavaWatchpoint, Creole represents *all* call relationships between visible nodes. With the filter functions provided by Creole we were not able to filter these nodes and edges from the graph. This led to a graph that contains 15 additional nodes, over 50 edges, and more than 200 edge crossings.

In contrast to DA4Java and Creole, Imagix-4D uses flat directed graphs for the visualization of source code. The downside of this visualization is the missing navigation facility for a combined top-down and bottom-up analysis by expanding and collapsing nodes. The graph of Imagix-4D contains significantly more edges and edge crossings than the graph of DA4Java. This is due to the fact that Imagix-4D shows *all* method calls between visible entities. Adequate functions to filter method calls between the caller methods and method calls between the callee methods are not provided by Imagix-4D.

4.2. Methods of JavaWatchpoint involved in outgoing dependencies

This task is a follow-up to the previous task. For its solution, we visualized the methods of class JavaWatchpoint that are involved in the outgoing call dependencies to other packages and classes. This task demonstrates the filter capabilities of DA4Java.

Effort estimation In the following we describe the steps to create and analyze the graph with each tool:

- DA4Java: We continued with the previous dependency graph (see Figure 6a) and applied the function 'Keep callees' to JavaWatchpoint. Next, we expanded the node representing class JavaWatchpoint and applied the function 'Filter internal dependencies' to filter the class internal method calls. This

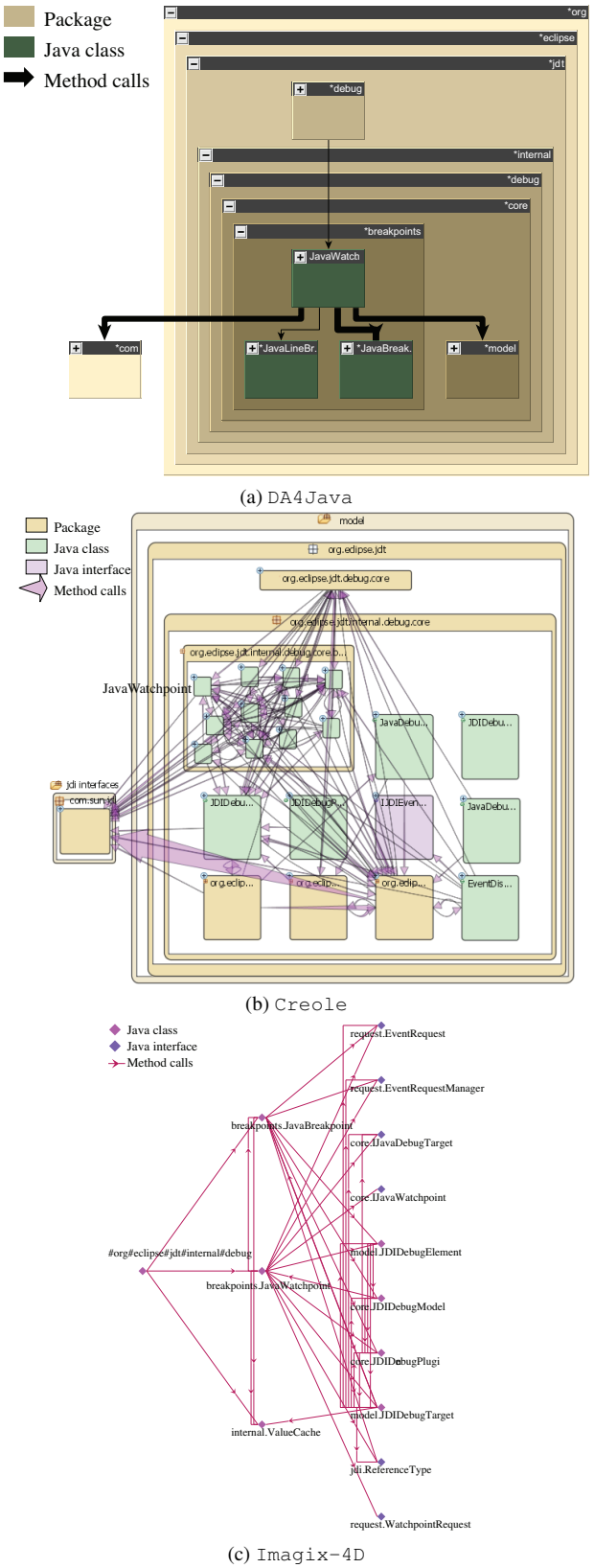


Figure 6. Task 1: Packages and classes depending on JavaWatchpoint.

function also filters child nodes that do not call or are called by methods of other classes. Figure 7a depicts the graph after applying the two filters. It shows 11 methods of `JavaWatchpoint` call methods of other packages and classes. The width of the edges shows that 4 methods in the center of the graph frequently call external methods.

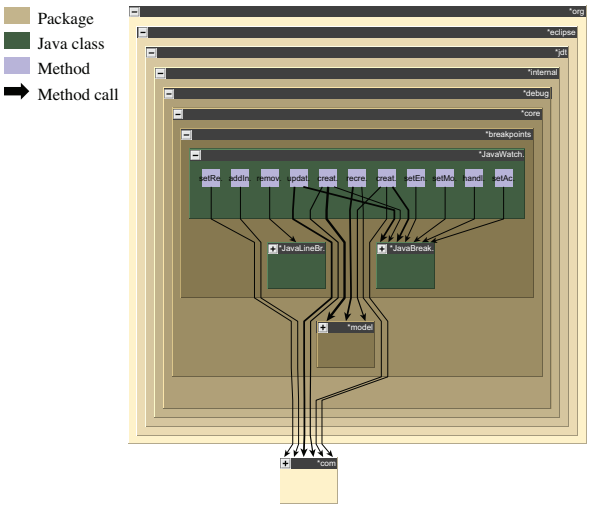
- `Creole`: Proceeding from the graph presented by Figure 6b, we zoomed into the package `breakpoints` and expanded its outgoing edges. We manually laid out the class nodes to identify the outgoing method calls of class `JavaWatchpoint`. Next, we expanded the node representing class `JavaWatchpoint` to view its members. We used the filter to hide field and constant nodes from the graph. After that, we expanded the corresponding outgoing edges of the node `JavaWatchpoint`. To identify the methods involved in outgoing calls we had to manually rearrange the method nodes. Figure 7b depicts the final graph. Although, `Creole` provides a zoom functionality, it was a tedious job to identify involved methods in this complex graph.

- `Imagix-4D`: We continued the analysis from the existing graph (see Figure 6c) and filtered out the nodes of classes with calls to `JavaWatchpoint`. Next, we configured the view to show classes, methods, and contains relationships. We selected the node representing class `JavaWatchpoint` and used 'Step down' to add the methods contained in `JavaWatchpoint`. After that we configured the view to display method calls which resulted in the graph depicted in Figure 7c. The identification of involved methods was time-consuming—we had to check each class in the graph and manually find out which methods use it.

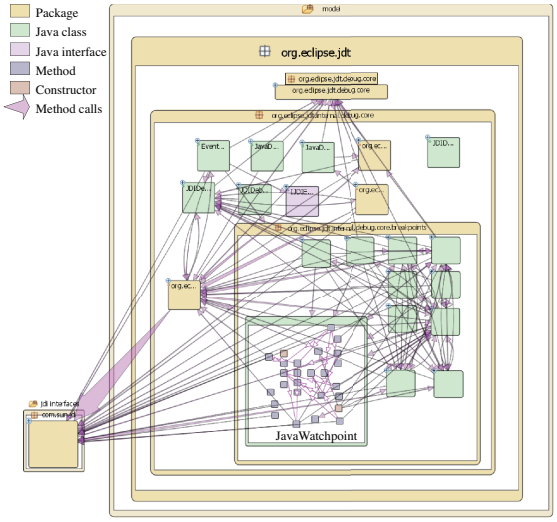
The creation and understanding of the graphs with `DA4Java` was less effort than with `Creole` and `Imagix-4D`. The main problem of `Creole` is the missing filter functionality which led to cluttered graphs that needed more effort to create and understand. Creating the graph with `Imagix-4D` was straight forward, however, identifying the involved methods was difficult. `Imagix-4D` lacks a function to filter internal method calls. Extending the highlighting feature to select dependent nodes via aggregated edges would have lowered the analysis effort significantly.

Graph size and aesthetic The following table represents the measures for each graph.

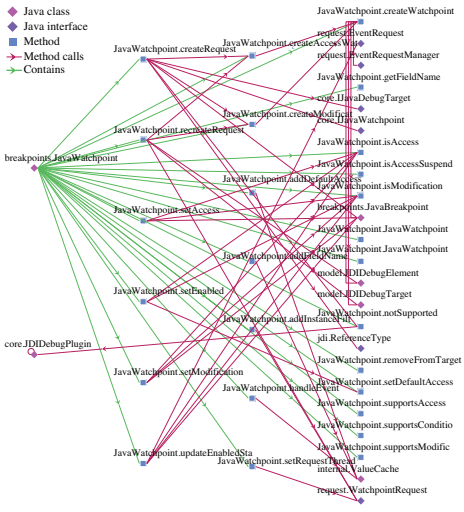
	#nodes	#edges	#edge crossings
DA4Java	23	17	10
Creole	51	> 100	> 250
Imagix-4D	38	≈ 64	≈ 230



(a) `DA4Java`



(b) `Creole`



(c) `Imagix-4D`

Figure 7. Task 2: Methods of `JavaWatchpoint` involved in outgoing method calls.

Also in this task DA4Java showed better results than Creole and Imagix-4D. The graph created with DA4Java contains significantly less nodes, edges, and edge crossings. According to the graph and the measures, the effect of Creole's missing filter functionality becomes even more apparent. Compared to 17 edges in the DA4Java graph, the Creole graph contains more than 100 edges leading to more than 250 edge crossings. Even in full screen mode the graph shows a network of overlapping edges that is difficult to grasp. For such graphs, filtering is a must.

Comparing the DA4Java graph with the Imagix-4D graph, we can clearly see the advantage of nested graphs which group nodes and edges according to the containment hierarchy. 26 additional 'Contains' edges are needed by Imagix-4D to visualize the containment hierarchy. Furthermore, Imagix-4D lacks the filters to remove edges between nodes such as class internal method calls. DA4Java provides such a filter which is called 'Remove node internal dependencies.' The result is a graph that contains significantly fewer edges, is easier to layout and understand.

4.3. Summary of results

In both tasks, DA4Java showed better results than Creole and Imagix-4D with respect to the effort to create the graph and perform the analysis. Examples pointed out the main advantage of DA4Java which is its set of add and filter features. They allowed us to create graphs that contained a significantly smaller number of nodes, edges, and edge crossings. For instance, in Task 1 the graph of Creole contained more than 50 edges although 5 edges were enough to provide the answer. Furthermore, the graphs created in Task 2 confirmed that the more details are added the more complex graphs become. With DA4Java the user is able to handle the increasing amount of information and create graphs that are task oriented. Answers were found within short time. In contrast, solving the task with Creole or Imagix-4D was difficult and needed significantly more effort from the user.

5. Related Work

Most of the existing program comprehension tools are geared to a top-down approach, and lack with regard to bottom-up exploration. Von Mayrhauser *et al.* found that program understanding is not unidirectional, *i.e.*, top-down or bottom-up exclusively [17]. They present a meta model that integrates both, the top-down model of Soloway *et al.* [12] and the bottom-up model of Pennington [8]. This calls for a better integration of the two approaches. DA4Java, in comparison with other tools, provides here 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 demonstrated 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 which is 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 [6]. It follows mainly a top-down analysis approach and uses Simple Hierarchical Multiperspective views (SHriMPs) [15]. 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. The main difference to DA4Java is its lack of features for the incremental composition of graphs.

Shrimp [14] is a further development of Rigi. It supports Simple Hierarchical Multiperspective views. 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 and its limitations were already discussed in this paper.

IBM's Structural Analysis for Java™ (SA4J) is a tool to analyze structural dependencies of Java applications and detect "anti-patterns."[4] 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.

CodeCrawler [3] uses 'Polymetric Views' to display various aspects of object-oriented software systems. Its focus is more on 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.

Softwarenaut [4] is a tool used for top-down exploration of large software systems and therefore has some of the already discussed limitations. Its combination of a detail view and overview view is interesting. 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 [5] is a tool that uses a 3D representation to visualize source code. It is a further de-

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

velopment of the SeeSoft [2] 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.

6. 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. Blurred graphs are therefore reduced by effective filter algorithms to enable a user a quick comprehension path in large software systems.

With our approach a user remains focused on the program comprehension task while increasing the understanding in a stepwise manner. We evaluated our tool in a case study with the JDT Debug plugin of Eclipse and compared it with two cutting edge tools Creole and Imagix-4D. Two typical program comprehension tasks were performed and we evaluated the user effort to create the graphs for the comprehension tasks as well as the size and complexity of these graphs. In both tasks, our approach outperformed Creole and Imagix-4D. Graphs created with DA4Java contained significantly fewer nodes and edges and needed less effort to understand.

As future work we foresee a controlled user experiment to do a detailed analysis of the features of our tool on more general program comprehension tasks such as described in [7]. That will give us additional input for fine-tuning our graph algorithms and the user interface.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] H. C. Purchase, D. Carrington, and J.-A. Allder. Empirical evaluation of aesthetics-based graph layout. *Empirical Software Engingeering*, 7(3):233–255, 2002.

[10] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[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] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the International Conference on Software Maintenance*, pages 275–284, Opio, France, October 1995. IEEE Computer Society Press.

[16] 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.

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