Representing Software Features in the Eclipse IDE

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Balsiger

November 2010

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz

Dr. David Röthlisberger

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Markus Balsiger
markus.balsiger@students.unibe.ch
http://scg.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

The Eclipse IDE only provides static views of the source code, thus missing runtime information of a software system. In a polymorphic object-oriented language like Java, however, it is difficult to understand the runtime behavior and therefore maintain the features of an application based on the static source code alone.
We present BOA, an Eclipse plug-in representing features and supporting developers in bug fixing. BOA is able to record features in terms of dynamic information stored in an invocation tree, to graphically represent recorded features and to present metrics calculated based on the dynamic information of a feature. The focus of BOA is the simple and user friendly use by creating a single compact view, XML storage, and an open and a flexible recording mechanism based on aspects. We describe the BOA plug-in and all the techniques applied to implement feature recording, transmission and analysis.
Finally, we analyze BOA's performance by experimenting with different recording techniques and evaluate the use of the plug-in by means of a case study. For this evaluation we create a set of scenarios to measure the performance impact of different recording techniques and document the use of BOA in different real-world software systems.

Acknowledgements

I especially thank David Röthlisberger, who shared his knowledge and ideas with me. His devotion to computer science and enthusiasm motivated me during this project. Without his encouragement and the hints and advice he gave me, I would not have been able to complete this thesis.
Also I want to thank Oscar Nierstrasz for giving me the chance to work on my bachelor thesis at the Software Composition Group.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Identification

1.1.1 The Addressed Problems

We address two general problems in software development in this project: The gap between the static source code and dynamic runtime behavior of a software system and the fact that modern IDEs do not provide a feature representation. These two problems exist in all programming languages and are especially relevant for polymorphic object-oriented languages such as Java, C++ or Smalltalk. Code reading and maintenance of complex software systems is difficult in such polymorphic languages, because of the gap between the static code and the dynamic behavior at runtime. We introduce three problems in more detail:

Dynamic behavior of a software system. The first problem derives from the fact that the dynamic behavior of an application is not always predictable from the static source code alone [5, 19]. In an object-oriented programming language like Java, C++, C# or Smalltalk, where classes inherit from others, the developer is not always able to determine the actual implementation of a method that was invoked at runtime simply by reading the static source code. A developer often does not precisely know what happens at the class- and method-level inside a running application when a feature is executed. We understand the term feature as a single well defined part of the solution to a system's problem domain.

Best practices make source code reading even more difficult. A feature is often scattered in many small pieces of code. Small pieces of code have their origin in many best practices encouraging developers to write small methods [2].

Gap between a feature and methods, classes and packages. It is a very natural way for a customer to talk about software by talking in terms of features, as first of all what the customer will pay for or where problems may arise are specified features [15], and second in most cases he does not have the knowledge to talk about source code or functions. What he thinks of is the behavior of a system, triggered by a user [7]. When software engineers talk about features, they talk about source fragments like methods, packages or classes. There is a big gap in between a feature and a piece of code, as these are not related to each other elsewhere than in the heads of the developers and sometimes in the software system documentation or as comments in the source code itself. As a software system grows, it is impossible to know every part of it by heart. A developer often is not the original author of the piece of code he has to maintain. Reading and understanding code is 50% - 60% of the work time job of every engineer [1, 3], but reading and understanding an unknown software system can be a difficult job. A developer might be confronted with unknown code because of the following facts:

- the developer is not the original author
- the code was written a long time ago
- lack of documentation

1.1.2 Requirements

To solve the two general problems in software development, a feature in terms of method invocations at runtime must be recorded and visualized inside the IDE. The requirements of a solution that solves our three detailed problems are:

- be embeddable into an IDE, providing feature information inside standard views like the source code editor and therefore help the developer in reading and understanding code
- be able to record, update and delete features in terms of method invocations without forcing the developer to change his habits or slow him down in the process of software development
- connect a feature with its corresponding methods

1.2 BOA - Feature Recording and Visualization

BOA is an Eclipse plug-in which enriches the standard views of the IDE with feature information in terms of method invocations and delivers additional information like metric data about the number of called methods or the call stack depth of a certain method within new views. BOA supports the developer in reading and understanding code by adding feature-relevant information like the relevance of a method for a feature or source code markers, which are icons displaying hints inside the editor or the package explorer. The main view of the plug-in is called *FeatureView*. The *FeatureView* is designed in a way that it can be easily embedded into most perspectives, which are settings describing the positioning of views in Eclipse, for example the

Java perspective, the project browser perspective or the debug perspective.

The process of recording a feature using the plug-in is similar to launching the application with the standard launch mechanisms of Eclipse. The recording slows down the developer in his work as little as possible, which means the applications starts and responds fast even with recording activated.

Besides the feature representation and information views, BOA offers the *BugMapper*, which assists the developer in finding methods that are relevant for a certain bug. The developer selects the features in which the bug arises and in which not, where the latter can be a subset of all features that are verified to be bug-free. The *BugMapper* then analyzes the selected features and compares the bug-affected with the bug-free ones. The result is a list of methods ordered by the relevance for the analyzed bug. From within this view, double-clicking brings the developer to the source code of the method, which connects the probably faulty method to the source code.

To make the plug-in usable for as many projects as possible, BOA is flexible in terms of the project setup like the folder structure, launching parameters, Java runtime or used technologies like AspectJ, which is supported. At least every plain Java project that Eclipse is able to launch can be analyzed by BOA. Also the plug-in represents and handles the running application in a similar way Eclipse does out of the box. For example, Eclipse's built-in console is connected to the standard output stream of the application, status information is available, and the application from which features are recorded can be launched and stopped from inside the IDE, just like the developer is used to. As soon as the run configuration was made once, it is stored in the IDE. To make the first-time experience as satisfying as possible, BOA supports the developer with heuristics that aim at avoiding recording classes that are not possible to record because of the maximum class file size of the Java Virtual Machine. The plug-in allows experienced developers to exclude classes by manually adding them to an exclude list. BOA avoids side-effects by running the application in a new Virtual Machine instance and communicating with it over sockets.

To ensure the suitability of BOA inside the incremental software development methods of agile software development, recorded features can be saved in an XML file and stored inside the project. The feature information is not only available right away from the start when creating or reopening a project, but can also be stored and versionized with the source code. As a result, feature information cannot only be recorded, changed, deleted and therefore analyzed during the whole development process, but also restored to a specific version of the software system.

Other than recording and presenting features, BOA calculates metric information like call stack depth or number of invocations of a certain method. This metric information helps developers when it comes to locating and eliminating performance bottlenecks.

1.3 Contributions

The contributions of this work are as follows:

Data gathering technique. Data gathering is the process of recording a feature in terms of method invocations. BOA comes with its own data gathering technique using aspects.

Eclipse plug-in BOA BOA including the *BugMapper* which helps developers finding faulty methods, all the view enhancements like the *FeatureView* and the BOA client which is woven into the applications classes for data gathering. A built-in storing mechanism enables the developer to store the features of any project during the development process.

Transmission technique. The transmission technique is the mechanism used to transmit the invocation information from the BOA client to the Eclipse plug-in, containing the model and the indexing solution which lowers the size of the transmitted data by sending already sent invocation information only as integer values pointing to the actual information.

User guide and software system documentation. Besides the implementation, we provide a small user's guide with installation instructions and a developer's guide.

1.4 Structure of the Thesis

After this short introduction, problem statement and solution description we discuss related work in the domain of feature representation, method invocation recording and metric analysis which we took into account when creating BOA. Furthermore we talk about implementation details and make an evaluation of BOA.

State of the Art. We take a look at work done in the topics of feature recording and feature representation in IDEs and the metrics used in this project. Also we discuss why currently available solutions in the domain of feature representation and recording do not fit the requirements we described in Section 1.1.2 - Requirements.

BOA. In this Chapter we present our solution step by step regarding the requirements we formulated. We introduce the plug-in by screenshots and functional descriptions, as well as the basic architecture of the recording system.

Implementation. In the fourth Chapter we describe the implementation of BOA and its mechanisms. We present the final product and describe why the final implementation of BOA is based on the decisions we made. We take a look at source code fragments and create a basic knowledge about the structure of BOA.

Evaluation. In the evaluation Chapter, we discuss the tests we made with BOA and give a summary of our lessons learnt when using BOA in multiple projects and compare different implementations of the recording technique using some basic benchmarks in various scenarios.

Summary and conclusions. In the Chapter summary and conclusions we discuss further perspectives of BOA.

Appendix. In the Appendix we offer a user's and a developer's guide.

Chapter 2

State of the Art

In this Chapter we study related work concerning dynamic information recording and feature representation in the IDE.
There are four feature related topics we would like to take a close look at and which influenced this project the most: Major, FeatureEnv, Seesoft and Senseo. Furthermore we would like to discuss other related research work and introduce a few metrics which BOA provides.

2.1 Dynamic Information Recording with Major

Major[1] is an advanced tool to weave aspects into applications and even into the Java runtime libraries. It is part of the Ferrari project. Major's name is recursively defined *Major Is AspectJ With Overall Rewriting*. As one might guess from the name, the underlying framework is AspectJ. The advantage of Major derives from a custom UDI. Major also includes Carajillo[2], a tool that allows Major to do efficient calling context reification and access to the calling context directly within aspects.

The distribution of Major features a set of example aspects which allow the developer to obtain precise invocation information. For instance, the examples contain the cct aspect, which creates the calling-context tree or the mem-leak aspect, which aims at detecting memory leak candidates. Major was used in Senseo and in this project in version 0.5.

Major is not enough to fit our formulated requirements. Major is a technique to record dynamic information of a running application, and is therefore taken into consideration when it comes to dynamic information recording in this project. But it is not a plug-in for the Eclipse IDE enabling the developer to record and visualize features.

[1] http://www.inf.usi.ch/projects/ferrari/MAJOR.html
[2] http://www.inf.usi.ch/projects/ferrari/CARAJillo.html

2.2 Feature Driven Browsing - Integration of Features in IDEs

David Röthlisberger et al. [17, 18] validated the usefulness of a feature-centric perspective on software maintenance. They implemented a prototype of a feature browser for the Squeak IDE in Smalltalk called *FeatureEnv*. Besides the structural and textual representation of the software system, the feature browser presented visual representations and metrics of features. The *FeatureDedicationMatrix* of BOA, which will be introduced later, was inspired by the *compact feature overview* of *FeatureEnv*. Figure 2.1 presents an overview of *FeatureEnv*.

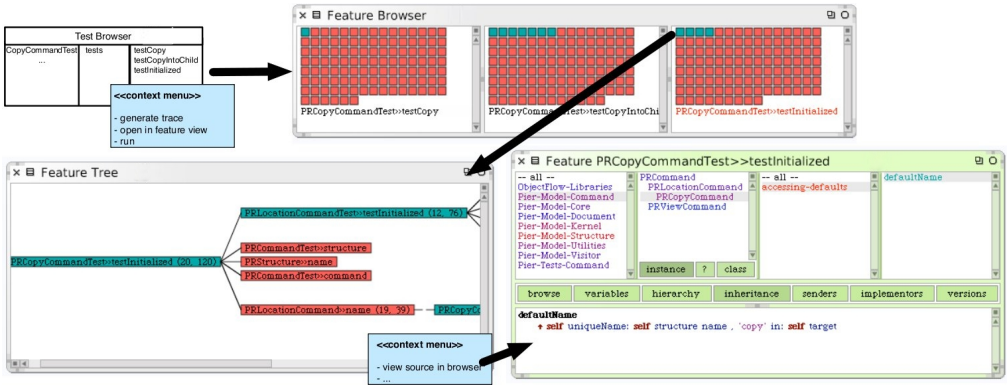

Figure 2.1: *FeatureEnv*'s views

The feature browser called *FeatureEnv* presented in the *Feature Driven Browsing* paper fulfills many of our requirements, for example the graphical feature information available inside the IDE. It supports a developer in reading and understanding code. But it does not support him in automatically locating faulty methods from a feature's dynamic information.

2.3 Seesoft-A Tool For Visualizing Line Oriented Software Statistics

Stephen G. Eick et al. [6] implemented a tool called *Seesoft* which allows developers to analyze code visually by mapping each line of code to a thin colored row and each source code file to a column. The coloring of such a row is based on statistical information which *Seesoft* calculates using the information from version control systems and profilers. The basic data is information about a line of code like the author, the number of developers changing a line, the last change or the total number of changes of a line.
The visualization is interactive, allowing the developer to view information about a line of code by hovering over a line with the mouse cursor or highlighting only lines of code touched by a selected modification request. We present an example screenshot of *Seesoft* in Figure 2.2. Due to *Seesoft*'s visualization allowing one to see desired information in an overview, *Seesoft* can be

used in application areas like code discovery, developer training, project management or system testing.

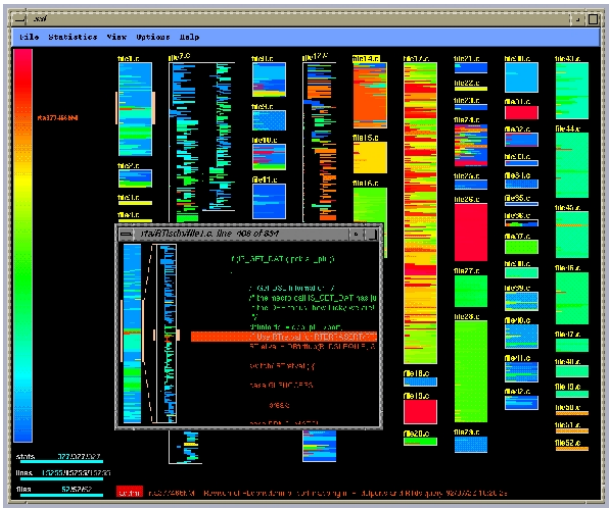

Figure 2.2: A screenshot of *Seesoft*'s visualization

The field experiences made with *Seesoft* show interesting facts about source code, for example that files changed by many developers have more bugs than files only maintained by one or two developers.
We were inspired by the visualization using colors and the statistical information used by *Seesoft*. Despite the visualization, *Seesoft* does not solve the problems we described. Seesoft is not embedded into an IDE and represents line based statistics, but not features.

2.4 Runtime Information in Eclipse - Senseo

Senseo [10] is an Eclipse plug-in created by Marcel Härry in his Master thesis. It enriches the Eclipse IDE's standard views with several metrics about the dynamic behavior of a software system. Also additional views for example a ring-chart and a collaboration view are provided. *Senseo* uses Major to record method invocations.

Senseo comes close to what we need to solve the described problem available at the time we started engineering. It is able to record method invocations in all Java libraries and even in the Java runtime itself by instrumentalizing the JRE. Senseo can recapture method invocations, and therefore build a whole stack of information about all possible invocation trees inside a software system. Also it provides a heatmap-like annotation inside the source code editor, enabling the developer to read invocation information along with the code. It captures many interesting

metrics, for example how many objects were instantiated from a single method invocation. Figure 2.3 presents the views of the Eclipse IDE enriched by *Senseo*.

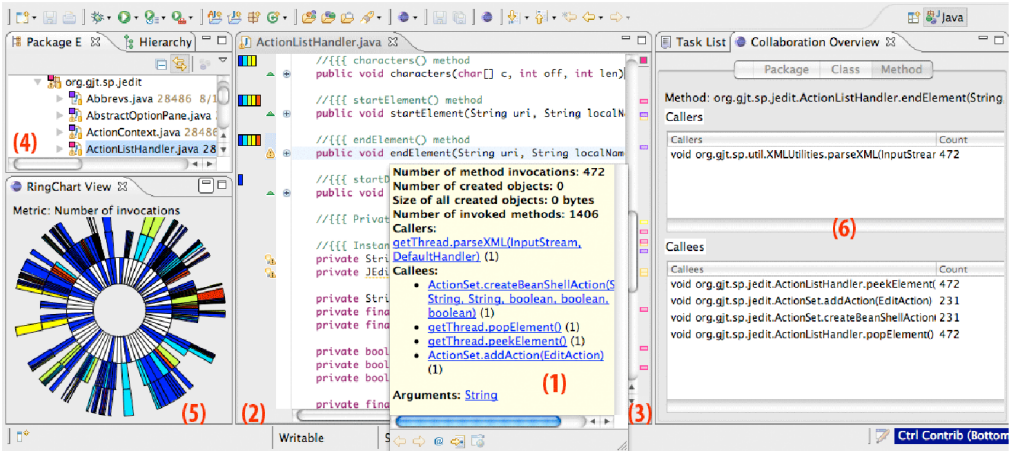

Figure 2.3: Eclipse with installed *Senseo* plug-in. (1) Tooltip information, (2) frequency of invocation, (3) average number of invocations, (4) augmented package explorer with dynamic information, (5) calling context ring chart, (6) dynamic collaborations view

The reason why we do not consider *Senseo* to be a solution for our specified problem is the fact that it does not capture method invocations in the context of a particular feature, but uses the dynamic information to analyze the application. *Senseo* helps the developer to understand code and offers metrics and views to locate bottlenecks, but it is not a feature-centric extension for Eclipse. The fact that *Senseo* is able to capture even invocations in the Java system libraries is one of the big advantages delivered by Major. But as BOA aims at helping software engineers to read a software system's code and find faulty methods of features, the capturing of the system libraries is not an important point, as a developer normally does not care about the actual runtime based implementation of a standard feature of the JRE when it comes to bug-fixing or code reading.

2.5 Other Related Research

Supporting developers in understanding object-oriented software systems by visualizing the runtime information is provided by [4, 9, 12, 14]. Other tools that use dynamic information of an application are for example the Program Explorer written by Danny B. Lange and Yuichi Nakamura [14] or GraphTrace by Michael F. Kleyn and Paul C. Gingrich [13]. In the domain of Java programming, Reiss [16] developed Jive[3], a tool to visualize the runtime activity of Java applications in real time.

[3] urlhttp://www.cse.buffalo.edu/jive/

Our approach is different from the ones above because we deliver a plug-in for Eclipse that aims at enabling the representation of features in terms of method invocation trees and supporting a developer in detecting faulty methods from recorded features throughout the entire development process.

2.6 Metric Information

Metrics are an important means in software engineering, as they enable software engineers to compare software systems. They can help developers to detect flaws in the implementation of a software system. For example showing the developer how many objects were created from a single call of a certain method tells him where an optimization might be necessary. We discuss what kind of metrics BOA provides in the following Sections.

2.6.1 Feature Dedication

Feature dedication is a metric expressing how specific a method is for a certain feature. For example a method is very specific for a certain feature if only that feature uses this method. Methods that are used by almost every feature are likely to be nondedicated for all features. The calculation of feature dedication is simple.

$$\frac{NumberOfFeatures - NumberOfFeaturesUsingMethod}{NumberOfFeatures - 1}$$

This metric returns a value between 0 and 1, where 1 means the method is very specific to a certain feature, and 0 means it is not dedicated at all. BOA uses the feature dedication in the *GraphicalTree*, which is a graphical representation of the invocation tree, to enable the developer to find important methods quicker. Also feature dedication is shown in the *FeatureDedicationMatrix*, a view displaying a colored square per method, allowing the developer to get an overview of the most important methods of a feature.

2.6.2 Bug Relevance

The Bug Relevance is a metric describing how relevant a method is for a certain bug. The information used to calculate Bug Relevance are the features in which the bug arises and in which it does not. What we actually describe with this metric is the fact that the chance of a method being wrong is smaller the more features use it without yielding undesired results. For example, analyzing only two features with a bug, and a method that is used only by these two features, the chance that this method is defective is bigger than for methods used by many correctly running features.
In a way, this metric describes the same as feature dedication. If the developer selects a single feature as bug-affected, and all other features in the software system as bug free, then what he

receives as a result by Bug Relevance is the dedication of the methods in the selected feature. The two differences are the following:

Multiple features. Bug Relevance takes multiple features into account to calculate the relevance, while the feature dedication only compares the occurrence of method invocations inside the invocation tree of a single feature with all recorded features of the software system.

Selective rest of features. Bug Relevance does not check the methods of multiple features against all other features, but just those we know to be free of a certain bug. Feature dedication in contrast compares to all other recorded features in the software system.

The formula which calculates the bug relevance is:

$$\frac{NumberOfMethodOccurrencesInFaultyFeatures}{NumberOfFeatures - NumberOfMethodOccurrencesInBugfreeFeatures}$$

It is important that the developer only selects the bug free features that are of the same kind. The same kind here means features that are similar to the bug-affected features. Otherwise the bug relevance suffers from noise. In most cases, the developer should keep the focus inside a certain group of features. For example all export functionalities or all features dealing with statistical calculations. All these features have a big chance of using a method in the same way, and therefore being vulnerable to the same bug. Truly, this is just a best practice, because it might also be the right choice to take all recorded features in the whole system into account. This aspect of bug-free feature selection is another reason why we created a tree-like category structure to store the features. The tree structure makes it easier to select a whole group of similar features.

2.6.3 Miscellaneous Metrics

We now introduce miscellaneous metrics for invocation trees. We explain every metric with an example based on Figure 2.4, which represents a simple invocation tree.

Maximum call stack depth of a method. The maximum call stack depth of a method is a metric which represents the maximum depth of the call stack from a method in the method invocation tree of a feature. Identical methods can have different call stack depths in different places inside the method invocation tree. For instance, a method calling another method, which returns a value without executing other methods, has a call stack depth of one.
In our example invocation tree, the *Maximum call stack depth* of method **C** is three (C here stands for both C* and C**).

Invocation depth. The invocation depth describes how deep in the invocation tree the selected method is. Identical methods can have different invocation depths at different places in the method invocation tree. For example, the first triggered method of a Java application's feature

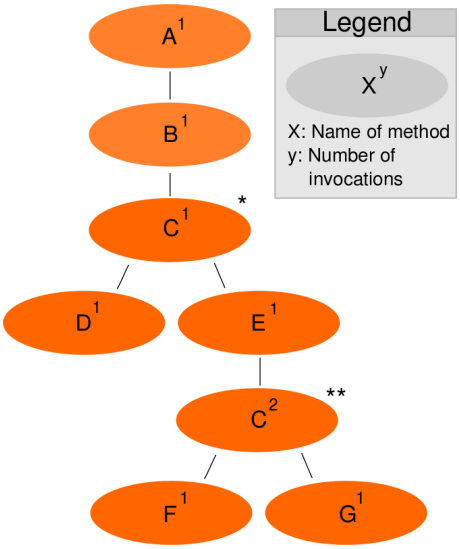

Figure 2.4: Example invocation tree

has the invocation depth 0.
In the example invocation tree, the invocation depth of method **C*** is two while method **A** has depth of zero.

Number of invocations. This metric shows how often a method has been executed at a specific position of the method invocation tree. Identical methods can have different numbers of invocations at different places of the method invocation tree. A method which was executed ten times in a row, for example in a loop, has a number of invocations of ten.
In the sample tree, the number of invocations of method **E** is one and the one of method **C**** is two.

Number of invocations in an entire feature. Unlike the number of invocations, this metric not only represents repetitions of a method execution in a row, but counts executions of a method inside a features method invocation tree.
In our example tree, the number of invocations of **C** is three and the one of **F** is two (because C** is called twice).

Number of method calls from a method. The number of method calls by a method describes the size of the invocation subtree of a method in terms of nodes of the method's subtree. For example, a method that calls two methods, which both again call two methods, has a number of methods called of seven.
In the example invocation tree, method **E** calls seven methods.

Number of method calls per feature A feature's number of method calls is equal to the number of methods called by the root method of the feature's method invocation tree.
The metric value of the feature represented by the sample tree is eleven.

Number of different callers. This metric describes how many different callers a method has. If a method A is called twice from method B and once from method C, then the number of different callers is two.
In the example invocation tree, method **G** has only one caller, while method **C** has the two different callers **B** and **E**.

Chapter 3

BOA

BOA is a plug-in for Eclipse. It can be installed and updated over the update-site mechanism. Eclipse's built in OSGi implementation automatically downloads other dependencies, for example the AspectJ plug-in.

After the installation, a new view called *FeatureView* is available. This is the main view of BOA. With this view, the developer records, updates, deletes and visualizes features. The *FeatureView* actually contains four sub-views:

- The *FeatureTree* containing all recorded features in a tree-like category structure
- The *SimpleTree* represents the method invocations of a feature and allows manipulation like removing invocation information recorded from a specific thread
- The *GraphicalTree* which represents the method invocations graphically and additional information like the signature or the feature dedication
- The *FeatureDedicationMatrix* displaying all methods of a feature and the corresponding feature dedication

Figure 3.1 shows the feature view after the successful installation of BOA.

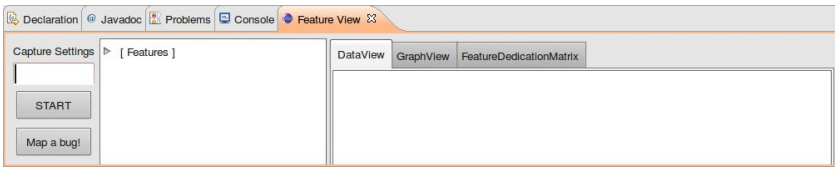

Figure 3.1: The *FeatureView*

The *FeatureTree* is used to select already recorded features. Features are organized in a tree-like

category structure because of two facts: First, the chance of complicating the feature search because of the list's length or different naming-patterns of the features is huge. With a tree structure, a developer will find a feature quicker. Second, when using the *BugMapper*, it is often a good choice to only select nearby bug-free features when analyzing bugs, where nearby means features of the same functional type. For example, a functional type can be all export features or all features dealing with a web service.

The *SimpleTree* is a fast and easy-to-read visualization of a selected feature's invocation tree, as it only represents the invocation tree without any further graphical information. In addition, it provides the developer with the ability to remove invocations from an accidentally recorded thread using the delete key. We will cover this in more detail later in Chapter four.

The *GraphicalTree* is the heart of the *FeatureView*. It renders the tree showing more information like the signature of a method while hovering over an item of the invocation tree and taking the user to a source code fragment or showing metric information like the number of invocations over the context menu. When first looking at the tree, most parts of it are collapsed to ensure quick loading and to not confuse the viewer with too much information. Each method invocation item is colored accordingly to its feature dedication, using a color heatmap. On Figure 3.2, a simple application is represented by the *GraphicalTree* view.

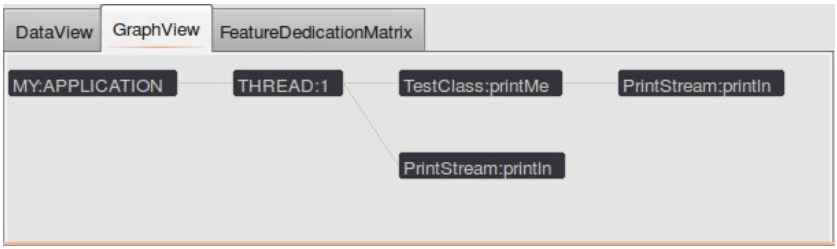

Figure 3.2: The *GraphicalTree* view

The *FeatureDedicationMatrix* displays a single square for every method in the selected feature. The squares representing the method invocations are heatmap colored like in the *GraphicalTree*. The developer benefits from a visualization of the most important methods of a selected feature. Figure 3.3 shows the *FeatureDedicationMatrix* of a feature which has only a single method in common with all other features in the system.

To run an application with feature recording activated, the developer has to launch it with BOA's own run configuration called *DataGathering*. Once configured, the run configuration is stored in Eclipse. The most important tabs of the *DataGathering* run configuration are the *LTW Aspectpath* tab, which provides load time weaving configuration, and the *Weaver Excludes* tab. We present these two views on Figure 3.4 and 3.5. The settings inside these tabs must be maintained by the developer. All other tabs are automatically set up by the plug-in or Eclipse

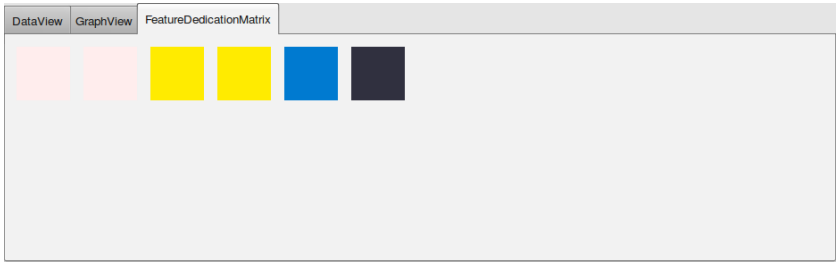

Figure 3.3: The *FeatureDedicationMatrix* with methods ordered by their feature dedication

and normally do not need any attention for the first run of an application. BOA uses aspects to weave classes and collect invocation information. To add support for related projects and even load time weaving AspectJ projects, the *LTW Configuration* tab is available. Inside the Exclude tab, classes, methods and packages which fail to weave or which should be excluded in every run can be excluded from recording using complete names or wild-cards. Also a heuristic check that analyzes classes and excludes them if their weaved size possibly grows bigger than the maximum allowed class file size of the Virtual Machine is available.

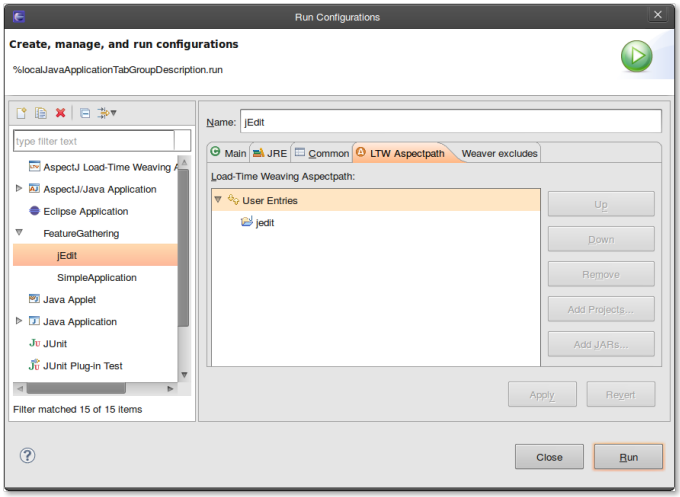

Figure 3.4: AspectJ's *LTW Aspectpath* configuration tab view

When the application is launched, feature recording can be started with the *Start* button in the *FeatureView*. A click not only starts the recording of a feature on plug-in side, but also activates BOA's client, which will then start recording. As long as the user does not record any feature information, the application runs faster than during recording. When hitting the *Start* button, the plug-in sends a command over a socket to the application. Receiving this command, the application connects to BOA over another socket and sends the invocation data in an optimized

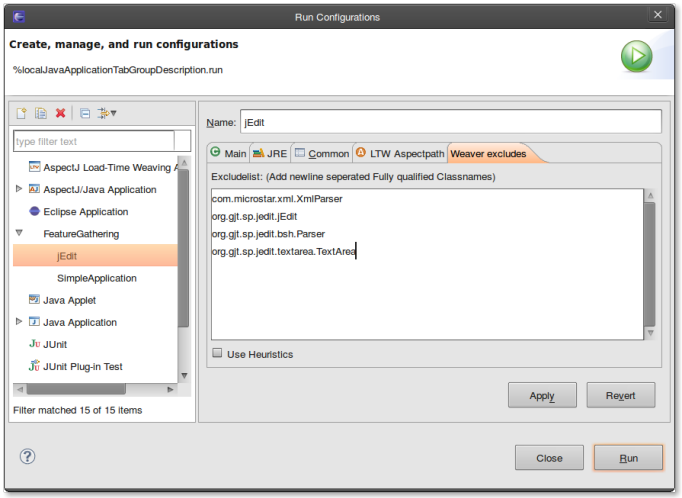

Figure 3.5: The *Weaver Excludes* tab with disabled heuristics check

manner to the plug-in, which builds up the invocation tree. When hitting *Stop* again, all socket connections from the application to BOA are closed by sending the corresponding command. A dialog asking the developer about the name and category for the recorded feature opens up. Categories are organized in a tree-like structure. Every category can hold sub-categories or features. From the beginning, the *Features* category with a single sub-category called *Uncategorized* is available. These two items in the category tree cannot be deleted. When a category is deleted, all the features inside the category or its sub-categories are moved to the *Uncategorized* category. After filling out the information, the updated feature information of the project is automatically saved and the new feature is available in the *FeatureView*.

The storage mechanism is storing the data inside an XML file called ”boa.xml”. The tree structure of the features and categories is written 1:1 as a simple DOM. This allows advanced users to import the structure into other tools which provide features that BOA does not, for example other metrics or another visualization. Furthermore it is easier to merge different versions of files containing stored features when they are not binary and human readable. When moving from a project to another using Eclipse’s ResourceView or the Navigator, the corresponding feature information is loaded automatically from the projects file or from a cache. When working with related projects, the *FeatureView* always shows the information that is currently of interest. However, when switching to a Java source file from another project via the editor, BOA does not load the project’s feature information, as reading other classes does not always mean developing in the corresponding project.

The *BugMapper* button brings up a selection window asking the developer about bug appearance information. The developer selects both features that are affected by a bug and those that are not. The part where features that are non-affected are selected is important because of two facts: first of all, it is possible that a feature was not yet tested for a certain bug. And second,

if all non-affected features inside the software system are selected as bug free features, noise of other features might lead to a fuzzy result. Selecting solely features that use certain objects and methods in a similar way leads to a reliable result. Hitting the *Map* button returns a list of methods ordered by their bug relevance. Double-clicking opens the editor with the corresponding source fragment. A screenshot of the *BugMapper* with an exemplary selection of bug affected and non-affected features is presented on Figure 3.6.

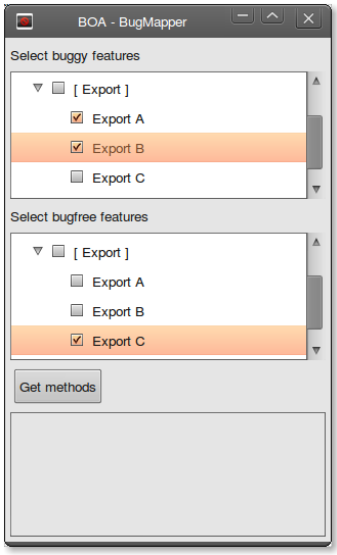

Figure 3.6: The *BugMapper*

Chapter 4

Implementation

We now discuss the following implementation details of BOA

- The fundamental decisions like the IDE, covering plug-in and environment
- The dynamic information gathering technique and how it evolved during this project
- The implementation of the data transmission between BOA and its client
- The storage mechanism of BOA and the plug-in design specific implementation details

4.1 Fundamentals

4.1.1 Java

BOA is a plug-in supporting Java software engineers. Java is widely used in new software engineering projects. Creating a plug-in for Java makes sense as it allows us to provide a useful addition for a big audience, also because of the fact that the development kit and Java applications in general run on multiple platforms. Our initial intent was to give Java developers a tool to solve their bugs or performance problems faster and to enable them to read and understand source code faster.
When this thesis was written, Java was available in version 1.6 and under the main development of Sun (Oracle). In this project, Sun's reference implementation and the open source version called OpenJdk were both applied as development and testing runtime to ensure that the solution works with at least these two often used Virtual Machines. Both were available in version 1.6.

4.1.2 Eclipse IDE

As introduced before, one of the main requirements of this work is to offer the *FeatureView* in a handy way. As many Java developers work in Eclipse or proprietary derivates like BEA Workspace or MyEclipse, creating a plug-in for the Eclipse IDE again makes sense to provide a solution for a big audience. BOA was integrated into Eclipse Classic in version 3.6 and tested under Linux.

4.2 Dynamic Information Gathering and Tree Creation

Dynamic information gathering defines the task of recording a feature in terms of method invocations and building a method invocation tree. How this recording process perturbs the running application and what information exactly is transformed into the invocation tree are the two most important specifications. A high amount of transmitted information can slow down the running application while skipping the transmission of important information decreases the available information for the developer, and thus the possible fields of application of the plug-in. The final implementation of BOA uses AspectJ to gather dynamic information. Because there are very few Java applications not applying multi-threading, method invocations are captured on multiple threads. Also besides the actually called method, the first declaration, which in Java can be for example a method of an abstract class, is recorded.

4.2.1 The Invocation Data Model

BOA has two internal representations of invocations. First we present the invocation data which is transmitted from the BOA client inside the running application to the plug-in. This model is transfered into a tree model when it comes to representation of invocation trees. The root node of such a tree model is stored together with a name in a feature. Such a feature is attached to a single category. Categories are used to organize and differentiate the features and are organized in a tree structure.

Figure 4.1 is the UML class diagram of the two different invocation information representations.

The requirements for the transmission model are simple and small by design. What BOA needs to know are the following seven invocation details:

- Package
- Class
- Method
- First declarator

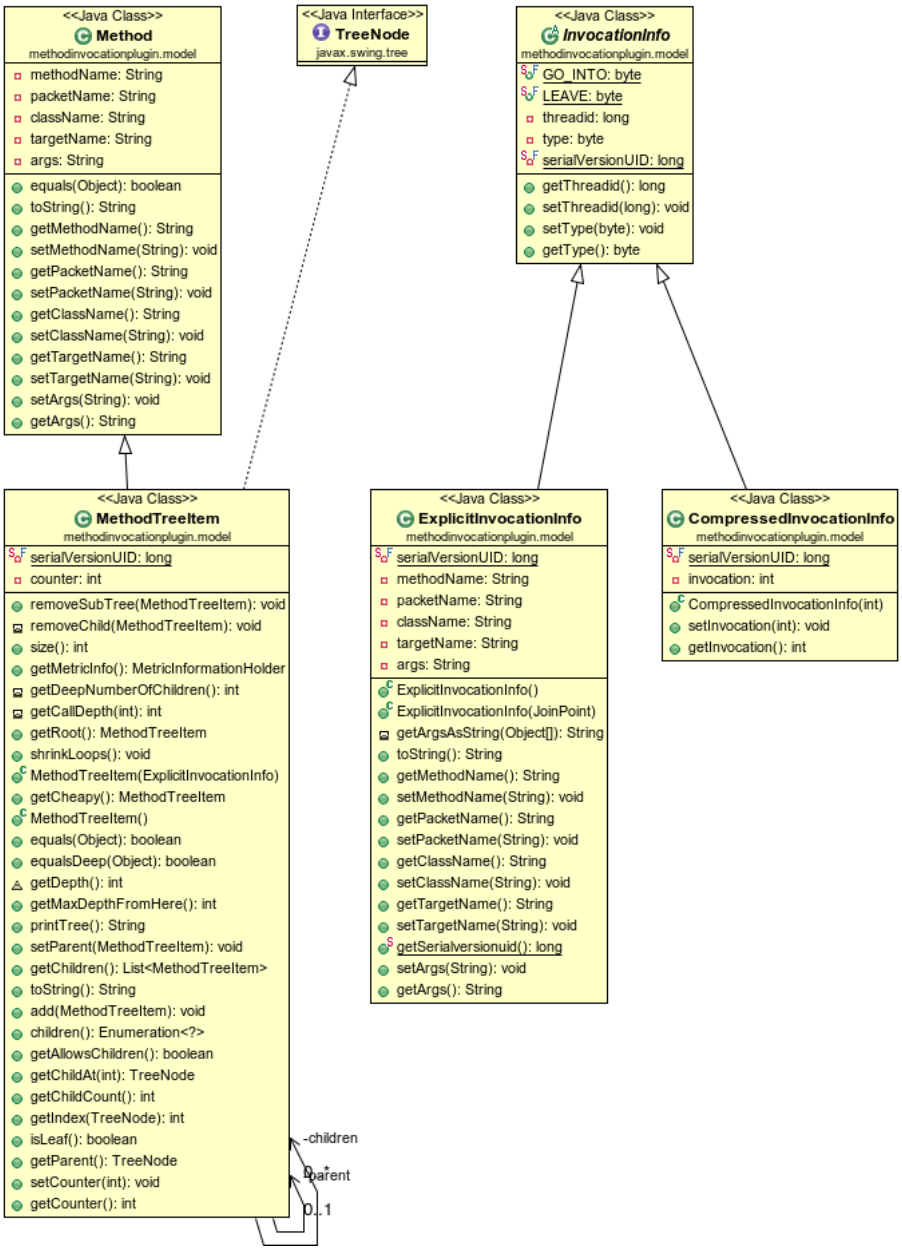

Figure 4.1: Invocation information model

- Definitor
- Before/After indication
- Thread ID

BOA uses a small transmission model because of the fact that the object is serialized and transmitted from the application to BOA over a socket. BOA's client in the running application transmits both the definitor (first declarator) and the actual executed method to give a little bit more detailed information when it comes to the *BugMapper*, while having the definitor available to check for loops more generally.

To represent the invocation tree structure, BOA uses another, more complex model with children and parent relationships. This more complex class is called *MethodTreeItem*. The *MethodTreeItem* implements SWT's TreeNode interface for out of the box representation inside SWT tree components. The legitimation of a tree structure is as follows: Without any invocation context information about a method invocation, the plug-in would lose too much information to provide a helpful set of metrics and invocation information to the developer. BOA could use a flat structure like a list. If for example every feature would be a list of methods that have been executed during runtime, and possibly a number expressing how many invocations occurred in that specific feature, the developer would not see which method calls produce deep call stacks, which different callers a method has, etc. The downside of a tree structure is the fact that it takes more resources to store, display and transmit the information. For that reason, optimizations are made to mitigate this negative aspect.

Whenever a new tree is recorded by BOA, the invocation structure is compressed [8, 11]. For example loops are compressed into items with a value describing the amount of invocations. A loop is recognized as such if one or more invocations in the current item repeat themselves twice or more often. Loops are summarized displaying the number of their invocations in the *GraphicsTree*. BOA ignores the difference between overwritten methods. For example, the two trees presented in Figure 4.2 are considered equal in BOA's loop detection.

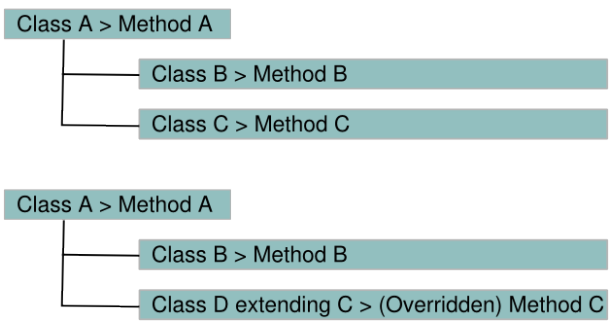

Figure 4.2: Equal subtrees when it comes to loops

The tree is built up in a very natural way on the plug-in side: Every thread of the running application has its own server thread in BOA building up the invocation information. At the beginning, a *MethodTreeItem* named by the thread id is created and stored inside a variable called *currentItem*. Every time a new invocation information arrives, BOA checks whether the item was sent *before* the actual method execution or *after*. In the first case, a new *MethodTreeItem* containing the invocation data is added as a child of the *currentItem*. If the method was executed and it was *after* the actual invocation, BOA puts the parent of the *currentItem* variable into *currentItem*. What BOA does is some kind of preorder, just without knowing the tree yet. We present a graphical interpretation of this process in Figure 4.3, followed by an exemplary source code snippet which would lead to this invocation tree and Table 4.1 explaining the tree generation process step by step.

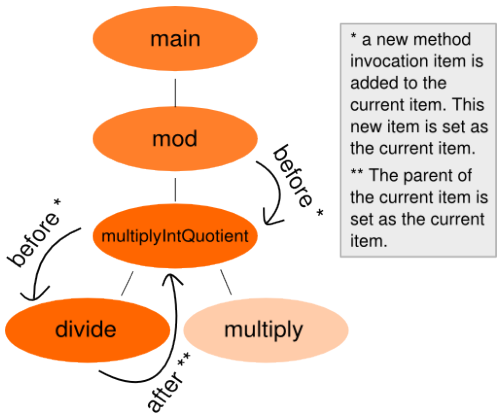

Figure 4.3: Tree creation process

```
public static void main(String[] args) {
        mod(7f, 3f);
}

private static float mod(float aFloat, float divisor) {
        return aFloat - MultiplyIntQuotient(aFloat, divisor);
}

private static float MultiplyIntQuotient(float aFloat, float divisor) {
        return multiply(divisor, (int) divide(aFloat, divisor));
}


private static float divide(float divident, float divisor) {
        return divident / divisor;
}

private static float multiply(float aFloat, float multiplicator) {
        return aFloat * multiplicator;
}
```

After the recording, all the thread roots are added to a new root, a *MethodTreeItem* called Application, and then stored in a feature.

Table 4.1: Tree creation process step by step

Description	**Transmitted invocation item**	**Invocation type**	**Current item**
Initialization of the tree. The current item is an item named by the thread id. In this example, the thread id is MainThread.	-	-	MainThread
Start of the application. Whenever an invocation item of the type before is transmitted, the arriving information is added to the current item as a child and replaces it in the current item variable. After this step, MainThread has 1 child called main and the current item is main.	main	before	main
Call of method mod. The invocation information of mod is set as the current item and added to the item called main as a child. The tree node called main now has one child called mod.	mod	before	mod
Call of method multiplyIntQuotient. The invocation information of multiplyIntQuotient is set as the current item. MultiplyIntQuotient is now a child of mod.	multiply-IntQuotient	before	multiply-IntQuotient
Call of method divide. Divide replaces the current item and is now a child of multiplyIntQuotient.	divide	before	divide
The thread returns to the multiplyIntQuotient method after the execution of divide. The parent of divide (multiplyIntQuotient) is set as the current item because of the after type.	divide	after	multiply-IntQuotient
Call of method multiply. Multiply is added to the children of multiplyIntQuotient. MultiplyIntQuotient has 2 children after this step: divide and multiply.	multiply	before	multiply
...	...	...	...

4.2.2 Examined Dynamic Information Gathering Techniques

Recording feature invocations in Java is tricky, as the language itself does not provide a satisfying possibility out of the box. But there are in fact several different solutions to this problem. In the next three subsections we discuss all the approaches considered possible and tested in this project. These are:

- The Java Debugger
- Byte code injection
- Aspect oriented approaches

Parsing Java Debugger Information

The first setup is using the Java Debugger, also called JDB. The Java Debugger, which actually was activated by a parameter of the Java application launcher *Java* in older versions of the JDK, is able to print out the signature of each method call of a running application. This is in fact sufficient for the earlier described model, but there is already the first drawback: JDB prints out the signature after every enter and exit of a method inside the call stack. This signature must then be parsed, which is time consuming. Also the Virtual Machine runs in debug mode, which does not allow runtime performance enhancements and therefore lowers the performance of the running application.

This is a short snippet of an exemplary run. In this point of execution, the method getBar(), which does not invoke any other method, is executed on the thread called „main". The command *trace methods* triggers the printout of the method invocation information and pauses the running application until the *cont* command is executed. Alternatively the command *trace go methods* does not pause the application.

```
> run foo.bar.Loop
run  foo.bar.Loop
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: trace methods
>
Method entered: "thread=main", foo.bar.Loop.getBar(), line=17 bci=0

main[1] cont
>
Method exited: return value = 1, "thread=main", foo.bar.Loop.getBar(), line=17 bci=3
```

There are three other, less important but bothersome problems with the Java Debugger solution: Although this is not an important requirement, you do not receive invocation information about standard Java library methods that are called. Second, the application first has to start in debug mode before the *trace go methods* command can be triggered. So the invocations start, while the plug-in might not be ready yet. The process needs some kind of synchronization to ensure that BOA does not lose the first invocations. Of course BOA could record all invocations to a

log file, parse and pick out the required information later on. But therefore BOA would have to add precise nanosecond time-stamps to the log file, which would again slow down the recording process. And last but not least: It is difficult to attach the JDB task to the console and task manager of the IDE, as BOA would have to filter the call signatures out from the actual printouts to a console. Otherwise the console output ends up in a mix of the applications print out to the standard output stream and the JDB logs.

Drawbacks of the Java Debugger:

- The slowdown of the Virtual Machine due to not available runtime enhancements
- The log parsing and filtering problems

Using Javassist to Inject Byte-Code

Another solution we examined is to inject byte code into compiled classes. By injecting the information transmission code and a technique that adds information about the method and class the code was injected into, the invocation information can be transferred directly to BOA. This approach is good because once the code injected, the developer can just run the binaries as usual. Here's an example of a simple printout statement that is injected into a method with the name *myMethod* inside the class called *myClass* using Javassist.

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get(myClass);
CtMethod cm = cc.getDeclaredMethod(myMethod, new CtClass[0]);
cm.insertBefore("{ System.out.println(\\"Here we go!\\"); }");
cc.writeFile();
```

The problem with this technique is the low level of abstraction. For example, byte code snippets can be added at the top and the end of a method's code. But if the method returns a value, this value has to be held back by storing it into a temporary variable before transmitting the information. Otherwise the injection generates dead code after a return. Even more complex is the fact that if an exception is thrown, the byte code at the end of the method is not invoked as the interpreter never steps over these lines of byte code. This leads to invalid invocation trees because no *after* is sent. Besides the code at the beginning and the end of a method, BOA needs to check whether the method has thrown an exception, and if so, still send the *after* invocation info. Injecting into catch-parentheses properly is difficult because inside a catch parenthesis, another exception can be thrown or exceptions can be uncatched, for example an *IndexOutOfBoundsException* or other exceptions extending *RuntimeException*.

Another more general problem of injecting data gathering code into byte code is that the resulting class file will be bigger than the original class file. So the Virtual Machine will have to handle more data, which decreases the performance of the software system. And, as we will see later, injecting byte code into huge classes can lead to invalid classes, as the size of a class file is limited by the class loader or by the Virtual Machine respectively.

Summarized, the reasons why byte code injection using Javassist is a suboptimal solution:

- Lack of abstraction
- Class file size (performance issues and limit of maximum class file size)

Aspect Oriented Approach

Aspect oriented programming (AOP) is a methodology which delivers concepts, constructs and many benefits to modularize cross cutting concerns in software engineering. AOP introduces the ability to weave together independent source code fragments with actual code.

AspectJ The most widely used aspect oriented framework for Java is called AspectJ[1]. The first step was to create the aspect, which is woven into the application's byte-code. We present the aspect used in BOA in the next paragraph. As we will see, the aspect is woven into every method enter and exit, so BOA again receives two invocation information packets per method call, one when the thread enters the method, and one when it leaves the method. Again it is simple and elegant to build up an invocation tree with this information.

To make sure that most classes are weavable and therefore their invocations can be captured, we tried to make the code which is injected as small as possible. As BOA's aspects inject code into classes, the class file size will grow and the problem of the Virtual Machine's maximum size of a class file does apply for this solution too. The reduction of the injected code has the nice side-effect that not only more classes are able to be recorded, but also the increase of memory load of the resulting application is smaller, which results in a slightly faster running application. In our first runs, the aspects were woven into the byte code of the application. So first we built a jar file, which was then passed to the AspectJ byte code weaver. The result was a completely woven jar file, which any Java runtime with the AspectJ runtime would be able to start. Due to the fact that the developer probably does not want to weave every library in the project on every run, we used the AspectJ load time weaver which weaves the aspect into the classes flexibly on load-time with a custom class loader. This load time technique makes the application a bit slower when new classes are loaded, like for example just right at the start of the application, but in general increases the overall performance when only the desired libraries are woven or not all libraries inside the class path are loaded.

One important feature of AspectJ is that the after advice automatically triggers the method when an exception is thrown. We never end up with incorrect trees when an exception is thrown. In this project we used AspectJ in version 1.6 including weaver version 7. The drawback of AspectJ is the fact that it is not easy to weave into the Java Runtime classes. There are many reasons why, for example:

- Native code cannot be woven

[1] http://www.eclipse.org/aspectj/

- Class files could get too big after weaving

To reach almost every class of the Java runtime, we tried to instrument it. We used the list of the Major project's implementation, about which we talk in the next Chapter, to get the first few classes to exclude in BOA's aspect. Then we run the AspectJ compiler to weave the aspect into Sun's Java runtime. After a few crashing runs unveiling more classes that were not weavable using BOA's aspect in the Java Runtime 1.6, we finally seemed to have all uninstrumentable classes excluded. But the problem arising after that was that the byte code weaving seemed to use a lot of RAM. We tried to weave the runtime library on a server with 4GB of RAM and different Virtual Machine arguments, but still we were not able to finish the instrumentation due to a memory exception. As the recording of all standard Java classes was not an important requirement, we did not continue these tests.

The aspect used in BOA. The most important thing about aspect oriented programming is the aspect which is woven into the byte code of an application. We now take a look at the aspect we used in BOA.

```
public aspect TracePointAspect {
        pointcut tracePoints(): call(* *..*.*(..)) && !within(ch.byteality.ba..*) && !
             within(methodinvocationplugin..*);

        before() : tracePoints() {
                try {
                        SimpleSerializerClient.getClient().pushInfo(thisJoinPoint, gointo
                             );
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        after() : tracePoints() {
                try {
                        SimpleSerializerClient.getClient().pushInfo(thisJoinPoint, leave)
                             ;
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

There are two advices, one *before*, and one *after* any method body specified in the tracepoint. What the injected code actually does is send the invocation information over the serialization client to the plug-in. The *gointo* and *leave* arguments describe whether a method is entered or left by the current thread. The thread id, which is the unique indicator for our thread differentiation, is added before sending the generated invocation information in the *pushInfo* method.
To ensure that no infinite loop is created on runtime, all classes from within this project are excluded. Also many Java standard classes and packages are excluded, but their exclusion is configured in the aop.xml, which is written by BOA's custom launcher, as we will see in Chapter 4.3.

The *SimpleSerializerClient*'s *getClient* method returns a thread-local *SimpleSerializerClient* instance. This way we ensure that every Thread has its own client and no synchronization is

needed. Further information about the use and the specific responsibility of a class are available in the developer's guide.

MAJOR As described in the state of the art Chapter, Major is a tool offering techniques to weave even into the Java runtime library. Therefore Major is able to deliver information about invocations on the Java runtime.

BOA used the cct-slow aspect which creates the calling-context tree. The cct-slow aspect was used in the Senseo project. A weaved Java application is able to capture metrics like new instances created. Major sends the data over a socket in a predefined interval. We used 5 seconds. While running, Major builds up a shadow stack and sends the whole information to the server, which was in our case BOA.

We did not use Major in BOA because of the fact that our Major launch configuration took more than twice the time to prepare and start the application compared to our AspectJ solution. Also it was difficult to attach the running application to the Eclipse IDE the way a developer is used to, for example the functionalities of the console or the kill-button of Eclipse.

4.3 Custom Data Transmission from the Application to BOA

First we implemented the data transmission using Java RMI. But quickly we found out that the use of RMI would slow down the recording and therefore the application. Also the whole RMI system seemed to be too extensive. We needed something more lightweight. So we chose to create our own data transmission system using the Java serialization technique.

To transmit the data from the application to the plug-in, BOA uses a simple client server architecture. The Eclipse plug-in is actually the server, to which the running application connects. The reason why we are using sockets is that the application should run by itself, as loose from the plug-in as possible. Only that way possible side-effects can be avoided.

As the application handles all the user requests and is therefore by nature the performance bottleneck, the BOA client which is added to the application is as thin as possible. It does nothing with the data but sending it over the serialization stream. The whole tree generation and tree item instantiation is done on the server respectively on plug-in side.

To make sure that no concurrency problems occur, every thread has its own thread-safe socket to transmit the data. On plug-in side, we only have to draw a difference between the different sockets, not about which thread actually triggered the invocation. We send the thread id with every invocation information, as the stream between the client socket of a thread and the server socket of the plug-in might break. In this case, a new connection is established from the client and

therefore the server would not know whether it is the same thread again or not. When the thread id is available, it is no problem to assign the invocation information to the correct invocation tree.

After a new invocation information arrives, the controller fetches the current tree item from a hash-table, of which the thread-id is the key value. To that tree item, the controller will add a new sub-item or move to its parent, as describe in Section 4.2.1. Synchronization is not needed on plug-in-side, as two threads never try to modify the same item.

To reduce the transmission load, the invocation data of a certain method is only sent once to the plug-in completely. After that first transmission, only an index number telling BOA what method invocation was detected is transmitted. The complete information packet is called *ExplicitMethodInformation*, while the index representation of such an item in the cache is called *CompressedInvocationInformation*. The *CompressedInvocationInformation* only holds the thread id, the type (which can be after or before) and the index of the corresponding *ExplicitMethodInformation*, which is an integer. Because of the fact that the smaller *CompressedInvocationInformation* is deserialized quicker than the complete information holding six strings and a long value, and because of the fact that Java does not deserialize serially, the *CompressedInvocationInformation* item is often available before the necessary *ExplicitMethodInformation* is deserialized. To solve this issue, BOA queues the *CompressedInvocationInformation* if the *ExplicitMethodInformation* at the specified index is not yet available and periodically checks if new information solves index-references to pending *ExplicitMethodInformation* in the queue. If so, the complete *ExplicitMethodInformation* is cloned from the cache array and attached to the tree where the compressed item would have gone before.

The cache mechanism works as follows: both BOA and BOA's client parts inside the running application hold a cache array of complete invocation data. Whenever an invocation is triggered, the BOA client checks the cache array whether the invocation was already sent to BOA or not. In the first case, only the index number of the item inside the cache array is transmitted within a *CompressedInvocationInformation* item. Otherwise, the whole *ExplicitMethodInformation* item is sent and stored inside the cache array for further compressed transmission use. On plug-in side, BOA does the same: whenever a complete information arrives, it is added to the current invocation tree and stored inside the cache array. When, a few seconds later, only a *CompressedInvocationInformation* passes the socket, the method item inside the cache array at the index from within the *CompressedInvocationInformation* is cloned and added to the current invocation tree. Clearly, items are only added to the tree if the queue is empty. Otherwise BOA might skip invocations, which would lead to an incorrect invocation tree. Figure 4.4 shows based on the example execution tree we used in Chapter 4.2 (Figure 4.3) how the cache array is filled with *ExplicitMethodInformation* objects and how the *CompressedInvocationInformation* objects are transformed into *ExplicitMethodInformation* objects again.

The lazy lookup works as follows: As soon as a *CompressedInvocationInformation* arrives, that means an integer value representing the cache array index in which the *ExplicitMethodInformation* is held, BOA checks whether the item in the cache array is null or not. If it is not null, then

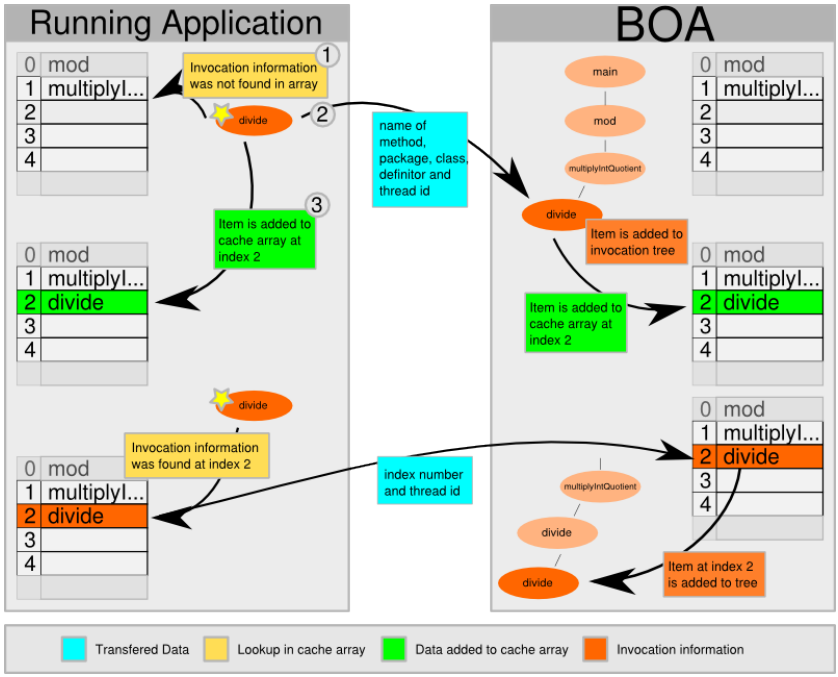

Figure 4.4: The cache array filling process

it is added to the tree or to the queue depending on whether there are already items inside the queue or not. Otherwise BOA would skip invocations that are currently inside the queue. The same applies to the *ExplicitMethodInformation*, with one difference: the information is also being added to the cache array, and BOA then tries to solve the references of the *CompressedInvocationInformation* indexes to the cache array inside the queue. Again, after that procedure, if the queue is empty, the invocation information is being added to the tree and otherwise to the queue. The following Figure 4.5 contains a flow chart describing the queue mechanism in detail.

4.4 Data Storage

It is important for the daily use to enable the developer to store the features in files or anywhere in the project in a way they can be versionized in a version control system such as SVN or Git. If not, feature analysis could not efficiently be done during the whole development process. A developer should be able to check out a project and have the recorded features for that specific state of the source code of the project right with the code. Only that way can we ensure that the feature representation is really used and useful during the software development.

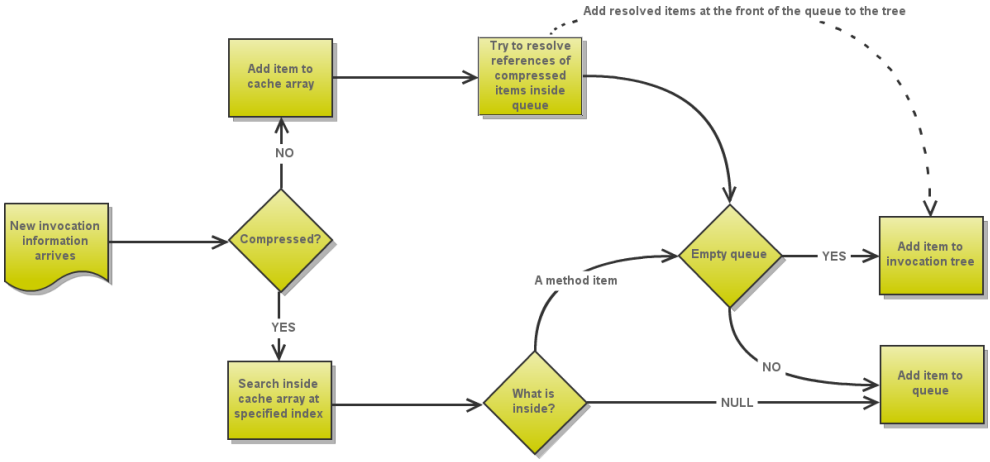

Figure 4.5: Queue process for compressed and uncompressed items

The method invocation trees are stored in a simple XML file called boa.xml. That way the trees can be imported or analyzed by other tools. Also because of the human readable format, they can be more easily repaired or merged. To store and read the XML data structure, BOA uses Dom4j, a very common Java library to read and write XML files.

4.5 Plug-in Design

4.5.1 Specifications

One requirement is the platform independency which should be ensured for this plug-in, as Eclipse is used on several platforms. Also it should embed in the standard views of Eclipse and fit the habits of a software developer. It should run under the newest stable release of Eclipse and Java available. The installation should be possible using the standard plug-in mechanism of Eclipse.

4.5.2 Eclipse Plug-in

The Eclipse rich client platform allows one to add plug-ins to any Eclipse environment by simply writing an OSGi bundle. For Eclipse this means creating a plug-in development project. Creating views or extending other parts of Eclipse is mostly done by implementing an interface or extending an abstract class of the Eclipse project and registering the new component inside the plugin.xml. For example the run configuration, which is added by implementing the *ILaunchDelegate* interface for the actual launch process controller and the *ILaunchConfigurationTabGroup* for the run configuration user interface. Eclipse OSGi implementation Equinox

handles dependencies, like in our case the Java development tools JDt, the debug core or the AspectJ plug-in called AJDt.
Eclipse plug-ins are updated over an update site, which responds to requests with an XML of update information. The IDE can update the plug-in itself without the developer having to interact or install anything manually.

Feature representation. As features in BOA are method invocations in a tree structure, the most simple way to represent a feature is to show the invocation tree. BOA represents a simple tree using the standard SWT tree widget. Besides that we wanted to integrate a more decent representation of method invocations. So we implemented a tree graph painter called *GraphicalTree* using a horizontal compact tree layout and SWT's canvas to paint on. To not overload the view and make it quicker to render, BOA collapses all tree items of depth four and more at the beginning. The following Figure 4.6 is the UML of the custom tree component. A tree is generated from a root *MethodTreeItem*, and transformed into a view model containing bounds, lines and rectangles. The *FeatureDedicationMatrix* is a custom component too, painting colored squares on a SWT canvas.

The heatmap used to represent the feature dedication of methods in the *GraphicalTree* and the *FeatureDedicationMatrix* is based on two images, one used for the background color and one used for text, for example the name of the method in the *GraphicalTree*. Depending on the feature dedication of a method, a color from inside the image is selected. Implementing the heatmap with images makes the heatmap representation easy to change, even in installations. Figure 4.7 shows the two heatmap images used by default in BOA.

Integrating information into Eclipse. BOA primarily features a single view, the *FeatureView*. This view is meant to be placed near the console and definition view at the bottom of Eclipse, just beneath the source code editor. This view features the start/stop button, the *FeatureTree*, the *SimpleTree*, the *GraphicalTree* view and the *MethodDedicationMatrix* for every selected feature. Also from inside this view the developer can open the *BugMapper* dialog to analyze the bug relevance of the selected feature's methods. We added new markers to the standard views of the IDE, precisely to the standard Java editor. For example when selecting a feature, a marker is placed on every class and method that was touched by this feature. That way the feature view supports the developer on navigating inside the source code. Figure 4.8 presents Eclipse's Java editor with markers on the left and right side. The left side simply marks currently visible methods used in the selected feature with an Eclipse icon, while on the right side, all methods of this class used in the selected feature are visible in an overview.

Launching an application using a custom launcher. We now take a look at how BOA launches the application with dynamic information gathering support.

We tried to reuse as much as possible from Eclipse to ensure that the flexibility of BOA is as big as the one of Eclipse's standard Java launcher. The source code of the application

is compiled into a directory with the standard Java development tools from Eclipse. From the project preferences, BOA reads all class path entries like libraries and other projects the application relies on. Then BOA's launcher creates the aop.xml, which contains all information about the aspects that should be woven on load-time into the application, the information about excluded classes and other weaver options like the verbosity. All project internal classes used to communicate with the Eclipse plug-in, the whole model of BOA and about 30 classes of the Java standard library are excluded automatically from the aspect weaving. If the user selects the heuristics check inside the run configuration's *ExcludeTab*, additionally each class file of the project is checked whether its resulting class file is potentially too big after load time weaving. This mechanism relies on the fact that the resulting size of the compiled class is more or less defined by the number of lines of code. BOA multiplies the number of methods in the class file that is woven by ten and adds this value to the number of lines of code of the file. If this resulting number of lines of code is bigger than 5000, the class is excluded from weaving. This is a very simple check. It is not very precise as a line of code does not always lead to the same number of bytes in a class file, but, probably because of the fact that classes with such a big number of lines are statistical outliers, in the tested scenarios this worked pretty well. Also the developer can add classes to exclude by himself in the run configuration dialog and disable this heuristic check. To simplify this manual entry, when an application is executed, the names of the excluded class files are copied to the clipboard, so the developer can simply open the run configuration and press ctrl+V to insert the classes BOA's heuristic check would exclude. After the generation of the aop.xml, BOA launches the application identically to the AspectJ plug-in.

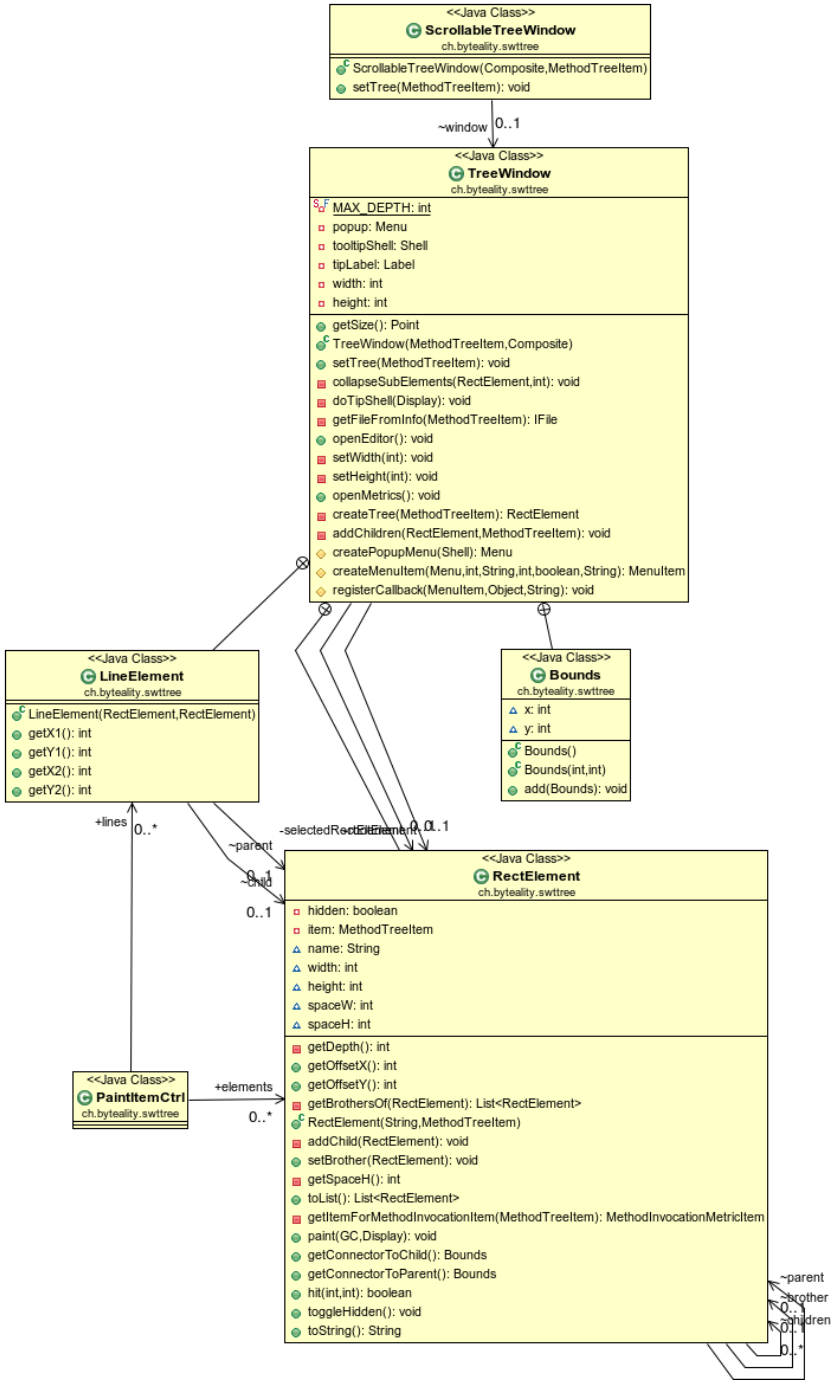

Figure 4.6: The UML of the custom STW tree used in the *GraphicalTree* representation

Figure 4.7: The heatmap color images. Left: Text, Right: Background

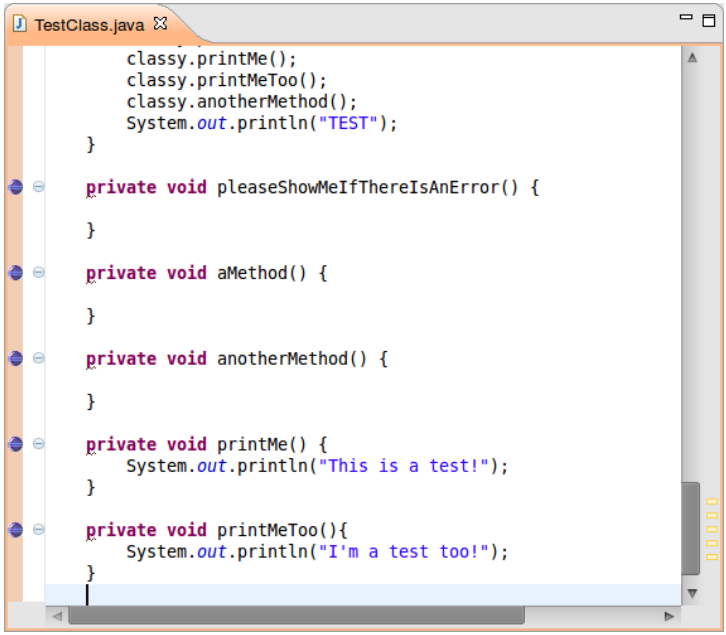

Figure 4.8: With markers annotated methods inside the Java editor of Eclipse

Chapter 5

Evaluation

As usability and speed are two of the main goals of BOA, we benchmarked the feature recording and tested the *BugMapper* on a sample scenario. We tested BOA with a real-world application which is currently under development and with jEdit and Pixelitor, two software systems hosted on SourceForge.

5.1 Analyzed projects

The three projects analyzed in this Chapter are:

jEdit. jEdit is a programmer's text editor supporting source code highlighting for Java and for about 130 other programming languages. The application is written entirely in Java and the project contains about 550 Java source files. jEdit is a SourceForge project [1] since 2001.
We used jEdit to benchmark different recording techniques.

Pixelitor. Pixelitor [2] is an open source image editor supporting many features like layers, filters or color adjustments. The project is hosted on sourceforge and contains about 530 Java source files.
Like jEdit, we used Pixelitor for our benchmarks.

ScanMe ScanMe is the internal project name of an application written for the Swiss rescue services. It is a Swing application used to trace patients during serious accidents like plane crashes, train accidents and other casualties. It uses Hibernate to store data inside a Derby database, Swing with custom components, the Restlet libraries to transmit data to a central server

[1] http://sourceforge.net/projects/jedit/
[2] http://pixelitor.sourceforge.net/

and about 25 other Java libraries. Information about patients is transmitted over the Internet to a server which is able to provide hospitals with information concerning arriving patients or the number of available transport vehicles like ambulances or helicopters. The advantage of ScanMe is the support of 2D barcodes for patients, vehicles and transport destinations. Registering the relocation of a patient is done with 5 scans, rather than by typing data in a computer by hand or writing it down on a sheet of paper. The transport information is available at once for all hospitals in Switzerland over a web-based application.
The ScanMe application was used to evaluate the *BugMapper*.

We first present our benchmarks and afterwards show our results of the test with the *BugMapper*.

5.2 Benchmarks.

To check whether the solution is suitable to replace the standard launch configuration and integrate into the standard work-flows of developers or not, we did a few benchmarks with and without feature gathering activated. We timed the start up times of jEdit and Pixelitor with a stopwatch.

The test environment. The scenarios presented in this Chapter were all executed on the following computer system:

- Lenovo T500
- 2 gigabytes of RAM
- Intel Core2Duo P8600 2x2.4Ghz (4788.2 Bogomips)

The software environment was as follows:

- Ubuntu Linux 10.04 (32Bit / i686), Kernel 2.6.32
- Java: 1.6.0_22-b44 (Sun)

The benchmarked techniques. On the following pages we present benchmark results of the following three dynamic information gathering techniques:

- Java Debugger
- BOA's final implementation (using AspectJ)
- Major using the cct-slow aspect

jEdit and Pixelitor started with standard Java run configuration from within Eclipse. The following Table 5.1 shows how long it took to start jEdit and Pixelitor from within Eclipse using the standard Java run configuration.

Table 5.1: jEdit and Pixelitor from within Eclipse using the standard Java run configuration

Run	Start jEdit [ms]	Start Pixelitor [ms]
1	3600	2430
2	2350	2360
3	2700	2350
4	2610	2410
5	2720	2410
Mean	2796	2392

The time used to start jEdit and Pixelitor is between two and three seconds, with Pixelitor being a little bit faster. The average time spent in these jEdit and Pixelitor runs is used to calculate the overheads of the following tested feature recording implementations. The overhead is the ratio between the mean time spent in the corresponding standard Java run and the time used in the measured run. An overhead of one means the measured run took exactly as long as the average Java run.

jEdit and Pixelitor started with the Java Debugger. The following durations were measured when starting jEdit with the Java Debugger, using the *trace go method* command to record invocations without halting. We present our measurements of the Java Debugger in Table 5.2.

Table 5.2: jEdit and Pixelitor started with the Java Debugger

Run	Start jEdit [ms]	Overhead jEdit	Start Pixelitor [ms]	Overhead Pixelitor
1	111870	40.1	72260	30.2
2	109540	39.3	73290	30.6
3	113680	40.7	71860	30.0
4	110010	39.4	72220	30.1
5	111510	39.9	72640	30.4
Mean	111322	39.9	72454	30.3

jEdit and Pixelitor with BOA and deactivated recording. The time was measured from the beginning when the application launcher was clicked with heuristics and recording enabled but recording deactivated (start button inside the *FeatureView* not clicked before launching, so no recording of invocations during the start phase of jEdit but the possibility to start recording anytime by clicking the start button).

The following class files inside the jEdit project were automatically excluded by the heuristics check:

- com.microstar.xml.XmlParser
- org.gjt.sp.jedit.jEdit
- org.gjt.sp.jedit.bsh.Parser
- org.gjt.sp.jedit.textarea.TextArea

Neither the heuristic check excluded class files of Pixelitor, nor had we to exclude files manually.

Table 5.3 lists our measured results of BOA's *Data Gathering* run configuration with deactivated recording.

Table 5.3: Time spent to start jEdit and Pixelitor with the *Data Gathering* run configuration and deactivated recording

Run	Start jEdit [ms]	Overhead jEdit	Start Pixelitor [ms]	Overhead Pixelitor
1	23140	8.3	13380	5.6
2	18830	6.7	11360	4.8
3	19420	6.9	11410	4.8
4	19390	6.9	12190	5.1
5	18510	6.6	11830	4.9
Mean	19858	7.1	12034	5.0

The fact that the first run was more than four seconds slower is because of the heuristics check which finishes faster for all subsequent runs. As there is no mechanism inside BOA to cache the results of the heuristics check, we assume that this speed up is the consequence of another cache mechanism, for instance by providing faster access to files after they have been read for the first time. We can say for sure that it is the heuristics check that takes more time in the first run, because in the second run the exclude information dialog comes up almost immediately.

jEdit and Pixelitor with BOA and activated recording. Again we tested jEdit and Pixelitor with BOA, but this time with recording activated. That means we clicked the start button before we actually started jEdit.

Again the following class files from jEdit were automatically excluded by the heuristics check:

- com.microstar.xml.XmlParser
- org.gjt.sp.jedit.jEdit
- org.gjt.sp.jedit.bsh.Parser

- org.gjt.sp.jedit.textarea.TextArea

Table 5.4 shows the time spent to start jEdit and Pixelitor with BOA and activated recording.

Table 5.4: Time spent to start jEdit and Pixelitor with the *Data Gathering* run configuration and activated recording

Run	Start jEdit [ms]	Overhead jEdit	Start Pixelitor [ms]	Overhead Pixelitor
1	26900	9.6	16130	6.7
2	22380	8.0	14200	5.9
3	22370	8.0	13980	5.8
4	23010	8.2	14180	5.9
5	22710	8.1	14010	5.9
Mean	23474	8.4	14500	6.1

As in the first BOA test runs, we measure three to four seconds less time consumption after the first run.

jEdit and Pixelitor with Major. First we measured the entire time taken by our experimental Major run implementation to compile the class files, create the jar file and start it with Major using the cct+slow aspect to record dynamic information. The following Table 5.5 shows the results of the first Major tests.

Table 5.5: Time spent to start jEdit and Pixelitor with Major.

Run	Start jEdit [ms]	Overhead jEdit	Start Pixelitor [ms]	Overhead Pixelitor
1	90210	32.3	34220	14.3
2	73200	26.2	31790	13.3
3	76030	27.2	30910	12.9
4	75010	26.8	31200	13.0
5	73870	26.4	31110	13.0
Mean	77664	27.8	31846	13.3

As we saw in the log file, the jar building took about half of the starting-time. The configuration of BOA's experimental Major run configuration is definitely not optimal, as not the entire build from the last launch has to be cleaned up. For example not every class has to be recompiled on every single run or even the jar file could be reused in some cases. The cct+slow aspects we used also delivers much more information than BOA's *Data Gathering*, for example information about the invocation inside the Java runtime libraries, and metrics that BOA's aspect does not deliver. When Major has properly started and jEdit or Pixelitor are launched, the performance of the application seems to be better than with BOA's final implementation. It is just the fact

that the preparations take a while and Major takes some time to start that makes these statistics look as if Major was in fact slower. To be fair and to show that Major is very fast, we measured the time spent to completely launch jEdit and Pixelitor from the point when all preparations were done. That means from the point when Major starts up. Table 5.6 presents the lower time consumption of Major ignoring the preparations.

Table 5.6: Time spent to start jEdit and Pixelitor with Major. No preparations.

Run	Start jEdit [ms]	Overhead jEdit	Start Pixelitor [ms]	Overhead Pixelitor
1	30210	10.8	11000	4.6
2	28560	10.2	11200	4.7
3	29410	10.5	11010	4.6
4	28210	10.1	11470	4.8
5	28850	10.3	10940	4.6
Mean	29048	10.4	11124	4.7

As we can see, Major is already very fast now. JEdit only takes about 15 to 20 seconds to start. The other ten to 15 seconds are used to initialize Major. Major hence scales definitely better than BOA's *Data Gathering*. Pixelitor even starts faster with Major than with BOA's own solution.

Summary. We now summarize the average time spent for every technique we benchmarked in Table 5.7.

Table 5.7: Summary of the benchmarks (OH stands for overhead)

Run	jEdit [ms]	OH jEdit	Pixelitor [ms]	OH Pixelitor
Normal Java run	2796	1	2392	1
Java Debugger	111322	39.9	72454	30.3
BOA, recording deactivated	19858	7.1	12034	5.0
BOA,recording activated	23474	8.4	14500	6.1
Major with preparations	77664	27.8	31846	13.3
Major without preparations	29048	10.4	11124	4.7

As we can see, both BOA techniques are fast. Sadly, the difference between recording activated and recording deactivated is quiet small. Running BOA's *Data Gathering* run configuration with deactivated recording only improves the speed by about 15 to 20 percent. Still the speed of BOA is the best out of the dynamic information gathering techniques we examined when ignoring the Major runs without preparations.

5.3 Detecting Faulty Methods

To evaluate the support of the plug-in concerning the help provided by BOA detecting faulty methods, we tested the *BugMapper* in a bug-fixing scenario.

Preparations. To check whether BOA's *BugMapper* helps in finding faulty methods or not, we first checked out the project from the subversion repository in a version in which we knew about a bug in the export functionality, located in a barcode generator. The barcode generators implement a simple interface with a method returning a byte array representing the binary image data of a barcode for a certain object in the system. We captured all export features and ten additional random features of the ScanMe application and stored them in the XML file.

Finding the Faulty Method We opened the *BugMapper* and tested two scenarios: The buggy feature against all other export features and the buggy feature against all features we recorded. Our defined goal was that the faulty method was in the first 5 methods listed by the *BugMapper*.

The results of the *BugMapper*. The *BugMapper* listed the faulty method on the fifth place when we tested the buggy feature against all export functionalities, and on the third place when we tested against all recorded features in the software system. The first 7 methods in the list had all the same bug relevance in the first test, while in the second test, two methods left their leading positions in the ranking. The two methods were the getter and the setter of a field of an exported object. As the rest of the export functionalities did not touch this object, the bug relevance for these accessors was very high. Due to the fact that a feature updating local information from the Internet touched these two methods, their bug relevance went down and we produced a better result when testing against all recorded features.

Conclusions. The conclusions of these results are that a developer should not always test the bug relevance against only a few other, very specialized features. Even though the check of a getter or setter is easy, the problem can also arise with other methods than accessors. Over all, we are happy with the result. Even though the *BugMapper* listed other methods first, the faulty method was in the first five listed methods in both scenarios.

Chapter 6

Conclusions and Perspectives

6.1 Conclusions

During this project we developed BOA, a plug-in to record, visualize and analyze features inside the Eclipse IDE. We used different feature recording techniques and selected the one fitting our formulated requirements best. BOA's final recording technique uses AspectJ. We created a custom run configuration providing BOA's feature recording technique called *Data Gathering*. We analyzed different feature recording techniques in an evaluation. Also we evaluated the Bug Relevance, a metric we implemented and which describes the relevance of a method for a specified bug. The transmission technique used in BOA was written completely from the ground up after using Java RMI. BOA is a fully functional plug-in for Eclipse and is released as a open source project on SourceForge.

6.2 Perspectives

BOA demonstrates that feature recording, analysis and representation inside Eclipse can be done in a way that developers can use it in their Java projects. The feature gathering technique as well as the user interface certainly need enhancements. The following future work is planned for BOA:

Faster data gathering technique. The data gathering technique could be improved regarding speed. The next step is to improve the speed of an application with dynamic information recording deactivated. We should be able to improve the speed by adding an advice checking the recording state of the plug-in when the application starts up, and that way reducing the lookup time on every invocation due to the fact that the BOA client currently has to check this

in a suboptimal way. If we can be sure that no recording is requested from the beginning, we can trust the recording-socket to inform the client when recording should start.

Improve the bug relevance. In our tests we discovered that in some cases, many methods are listed on top of the list with the same bug relevance. To improve the bug relevance, other factors like the number of lines of code inside a method could be considered. Even information from repositories like the age of a method or the number of developers working on a piece of code could be used to improve the metric.

Gathering more information. The recognition and transmission of information about other performance relevant facts like the number of created objects or the execution time of a methods execution could be integrated into BOA's models and views.

Appendix A

Installation

We assume that the user has already installed the following software on his computer:

- JDK 1.6 from Sun / Oracle[1]
- Eclipse Classic 3.6[2]

The best way to install BOA is to use the update site mechanism of Eclipse. That way not only the installation process is processed almost automatically, but also updates can automatically be downloaded and installed by Eclipse.
First a new site location has to be set up. Click *Help - Install new software* in Eclipse's menu bar. In the following window, click the *add* button. Enter a name (e.g. BOA update site) and the update site URL of BOA. The current update site URL can be found on the SourceForge page[3]. Click OK and select the site you just added in the drop-down at the top of the installation window. A tree-like category structure should be available now. Select the plug-in called BOA and hit next. Figure A.1 presents how the installation window could look like before clicking next.

The next view will display the plug-ins installed by the operation. Clicking next again brings up the licence. Accept the license agreement by selecting the upper radio button and hit next again to install BOA.
Further help is available in Eclipse by selecting *Help - Install New Software*.

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[2] http://www.eclipse.org/downloads/moreinfo/classic.php
[3] http://boaforeclipse.sourceforge.net/

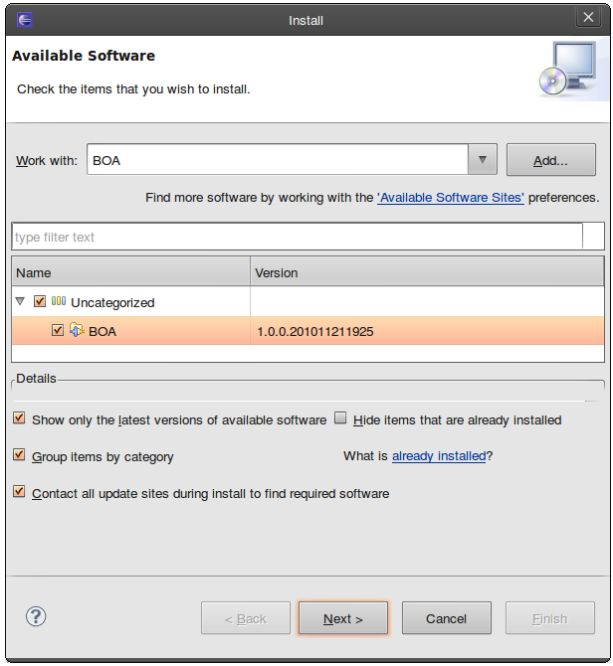

Figure A.1: Eclipse's installation window with BOA selected for installation

Appendix B

User's Guide

We now present two basic tasks that can be done with BOA. In the following tutorials we assume BOA is properly set up and the developer is developing a standard Java application he wants to record features from.

Recording and Representing a Feature

To start a Java application from within Eclipse with feature recording enabled, a *Data Gathering* run configuration has to be configured first. To create a run configuration for a project, right-click on the main-type of the project. The main-type is the Java class containing the static main method starting the application. In the appearing context menu, click on *Run As*. In the sub-menu, select *Run Configurations...* to get to Eclipse's run configurator which enables the user to manage the run configurations.

In the left column presenting all possible run configurations of Eclipse with their corresponding already saved run configurations, right-click on *Feature Gathering* to select *New*. A new run configuration appears. The Java application specific details under *Main* should automatically be filled out by Eclipse. If not, use the configuration details from Eclipse's standard Java run configuration used so far to launch the application.

Next, open the tab called *LTW Aspectpath*. Click on *User Entries* and afterwards on the button on the right of the window called *Add project*. Select all projects related with the application as well as the project itself and click *OK*. The next step is to make sure the heuristic check to exclude big Java classes is enabled. To make sure, open the *Weaver Excludes* tab and select the *Use Heuristics* check-box. Click apply and *Run* to launch the application with feature recording enabled.

The application should start up now. Referencing our benchmarks, the speed of the application should be around 7 times slower. When the application has finished loading and all preparations like loading files or creating a specific application state are done, go to the *FeatureView* and hit the start button. From now on, every method invocation inside the Java application will be recorded. Execute the desired feature and hit stop again in the *FeatureView*.

A new dialog asking for information about the feature appears. Fill out the information to store the newly recorded feature. Now, other features can be recorded. Therefore hit start again, just as in the first case. When all the features are recorded, close the application from within the application or kill it with the red button in the top-right corner of Eclipse's built in console.
We recommend to create the category structure first, and then start recording. The categories should order the features in a way that features in the same category fulfill similar demands. For example, rather put all export functionalities in one category than all features handling operations on the User object of a project. As a consequence, features are easy to find and in a comfortable structure for further use with the *BugMapper*.
Double clicking on a feature in the *FeatureTree* view opens the selected feature in all other representations of the *FeatureView*.

Searching a Bug with BOA

We assume that all features of the analyzed application are already recorded. To start the *BugMapper*, click the *Map a Bug* button in the *FeatureView*. A new dialog asking for information about the bug appearance opens up. The two trees presented are used to select the features in which the bug arises and in which not. First, select all features in which the bug arises. Features as well as entire categories containing sub categories and features can be selected.

In the second tree, select the features or categories in which the bug did not arise. Do not select features that were not tested yet or categories containing such. In our tests, we often received better results when selecting only bug free features in the same domain as the bug affected, for example all export features.

The *BugMapper* will create a list of methods ordered by their bug relevance when the *Map* button is clicked. Double click on a method inside the list to navigate to the corresponding source code fragment inside the Java source code editor.

Appendix C

Developer's Guide

In this Chapter we will shortly introduce the most important packages and classes of BOA. The description should help a developer when it comes to contribute code.

ch.byteality.swttree

The custom SWT base tree used for the *GraphicalTree* representation in the *FeatureView* is completely inside this package. It depends on the *MethodInvocationTree* model of BOA.

TreeWindow The *TreeWindow* renders the *GraphicalTree* and contains three internal classes.

Bounds *Bounds* is a class that encapsulates the boundary variable of a rendered item

LineElement The *LineElement* is used to create the connecting lines between two *RectElements*.

RectElement The *RectElement* represents a renderable *TreeNode*. It also contains the whole layout logic.

A *TreeWindow* is updated using the *setTree(MethodTreeItem)* method.

ScrollableTreeWindow The *ScrollableTreeWindow* class is the most simple way to embed the tree inside a component. The class extends SWT's ScrolledComposite class and instantiates its own *TreeWindow*, wrapping its interface to enable tree updates.

PainItemController The *PaintItemController* holds all items that should be painted in a list. It enables the *TreeWindow* to display more than a single tree.

methodinvocationplugin.ctrl

The ctrl or controller package contains all controllers as well as the socket server and the *BugMapper* logic of BOA.

Aspecter The Aspecter is responsible to build up the invocation tree and holds the cache array. The class implements the process of caching indexed information and queueing unresolvable references of *CompressedInvocationInfo* objects.

DataController The DataController implements the *SelectionListener* of Eclipse and therefore reacts on user interaction with Eclipse's navigator and resource browser by reading the projects xml or writing it if a new feature was recorded.

Launcher The Launcher implements the *ILaunchDelegate*. The class starts an application with built in feature recording ability. It is also the class that implements the heuristic check whether a class should be excluded due to its potential size after weaving or not.

MappingController The *MappingController* is the heart of the *BugMapper*, calculating the Bug Relevance for a specified bug.

SimpleSerializationServer and SimpleSerializationServerThread The *SimpleSerializationServer* passes the incoming connections over socket port 1337 on to the *SimpleSerializationServerThread*, which holds an *ObjectInputStream* to deserialized the invocation information from the BOA client. To build up the tree, the *SimpleSerializationThread* calls the Aspecter's *addMethodCall* method when a new objects passes the stream.

SpeedyGonzalesCheese *SpeedyGonzales* is the speed up mechanism that leads to the optimized transmission, only transferring data when recording is activated. The *SpeedyGonzalesCheese* is a socket client, that sends the start and stop commands to the BOA client.

methodinvocationplugin.model

This package contains the different models used in BOA. Therefore it is important that the newest library is available as a jar inside the class path of the running weaved application.

CompressedInvocationInfo and ExplicitInvocationInfo These two classes are used to transfer the invocation information from the BOA client to the plug-in. As described in Chapter 4, the *CompressedInvocationInfo* only holds the index of the actual *ExplicitInvocationInfo*, while the last one holds the whole information. When new information is added to the *ExplicitInvocationInfo*, it is important that the *cheapClone* method is correctly extended. Otherwise copied invocation information from the cache array will suffer from information loss.

Feature The Feature is holding a name and a root of the type *MethodInvocationTree* representing a single feature.

FeatureGroup Categories are organized in *FeatureGroups*. A *FeatureGroup* can hold other *FeatureGroups* as sub-categories and *Features*.

MetricInformationHolder The *MetricInformationHolder* simply holds the metric information of a *MethodInvocationItem*.

TreeUtil This class is used for common used tree transformations and other stuff. A developer should write general tree utility methods in here.

methodinvocationplugin.views

The views package contains all user interfaces of BOA.

LauncherUI The LauncherUI contains all the launcher tabs that are available to the run configuration of the *Feature Gathering* technique of BOA. These include

- The JavaMainTab
- The JavaJRETab
- The CommonTab
- The LTWAspectPathTab
- The CustomLTWExcludeTab

When adding new views, it is important that they are correctly set up in the initializeFrom, setDefaults and performApply methods. All tabs have to implement or extend a class implementing *ILaunchConfigurationTab*.

FeatureView The *FeatureView* is the main view of BOA. It is the only one besides the *LauncherUI* that is directly integrated inside Eclipse and therefore registered in the plugin.xml

MetricWindow The MetricWindow is used to show the metric information of a method inside a feature.

methodinvocationplugin.views.components

The views.components package contains all components written for BOA.

BugMapperFrame This is the view of the Bugmapper.

MethodDedicationMatrixUtil This view is used to generate the heatmap using images of such a heatmap representation.

MethodDedicationMetrix This component renders a square per method, relying on the *MethodDedicationMatrixUtil* to render the heatmap color.

Bibliography

[1] Victor Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[2] Robert C.Martin. *Clean Code, A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[3] Thomas A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[4] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 326–337, October 1993.

[5] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[6] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992. Depth.

[7] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[8] Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In *Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 220–234. Springer Verlag, 1990.

[9] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3D. In *Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization)*, September 2006.

[10] Marcel Haerry. Augmenting eclipse with dynamic information. Master's thesis, University of Bern, May 2010.

[11] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of 1st International Workshop on Dynamic Analysis (WODA)*, May 2003.

[12] Dean Jerding, John Stasko, and Thomas Ball. Visualizing message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996.

[13] Michael F. Kleyn and Paul C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, volume 23, pages 191–205. ACM Press, November 1988.

[14] Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, New York NY, 1995. ACM Press.

[15] Alok Mehta and George Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.

[16] Steven P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[17] David Röthlisberger, Orla Greevy, and Adrian Lienhard. Feature-centric environment. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft 2007) (tool demonstration)*, 2007.

[18] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 79–100. ACM Digital Library, 2007.

[19] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.