Feature Driven Browsing[⋆]

David Röthlisberger Orla Greevy Oscar Nierstrasz

Software Composition Group
University of Bern – Switzerland

Abstract. Development environments typically present the software engineer with a structural perspective of an object-oriented system in terms of packages, classes and methods. From this perspective it is difficult to gain an understanding of how source entities participate in a system's features at runtime, especially when using dynamic languages such as Smalltalk. In this paper we evaluate the usefulness of offering an alternative, complementary feature-centric perspective of a software system when performing maintenance activities. We present a feature-centric environment combining interactive visual representations of features with a source code browser displaying only the classes and methods participating in a feature under investigation. To validate the usefulness of our feature-centric view, we conducted a controlled empirical experiment where we measured and compared the performance of subjects when correcting two defects in an unfamiliar software system with a traditional development environment and with our feature-centric environment. We evaluate both quantitative and qualitative data to draw conclusions about the usefulness of a feature-centric perspective to support program comprehension during maintenance activities.

1 Introduction

System comprehension is a prerequisite for software maintenance but it is a time-consuming activity. Studies show that 50-60% of software engineering effort is spent trying to understand source code [1, 2]. Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand runtime behavior purely by inspecting source code [3–5]. In particular, with dynamic languages it is nearly impossible to obtain a complete understanding of a system by just looking at its static source code. The task of understanding a software system is further exacerbated by a best practice in object-oriented programming to scatter behavior in many small methods, often in deep inheritance hierarchies [6].

The problems of understanding object-oriented software are poorly addressed by current development tools, since these tools typically focus only on a structural perspective of a software system by displaying static source artifacts such

[⋆] Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 79-100. DOI 10.1145/1352678.1352684.

as packages, classes and methods. This is also true for modern Smalltalk dialects and development environments such as Squeak [7] or Cincom VisualWorks [8].

A system may be viewed as a set of features. Each feature represents a well-understood abstraction of a system's problem domain. Typically maintenance requests are expressed in terms of features [9]. Thus, understanding how features are implemented is a prerequisite for system maintenance. A feature denotes a unit of behavior of a system. It exists at runtime as a collaboration of objects exchanging messages to achieve a specific goal. However, as it is not explicitly represented in the source code, it is not easy to identify and manipulate. In this paper, we adopt the definition of a feature as being a unit of observable behavior of a system [10].

As traditional development environments offer the software engineer a purely structural perspective of object-oriented software, they make no provision for the representation of behavioral entities such as features. To tackle this shortcoming, we propose to support the task of understanding during maintenance by augmenting a static source code perspective with a feature perspective of a software system. We present a novel feature-centric environment providing support for visual representation, interactive exploration, navigation and maintenance of a system's features. To motivate our work, we address the following questions:

- How useful is a feature-centric development environment for understanding and maintaining software?
- How do we quantitatively measure the usefulness of a feature-centric development environment?
- How do software engineers subjectively rate the usefulness of a feature-centric perspective of a system to perform their maintenance tasks?

The fundamental question we seek to answer is if software engineers can indeed better understand and maintain a software system by exploiting a feature-centric perspective in a dedicated feature-centric development environment. We want to determine if a feature-centric perspective is superior to a structural perspective to support program comprehension. To address this question, we implemented a feature-centric development environment in Squeak [7], a dialect of Smalltalk. Our feature-centric development environment acts as a proof of concept for the technical feasibility of our approach and as a tool which we can actually validate with real software engineers.

The key contributions of this paper are: (1) we present our feature-centric development environment, and (2) we provide empirical evidence to show its usefulness to support comprehension and maintenance activities as compared with the structural views provided by a traditional development environment.

Paper structure. In the next section we expand on the problem of feature comprehension and provide a motivating example. Based on this, we formulate our hypotheses of the usefulness of a feature-centric perspective for performing maintenance tasks. In Section 3 we introduce our *feature browser* proof of concept tool, allowing a developer to work in a feature-centric development environment. We validate the usefulness of our feature-centric development environment by conducting an empirical study in Section 4. We present the results and evidence of our study in Section Section 5. We report on related work in Section 6 and finally we conclude in Section 7.

2 Problem of Feature Maintenance

It is a generally accepted *best practice* of object-oriented programming that functionalities or *features* are implemented as a number of small methods [4]. This, in addition to the added complexity of inheritance and polymorphism in object-oriented software, means that a software engineer often needs to browse a long chain of small methods to understand how a feature is implemented. In Figure 1 we illustrate this with an example from Pier [11], the system we chose as a basis for our experimentation. Pier is a web content management system encompassing a Wiki application [11] implemented in Squeak Smalltalk [7]. Figure 1 shows a small part of the class hierarchy of Pier and an excerpt of a call tree, generated by exercising the *copy page* feature. The call tree reveals that the feature implementation crosscuts the structural boundaries of the Pier system.

A software engineer, faced with the task of maintaining the *copy page* feature, first needs to locate the relevant classes and methods, and then browses back and forth in the call chain to establish a mental map of the relevant parts of the code and to gain an understanding of how the feature is implemented. This is a cumbersome and time-consuming activity and often the software engineer loses time and focus browsing irrelevant code.

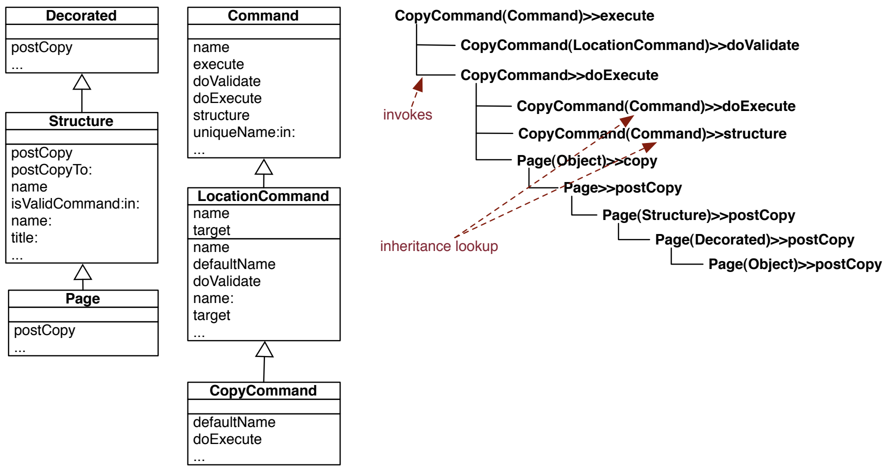


Fig. 1. The relevant Pier Class Hierarchies for the *copy page* Feature and its Call Graph

2.1 Making Features Explicit in the Development Environment

From the perspective of a software engineer, a feature consists of a set of all methods executed while performing a certain task or activity in a software system. The relationships between the methods of a feature are dynamic in nature and thus are not explicit in the structural representation of the software [12]. For our purposes, we represent features (*i.e.,* dynamic units of behavior) in terms of their particpating methods. We aim to support the software engineer when maintaining or fixing a defect in a feature. Thus we need to capture and represent features as explicit entities. The behavior of a feature may be captured by triggering an activity from the user interface. Alternatively, as described in the work of Licata et. al. [13], test cases are typically aligned with features. For our experimentation we opted to use test cases to trigger features. Furthermore, by using test cases we can better control the volume of dynamic information captured to represent each feature.

Our premise is that by explicitly representing features in the development environment, we support maintenance activities by providing the software engineer with an explicit map between features and source entities that implement the feature. By focusing the attention of the maintainer on only relevant source entities of a given feature or set of features, we improve the understanding and the ease with which she can carry out maintenance tasks.

We state our hypotheses as:

- *A feature-centric development environment decreases the time a software engineer has to spend to maintain a software system (e.g., to correct a bug) compared to a traditional development environment which provides only a structural perspective of the code*
- *A feature-centric development environment improves and enriches the understanding of how the features of a software system are implemented*

We refine our hypotheses in Section 4, when we describe the details of our empirical study. Our qualitative and quantitative evaluation of the findings of our experimentation reveal that these hypotheses do indeed hold.

3 Proposing a Solution: A Feature-centric Environment

The foundation of our approach is a *feature browser* tool which we embed in the software engineer's integrated development environment (IDE). The purpose of the *feature browser* is to augment an IDE with a feature perspective of a software. We implemented our prototype *feature browser* in Squeak Smalltalk [7]. The *feature browser* complements the traditional structural and purely textual representation of source code in a browser by presenting the developer with interactive, navigable visualizations of features in three distinct but complementary views. These views are enriched with metrics to provide the software engineer with additional information about the relevancy of source artifacts (*i.e.,* classes and methods) to features.

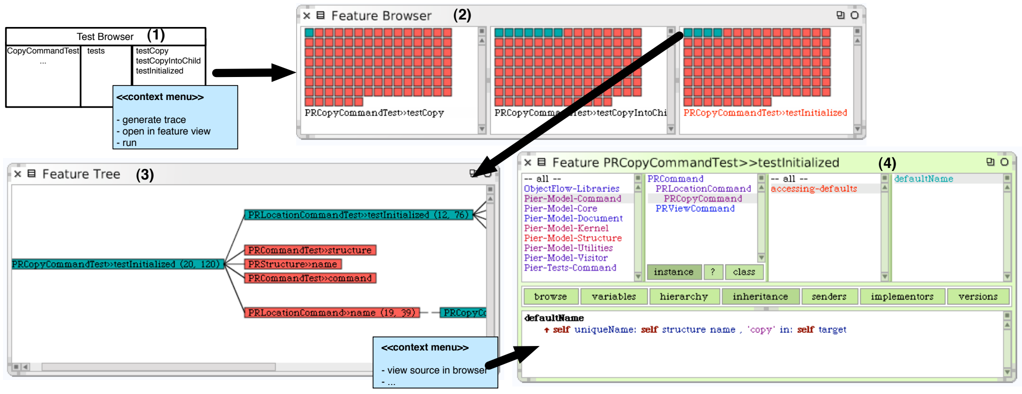
Initially, we introduce the key elements of our feature-centric environment. Subsequently, we describe how feature-centric environment promotes software engineer's comprehension of scattered code in object-oriented programming while performing maintenance tasks on a system's features.

3.1 Feature Affinity in a Nutshell.

In previous work [14], we defined a *Feature Affinity* measure to assign a relevancy scale to methods in the context of a set of features. Feature Affinity defines an ordinal scale corresponding to increasing levels of participation of a source artifact (*e.g.,* a method) in the set of features that have been exercised. For our feature-centric environment we consider four Feature Affinity values: (1) a *singleFeature* method participates in only one feature, (2) a *lowGroupFeature* method participates in less than 50% of the features, (3) a *highGroupFeature* method participates in 50% or more of the features and (4) an *infrastructuralFeature* method participates in all of the features.

We exploit the semantics of *Feature Affinity* to guide and support the software engineer during the navigation and understanding of one or many features. We assign to the visual representation of a method a color that represents its *Feature Affinity* value. Our choice of colors corresponds to a heat map (*i.e.,* a cyan method implies *singleFeature* and red implies *infrastructuralFeature*, *i.e.,* used by all the features we are currently investigating).

3.2 Elements of the feature-centric environment



```
defaultName
    ↑ self uniqueName: self structure name , 'copy' in: self target
```

Fig. 2. The Elements of our Feature Browser Environment

The feature-centric environment contributes three different visualizations for one and the same feature: (2) the compact feature overview, (3) the feature tree view and (4) the feature artifact browser. (1) is the test runner which is not directly part of the feature-centric environment but a separated tool.

Compact Feature Overview

The Compact Feature Overview presents a visualization of two or more features represented in a compacted form. The Compact Feature view represents a feature as a collection of all methods used in the feature as a result of capturing its execution trace. Each method is displayed as a small colored box; the color represents the *Feature Affinity* value. The methods are sorted according to their *Feature Affinity* value. The software engineer decides how many features she wants to visualize at the same time (see Figure 2 (2)). Clicking on a method box in the Compact Feature View opens the Feature Tree View, which depicts a call tree of the execution trace. This visualization reveals the method names and order of execution. All occurrences of the method selected in the Compact Feature View are highlighted in the call tree.

Feature Tree

This view presents the method call tree, captured as a result of exercising one feature (see Figure 2 (3)). The first method executed for a feature (*e.g.,* the "main" method) forms the root of this tree. Methods invoked in this root node form the first level of the tree, hence the nodes represent methods and the edges are message sends from a sender to a receiver. As with the Compact Feature Overview, the nodes of the tree are colored according to their *Feature Affinity* value.

The key challenge of dynamic analysis is how to deal with the large amount of data. For our experimentation we chose Pier [11], a web-based content management system implemented in Smalltalk. We obtained large traces of more than 15'000 methods. We discovered that it is nearly impossible to visualize that amount of data without losing the overview and focus, but still conveying useful information. To overcome this we applied two techniques: First, we compressed the execution traces and the corresponding visual representation as a feature tree as much as possible without loss of information about order of execution of method sends. Second, we opted to execute test cases of a software system rather than interactively trigger the features directly from the user interface. For instance, instead of looking at the entire *copy page* feature initiated by a user action in the user interface, we analyze the *copy page* feature by looking at the test cases that were implemented to test this feature. As stated in the work of Licata [13], features are often encoded in dedicated unit test cases or in functional tests encoded within several unit test cases. In the case of a software system that includes a comprehensive test suite, it is appropriate to interpret the execution traces of such test cases as feature execution traces.

Furthermore, we tackle the problem of large execution traces by compressing feature trace with two different algorithms: First we remove common subexpressions in the tree and subsequently we remove sequences of recurring method calls as a result of loops.

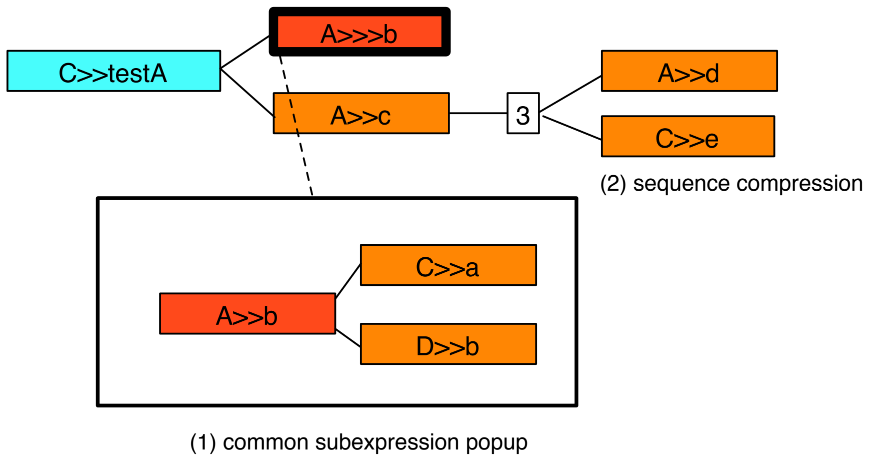


Fig. 3. The Common Subexpression and Sequence Compression of the Feature Tree

Common Subexpression Removal

A subexpression in a tree is a branch which occurs more than once. If, for example, a pattern "method c invokes methods a and b" occurs several times in a call tree, we identify this pattern as a common subexpression. Our analysis reveals that the execution traces of features typically contain many common subexpressions. By compacting the representation of these subexpressions, we reduce the tree by up to 30% on average. Our visualization still includes an expandable root node of a common subexpression branch in the tree, the subexpression can be opened in a pop-up window by the software engineer. Figure 3 (1) shows a schematic representation of how we display common subexpressions in our feature tree view.

To perform the removal of common subexpressions, we applied the algorithm presented in [15].

Sequence Removal

Often a feature trace contains several sequences, *e.g.,* methods invoked in a loop. It is straightforward to compress these nodes included in a loop by only presenting them once. Furthermore, we indicate how often the statements of a loop are repeated. If, for example, the methods d and e are executed three times in a loop, we add an *artificial numeric* node labeled with a '3' to the tree and link the nodes for d and e. to this node (as shown in Figure 3

(2)). To detect and compact sequences, we implemented a variation of the algorithm presented in [16].

Despite having applied these techniques, the feature tree is still complete and is easily read and interpreted by the software engineer. No method calls occurring in the feature are omitted.

Initially a feature tree is displayed collapsed to the first two levels. Every node can be expanded and collapsed again. In this way, the software engineer can conveniently navigate even large feature trees.

When a user selects a method node in the compact feature overview, all the occurrences of this method are highlighted in the feature tree. Also the tree is automatically expanded to the first occurrence of the selected method and the feature tree window is centered on that node. The user can navigate through all occurrences of the desired method by repeatedly clicking on the corresponding node in the compact feature overview. By opening a method node in the feature tree the engineer is able to follow the complete chain of method calls from the root node to the current opened method node. No intermediate calls of methods belonging to the software system under study are omitted.

Every node of the tree provides a button to query the source code of the corresponding method in the feature artifact browser.

Feature Artifact Browser

The source artifacts of an individual feature are presented as text in the *feature artifact browser* (see Figure 2 (4)). It exclusively displays the classes and methods actually used in the feature. This makes it much easier for the user to focus on a single feature of the software. Our *feature artifact browser* is an adapted version of a standard class browser available in the Squeak environment. It containing packages, classes, method categories and methods in four panes on the top, while the lower pane contains the source code of the selected entity. This version of the class browser not only presents static source artifacts, but also the feature affinity metric values by coloring the names of classes and methods accordingly.

The three distinct visualizations provided by the feature browser are tightly interconnected so that a software engineer does not lose the overview when performing a maintenance task. For instance, the user selects a method in the compact feature overview, the tree opens with all occurrences of the selected method. From the tree perspective, the software engineer can choose to view a method as source code in the feature artifact browser by double-clicking on a node that represents the method of interest in the feature tree.

To initiate a maintenance session with the *feature browser*, the software engineer selects test cases that exercise features relevant to her specific maintenance task. To ease the collection of features traces, we extended the standard class browser of Smalltalk (called OmniBrowser) to seamlessly execute instrumented test cases and capture feature traces as shown in Figure 2 (1). Thus, once she has executed the instrumented test cases, the software engineer launches a feature browser with an initial number of features.

3.3 Solving Maintenance Tasks with the Feature Environment

The feature-centric environment supports program comprehension and feature understanding for maintenance tasks in three ways:

Firstly, by comparing several features with each other in the compact feature overview, software engineers gain knowledge about how different selected features are related in terms of the methods they have in common. Since the features included in this compact feature overview can be arbitrarily selected, a developer can compare any features with each other. However, when performing a specific maintenance task, it makes sense to select related test cases to *e.g.,* compare failing tests with similar, non-failing tests to determine which parts of a software system may be most likely responsible for a defect. If, for example, only one test method of a test class exercising the *copy command* feature of a software system is failing, then we compare this test method to all test methods exercising the *copy command* feature. We assume that by looking first at the methods that are only used in the single failing test method, we are more likely to be faster at discovering the defect, as the chances are high that one of the *singleFeature* methods (*i.e.,* methods unique to one compact feature view) are responsible for that defect. The aim of the compact feature overview is to support the quick identification and rejection of candidate methods that may contain a defect.

Secondly, the feature tree provides an insight into the dynamic structure of a feature, an orthogonal dimension compared to the static structure visible in the source code. The nodes in the tree are also colored according to the Feature Affinity metric which guides the software engineer to identify faulty methods. In the example described above where a single feature is failing the most likely candidate methods responsible for the defect are colored in cyan in the feature tree. Since this tree is complete, *i.e.,* it contains all method calls for that specific feature, chances are high that the engineer discovers the source of the defect in one of these single methods that are easy to locate in the feature tree due to their coloring.

The software engineer can also navigate and browse the tree to obtain a deeper understanding of the implementation of the feature and the relationships between the different methods used in the feature. For every method of a feature, the software engineer can easily navigate to all occurrences of this method in the feature tree to find out how and in which context the given method is used. This helps one to discover the location of a defect and the reason why it occurs. The feature tree transforms and improves the understanding of the dynamic structure of a feature and reveals where and how methods are used. For every node in the feature tree, the developer can view the source code of that method in the feature artifact browser.

Thirdly, the feature artifact browser helps the developer to focus only on entities effectively used in a feature. The number of methods that might be responsible for bugs is thus reduced to a small subset of all existing methods in a class or a package. Since only packages, classes and methods are presented to the developer in the feature artifact browser, it is much easier for her to find information relevant for a defect or another maintenance task, *i.e.,* classes or methods. Hence the feature artifact browser helps one to focus on relevant source artifacts and to not lose track and context.

4 Validation

To obtain a measure of the usefulness of our feature-centric environment and its concepts in practice, we conducted an empirical experiment with subjects using and working with the feature-centric environment. The goals of the experiment were to gain insight into the strengths and shortcomings of our current implementation of the feature-centric environment, to obtain user feedback about possible improvements and enhancements, and to assess the practical potential of the feature-centric environment. Our primary goal was to gather quantitative data that would indicate how beneficial was the effect of using the feature-centric environment as compared with the standard structural and textual representations of a traditional development environment. We introduce and describe the experiment in this section, formulate the hypotheses we address and describe precisely the study design. Finally, we present the results we obtained from the experiment.

4.1 Introducing the Experiment

To validate our feature-centric environment we asked twelve subjects (computer science graduate students) to perform two equally complex maintenance tasks in a software system, one task performed in the feature-centric environment and the other in the standard environment of Squeak Smalltalk (*i.e.,* using OmniBrowser). As a maintenance task, we assigned the subjects the correction of a defect in the software system. The presence of the defect is revealed by the fact that some of the feature tests are failing.

In this experiment we seek to validate three hypotheses concerning our feature-centric environment. If the result of the experiment reveals that the hypotheses hold, we then have successfully obtained clear evidence that our feature-centric environment supports a developer to perform maintenance and that the feature affinity metric we applied is of value in practice.

4.2 Hypotheses

We propose the three null hypotheses listed in Table 1. The goal of the experiment is to refute these null hypotheses in favor of alternative hypotheses which would indicate that a feature-centric environment helps the software engineer to discover and correct a defect and hence improves program comprehension.

4.3 Study design

Study setup.

$H0_1$	The time to discover the location of the defect is equal when using the standard browser and our feature-centric environment. (formally: $\mu_{D,FB} = \mu_{D,OB}$, where $\mu_{D,FB}$ is the average discovery time using the feature-centric environment and $\mu_{D,OB}$ the average discovery time using OmniBrowser)
$H0_2$	The time to correct the defect is equal when using the standard browser and our feature-centric environment. (formally: $\mu_{C,FB} = \mu_{C,OB}$)
$H0_3$	The feature-centric environment has no effect on the software engineer's program comprehension. (formally: average effect $\mu_{E,FB} = 0$)

Table 1. Formulation of the null hypotheses

During the experiment, subjects were asked to correct two bugs in a complex web-based content management framework written in Smalltalk. Our software system, Pier [11], consists of 219 classes of 2341 methods with a total of 19261 lines of code. The two defects were approximatively equally complex to discover and correct. For both bugs we slightly changed one method in the Pier system. As a result of our change, some (feature) tests failed. We presented the subjects with these failing tests as a starting point for their search for the defect. In Pier a unit test class is dedicated to a certain feature (*e.g.,* copying a Wiki page) and the different methods of a test class are different instantiations of that feature (*e.g.,* different parameters with which the feature is exercised). This is in line with the argumentation presented in the work of Licata [13] saying that features are often encoded in unit test cases.

In our experiment, we introduced two different defects in the *copy page* feature. This feature is tested by a dedicated test class with five test methods. The two defects produce failures in different test methods of the *copy page* test class. For the experiment we select all the five test methods exercising the *copy page feature* and show them in row in the Compact Feature View our feature-centric environment. As these five exercised test methods are variants of the same feature they are clearly related to each other, which means that if one test method reveals a failure but the others don't, it is most likely to spot that failure by just looking at the different methods the failing test is executing.

We conducted the experiment with twelve graduate computer science students as subjects with varying degrees of experience with the Smalltalk programming language and the Squeak development environment. All subjects had between one and five years of experience with the language, but only between zero and four years of experience with the Squeak development environment. None of the subjects was familiar with the design and implementation of the Pier application in detail.

Before starting the experiment, we organized a workshop to introduce the concepts and paradigms of our feature-centric environment. Every subject could experiment with our feature browser for half an hour before commencing our experiment consisting of the task of defect location and correction. Furthermore, we briefly introduced the subjects to the design and the basic concepts of Pier by presenting an UML diagram of the important entities of the application. The experiment was conducted in a laboratory environment, as opposed to the subjects' normal working environment. While performing the experiment, we observed the subjects. Afterwards we asked them to respond to a questionnaire to gather qualitative information about the feature-centric environment. The questionnaire contains several questions about the usefulness of the feature-centric environment to understand the program and to perform the requested maintenance task. For every question, the subjects could choose a rating from -3 to 3, where -3 represents a hinderance to program comprehension, 0 no effect and 3 very useful. In addition, the subjects could provide qualitative feedback, *e.g.,* what shortcomings of the environment suggested improvements. The results of these open suggestions, as well as the observations of the experimenters form the qualitative part of our study.

Every subject had to fix both defects, one using our feature-centric environment and the other one using the standard class browser, *i.e.,* OmniBrowser. Both the debugger as well as the unit-test runner were available for use to complete the task. We prohibited use of every other tool during the experiment. From subject to subject we varied the order in which they fixed the defects as well as the order in which they used the different browsers. Hence there are four possible combinations to conduct an experiment with a subject and each of these four combinations were exercised three times. A concrete combination was randomly chosen by the experimenters for any subject.

Dependent variables.
We recorded two dependent variables: (i) the time to discover the location (*i.e.,* the method) where the defect was introduced, and (ii) the time to actually correct the defect completely. We considered the goal as being achieved when all 872 unit tests of Pier ran successfully. The bugs had to be fixed in the right method, thus, they were carefully chosen so that they could only be corrected in this method.

4.4 Study Result

Initially we report on the quantitative data we obtained by recording the time the subjects spent to find and correct the defects using the different browsers. Then we evaluate the results from the questionnaire and finally we present qualitative feedback reported by the subjects.

Time Evaluation.
Figure 4 compares the average time spent to correct the bugs, the average times were aggregated independent on the used browser or on the order in which the different defect were addressed by the subject. The figure clearly shows that the two bugs were approximately equally complex, thus allowing us to compare the time the different subjects spent in different environments to correct the two defects. We initially selected these two defects after having assessed their complexity in a pre-test with two subjects working in the standard Squeak browser.

These two subjects needed approximately the same time to correct both defects and subjectively considered the two bugs as equally complex and difficult to correct.

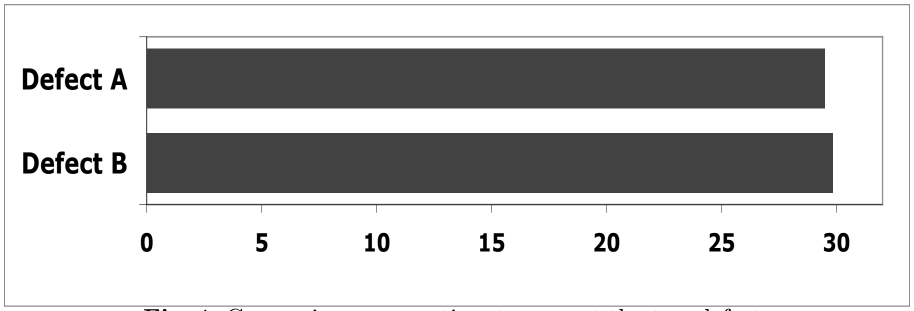


Fig. 4. Comparing average time to correct the two defects

Figure 5 compares the total time the subjects spent to discover the location of the defect once using the feature-centric environment and once using the OmniBrowser. The average benefit of using the feature-centric environment is 56 percent which are in total 56 minutes less than using OmniBrowser. During the experiment we considered that the correct location of the defect has been discovered when the subject announces that the defect has to be in the specific faulty method. The subjects were asked to name the faulty method as soon as they believed to have found it. The situation is similar when considering the time spent to fully correct the defects (see also Figure 5), which is the time to discover the defect plus the amount of time to edit and correct the faulty method. Here we get a relative improvement of 33 percent and an absolute of 100 minutes saved when using the feature-centric environment instead of OmniBrowser. Figure 6 presents boxplots showing the distribution of the discovery and correction time the different subjects spent in different browsers. The different defects are not identifiable in these boxplots, only the different browsers. To measure the whole correcting time we considered the bug as being corrected as soon as all the tests of Pier run successfully.

Evaluation of the Questionnaire.

In our questionnaire we mainly asked the subjects how they rate the effect of the different aspects of the feature-centric environment on program comprehension. We asked about the overall effect on program comprehension, the effect of the feature overview, of the feature tree, feature class browser and of the feature affinity metric. Furthermore, we asked how well certain parts of the feature-centric environment were understood by the subjects and how well they could interact with the different parts. In Table 7 we present the details of our questionnaire and the average results we got from the subjects. They could choose

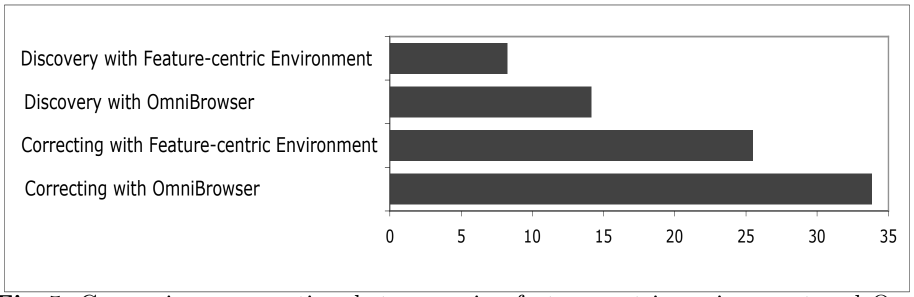


Fig. 5. Comparing average time between using feature-centric environment and OmniBrowser to discover resp. correct a defect

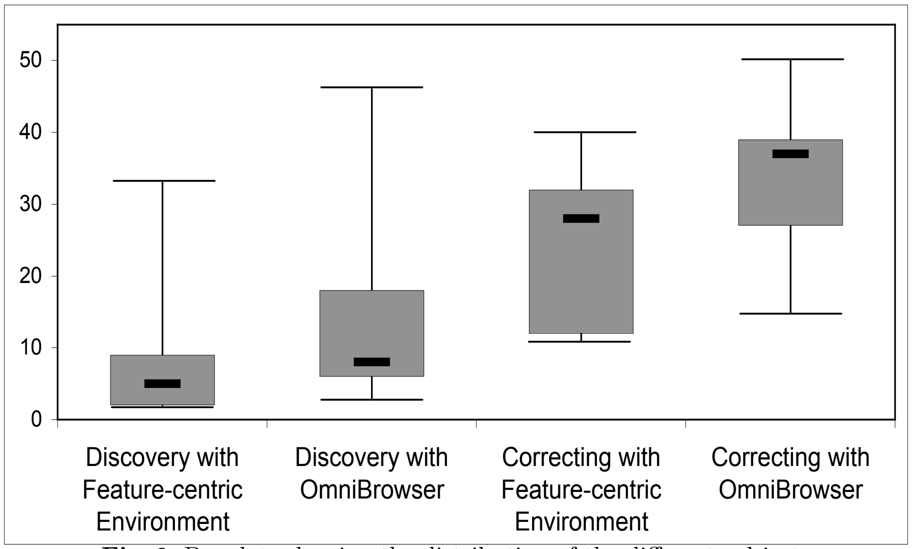


Fig. 6. Boxplots showing the distribution of the different subjects

	Question	Result
1.	General effect on Program Comprehension	1.16
2.	Effect of Compact Feature Overview	1.58
3.	Effect of Feature Tree Browser	1.33
4.	Effect of Feature Class Browser	1.50
5.	Effect of Feature Affinity Metric	1.42
6.	Understanding the subexpression compression in Feature Tree	1.58
7.	Understanding the sequence compression in Feature Tree	1.75
8.	Understanding the navigation in Feature Tree	2.00
9.	Interaction with Compact Feature View	1.50
10.	Interaction between Feature Tree and Feature Class Browser	1.58

Fig. 7. Questionnaire.

between a rating from -3 to 3, so an average rating of 1.16 for *e.g.,* "General effect on Program Comprehension" reveals a positive, although not a very strong effect. As an example, we depict in Figure 8 the results for the question about the effect of the compact feature overview on program comprehension. The ratings were in average 1.58 which denotes that the subjects considered the effect in average as "good".

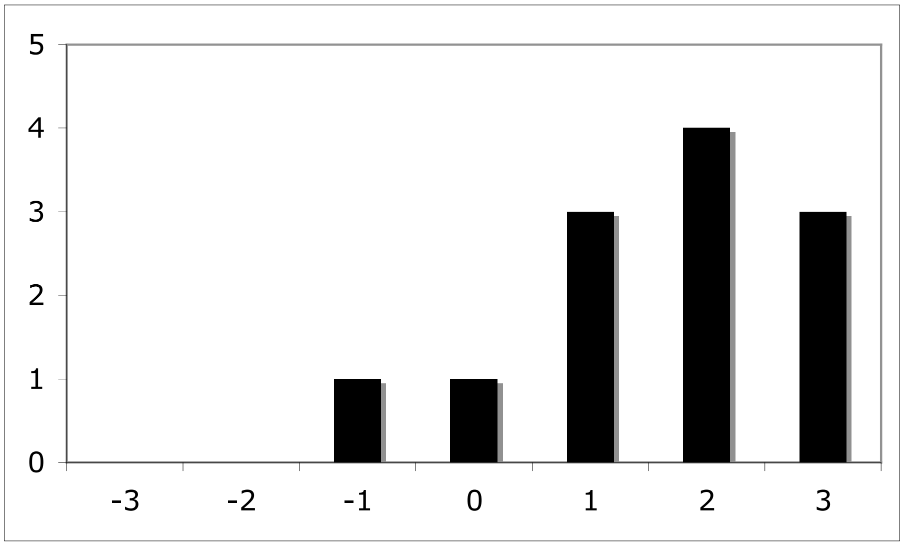


Fig. 8. Comparing the average results for the effect of compact feature overview on program comprehension

Statistical Conclusion.

To test the first two hypotheses formulated in Table 1 we apply the one-sided independent t-test [17] with an α value of 10% and 22 degrees of freedom. One requirement for applying the t-test is equality of the variance of the two samples. For the two discovery time samples we determine a variance of 92 and 112, respectively, for the correcting time samples the variances are more closes to each other, 102 and 110, respectively. For the correcting time the variance requirement is fulfilled, for the discovery time we are careful and assume that this requirement is not fulfilled. Another requirement for applying the t-test is a normal distribution of the underlying data which we justify with the Kolmogorow-Smirnow-Test. With an α value of 5% the result of this test allows us to assume normal distribution for the correcting time samples, with an α value of 10% we can also assume normal distribution for the discovery time samples.

These preliminary tests allows us to use the t-test at least for the correcting time. We also use it for the discovery time, but we are skeptical about the result in that case. For the discovery time we calculated a t value of 1.32. The t distribution tells us the probability that t > 1.32 is exactly 10% which means that we can just barely reject the null hypothesis $H0_1$. Because the requirements for the t-test are not properly fulfilled and because an α value of 10% is probably to low to justify a rejection of the null hypothesis we cannot prove the positive effect of the feature-centric environment on the discovery time of a defect.

For the time to correct a defect we obtain a t value of 1.86 which is even greater than the t value of the 95% confidence interval (t = 1.717). This means that we can reject $H0_2$ even with an α value of 5% and accept the alternative hypothesis $H1_2$ saying that the feature-centric environment speeds up the time to correct a defect ($\mu_{C,FB} < \mu_{C,OB}$).

To test the third hypothesis we use the results of the questionnaire and apply the one-sided Wilcoxon signed-rank test [18]. We cannot assume normal distribution for the underlying data in this case since the ratings were almost all positive. We only apply the test to the answers for the general effect of the feature-centric environment. We calculate a W value of 26 which is exactly equal to the S value we find in the tabular denoting the 95% confidence interval. This means that we can reject $H0_3$ with an α value of 5% and hence accept the alternative hypothesis which says that the feature-centric environment has a positive effect on program comprehension.

5 Discussion

In this section we report on the main threats to validity of our experiment and on the conclusions we can draw from this study. Furthermore, the subjects provided us with a wide range of constructive criticism and suggestions for improvements to the feature-centric environment, which we also outline in this section.

5.1 Threats to Validity

We distinguish between external, internal and construct validity [19].

– *External validity* depends on the subjects and the software system in which subjects are asked to correct defects during the experiment. In our case the software system, Pier, is a complex real-world application comparable to other industrial object-oriented systems. The subjects, however, are students and research assistants who may not be directly comparable to programmers in industry. Furthermore, the experiment was conducted in a laboratory environment which biases the performance of the subjects. Hence the results are not directly applicable to settings in practice, although we consider the aforementioned influences as small.

– *Internal validity* is jeopardized by the fact that not all subjects had the same amount of experience with the programming language and especially with the Squeak development environment. While everybody was quite familiar programming in Smalltalk (at least one year of experience), the particular environment was new to some of the subjects. But because these problems were present for both defects and browsers, we believe that they do not bias the results tremendously. Furthermore, we changed the order of the bugs and browsers from subject to subject, hence the results were only slightly biased by the fact that subjects had more insight into the development environment and also into the software system when fixing the second bug than they had for the first bug. However, the different amount of experience of the subjects is nonetheless a shortcoming of this study.
  Another important issue is that subjects could also use other tools than just the two different browsers, *e.g.,* the debugger. As it is usually necessary to use a debugger in a dynamic language to find a bug, we did not prohibit its usage. Some subjects performed the task predominantly by using the debugger to find the bug whereas they made the necessary corrections often in the provided browser. These other available tools clearly bias the result of our experiment to a degree which is hard to estimate.
  Yet another important issue is that the subjects did not know or use the feature-centric environment previously. It is a complete new environment with a very different approach to look at a software system and its features. The other environment (*i.e.,* the OmniBrowser) was well-known by all subjects, since the paradigm applied in this browser is the standard way of browsing source code in most Smalltalk dialects (and indeed for other object-oriented languages).

– *Construct validity.* Our measure, the time to find and correct a defect, is adequate to assess the contribution of the feature-centric environment to maintenance performance. However, this time is certainly biased by many other factors than the browsers in use, such as the experience of the developer, her motivation, the use of other tools, etc. To assess the effect of the feature-centric environment on program comprehension we used our questionnaire to obtain feedback on how the subjects personally judge the effects on program comprehension. These answers are certainly subjective and may hence not be representative. Thus the applied measures are not a perfect assessment of the effects of the feature-centric environment on maintenance performance and on program comprehension, although at least the former is still relatively well assessed with the applied measures.

5.2 Study Conclusion

Two main issues of our empirical study are: (1) the subjects participating the experiment do not have the same experience with the programming language and the development environment, and (2) they were unfamiliar with our feature-centric environment as they had never used it before. On the other hand, they are familiar with the standard development environment. Another important issue is that due to the limited number of subjects participating in the experiment, it is difficult to draw statistically firm conclusions. This study nonetheless gives us worthwhile insights into how software engineers use and judge our feature-centric environment and defines a framework for further studies of this kind. The results we obtained motivate us to proceed with our work on this environment and the subject feedback provides us also several new ideas and approaches that we intend to pursue. We conclude that performing this study was crucial to validate and improve our work.

5.3 Elements of the Feature-Centric Environment

In this section we discuss some of the interesting ideas and suggestions we obtained as feedback from the subjects who participated in the experiment. Furthermore, we also discuss some of the issues of feature analysis inherent in our feature-centric environment:

– *Bidirectional interactions*. Providing the capability to navigate the tree by clicking and selecting the textual representations of the methods in the feature artifact browser and being able to click on nodes in the tree to select the same methods in the compact feature view is a useful improvement to the feature-centric environment. This helps one to navigate and understand more quickly the structure and implementation of a feature.
– *Bind tree to debugger*. Using a debugger is not an easy task since we only see a slice of a program in the debugger but not the overall structure. If we could use the feature tree to step through a running program to debug it, we would get a better overview and understanding of the overall structure of a feature. Hence a promising extension of the feature-centric environment would be to add debugging facilities to the feature tree, such as stepping through a program, inspecting variables and changing methods on the fly.
– *Delta debugging*. Using the feature-centric environment to discover and correct a defect in a feature is a frequent task which we can ease by analyzing test cases representing features. Careful analysis of test cases with delta debugging approaches [20] allows us to rate methods used in a feature according to their probability being responsible for a defect. If we present the methods in the compact feature view sorted by their probability to contain erroneous

code a developer can very quickly focus on the right methods to correct a defect.

- *Performance analysis* By enriching the feature-centric environment, in particular the feature tree, with more dynamic information such as execution time or memory consumption, the tool will be well suited for performance analysis. The feature tree can easily reveal which branch consumes the most resources or which specific method call takes the most time to execute. Another useful enhancement is a mechanism to compare different executions of a feature (*e.g.,* with different parameters) with each other to emphasize differences in execution time or consumption of resources.
- *Scalability Issues.* Dynamic analysis approaches are required to manipulate large amounts of data. We address this issue by using test cases to trigger the behavior of features. Furthermore, we present the user with both a compressed feature view as well as the entire call tree of a feature. To reduce the call tree, we applied compression algorithms. Our feature representations could also be reduced by applying selective instrumentation and filtering techniques [16].
- *Coverage.* By using test cases to represent features we do not obtain full coverage of all possible execution paths of a feature. Other feature identification approaches are also subject to this limitation [10, 21]. We argue that although full coverage is desirable, it is not essential to support a feature-centric approach to software maintenance.

6 Related Work

The problems of understanding object-oriented software when browsing and maintaining a system are well-documented [3–5, 22]. Wilde and Huitt [5] described several problem areas related to object-oriented code and they suggested tracing dependencies was vital for effective software maintenance. To fully understand a method, its context of use must be found by tracing back the chain of messages that reach it, together with the various chains of methods invoked by its body [5]. Nielson and Richards [23] investigated how difficult it was for experienced software engineers to learn and use the Smalltalk programming language. They found the distributed nature of the code caused problems when attempting to understand a system. Chen et. al. also highlighted class hierarchies and messages as posing difficulties when trying to maintain object-oriented code [24]. It is exactly such problems we aim to address with our feature-centric environment.

Some empirical analysis work has been carried out to obtain evidence to support their claim that understanding object-oriented software to perform maintenance tasks is difficult. In the work Brade et. al. [25], they presented a tool (Whorf) to provide explicit support for delocalized plans (conceptually related code that is not localized contiguously in a program). Whorf provided hypertext links between views of the software to highlight interactions between physically disparate components. They performed an experiment with software engineers to measure how quickly it took them to identify relevant code to perform an enhancement to a software system, once with paper documentation and once with Whorf . Their results show that using Whorf improved efficiency when performing a maintenance task. The focus of this work was to support documentation strategies for object-oriented programs.

Dunsmore et. al. [4] highlighted the shortcomings of traditional code inspection techniques when faced with the problem of delocalization in object-oriented software. They present the results of an empirical investigation to measure how long it took software engineers to locate defects in a software system.

In contrast to the work of Dunsmore et. al. and Brade et. al. [4,25], our experiment focuses on measuring the usefulness of our feature-centric environment. With our approach, we provide the software engineer integrated development environment support to locate the parts of the system that are relevant to a feature.

Other related research is that of runtime information visualization to understand object-oriented software [12, 26–28]. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [29], Jinsight and its ancestors [30], and Graphtrace [27]. De Pauw et. al. present two visualization techniques. In their tool Jinsight, they focused on interaction diagrams [30]. Thus all interactions between objects are visualized.

Reiss [28] developed *Jive* to visualize the runtime activity of Java programs. The focus of this tool was to visually represent runtime activity in real time. The goal of this work is to support software development activities such as debugging and performance optimizations.

In contrast to the above approaches, our focus was to directly incorporate interactive and navigable visualizations of the dynamic behavior of features into the development environment. In this way, we emphasize the importance of providing the software engineer with direct access to the information during a maintenance session.

7 Conclusion

In this paper we presented our feature-centric environment which allows us (1) to visually compare several features of a software system, (2) to visually analyze the dynamic structure of a single feature in detail and, (3) to navigate, browse and modify the source artifacts of a single feature in a feature artifact browser focusing on the entities actually used in that feature. All these visualizations are enriched with the feature affinity metric to highlight parts of a feature relevant to a specific maintenance task.

The views on features are fully interactive and interconnected to ease and enhance their usage in maintenance activities. We validated our feature-centric environment by carrying out an empirical study with twelve graduate computer science students. The results of our experiment are promising because they clearly reveal that the feature-centric environment has a positive effect on program comprehension and in particular on the efficiency in discovering the exact locations of software defects and in correcting them efficiently. We recognize that as our experiment had only a low number of participating subjects, it is difficult to generalize the results. However, feedback of the users in addition to the quantitative results of our analysis are encouraging. This motivates us to continue investigation and development of a feature-centric environment that represents features as first-class entities for the software engineer. In particular, we aim to focus on ideas mentioned in Section 5.2, for example providing bidirectional interaction to the feature-centric environment or to add debugging facilities to the feature tree.

Acknowledgments

We thank all the participants of our experiment.

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

1. V. Basili, Evolving and packaging reading technologies, Journal Systems and Software 38 (1) (1997) 3–12.
2. T. A. Corbi, Program understanding: Challenge for the 1990's, IBM Systems Journal 28 (2) (1989) 294–306.
3. S. Demeyer, S. Ducasse, K. Mens, A. Trifu, R. Vasa, Report of the ECOOP'03 workshop on object-oriented reengineering (2003).
4. A. Dunsmore, M. Roper, M. Wood, Object-oriented inspection in the face of delocalisation, in: Proceedings of ICSE '00 (22nd International Conference on Software Engineering), ACM Press, 2000, pp. 467–476.
5. N. Wilde, R. Huitt, Maintenance support for object-oriented programs, IEEE Transactions on Software Engineering SE-18 (12) (1992) 1038–1044.
6. J. Nielsen, J. T. Richards, The Experience of Learning and Using Smalltalk, IEEE Software 6 (3) (1989) 73–77.
7. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press, 1997, pp. 318–326.
8. Cincom Smalltalk, http://www.cincom.com/scripts/smalltalk.dll/ (Sep. 2003).
9. A. Mehta, G. Heineman, Evolving legacy systems features using regression test cases and components, in: Proceedings ACM International Workshop on Principles of Software Evolution, ACM Press, New York NY, 2002, pp. 190–193.
10. T. Eisenbarth, R. Koschke, D. Simon, Locating features in source code, IEEE Computer 29 (3) (2003) 210–224.
11. L. Renggli, Magritte — meta-described web application development, Master's thesis, University of Bern (Jun. 2006).
12. D. Jerding, J. Stasko, T. Ball, Visualizing message patterns in object-oriented program executions, Tech. Rep. GIT-GVU-96-15, Georgia Institute of Technology (May 1996).

13. D. Licata, C. Harris, S. Krishnamurthi, The feature signatures of evolving programs, in: Proceedings IEEE International Conference on Automated Software Engineering, IEEE Computer Society Press, Los Alamitos CA, 2003, pp. 281–285.
14. O. Greevy, Enriching reverse engineering with feature analysis, Ph.D. thesis, University of Berne (May 2007).
15. J.-M. S. Philippe Flajolet, Paolo Sipala, Analytic variations on the common subexpression problem, in: Automata, Languages, and Programming, Vol. 443 of LNCS, Springer Verlag, 1990, pp. 220–234.
16. A. Hamou-Lhadj, T. Lethbridge, An efficient algorithm for detecting patterns in traces of procedure calls, in: Proceedings of 1st International Workshop on Dynamic Analysis (WODA), 2003.
17. G. K. Kanji, 100 Statistical Tests, SAGE Publications, 1999.
18. F. Wilcoxon, Individual Comparisons by Ranking Methods, International Biometric Society, 1945.
19. M. O'Brien, J. Buckley, C. Exton, Empirically studying software practitioners - bridging the gap between theory and practice, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE Computer Society Press, 2005.
20. A. Zeller, Isolating cause-effect chains from computer programs, in: SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, ACM Press, New York, NY, USA, 2002, pp. 1–10.
21. N. Wilde, M. Scully, Software reconnaisance: Mapping program features to code, Software Maintenance: Research and Practice 7 (1) (1995) 49–62.
22. A. Hamou-Lhadj, E. Braun, D. Amyot, T. Lethbridge, Recovering behavioral design models from execution traces, in: Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), IEEE Computer Society Press, Los Alamitos CA, 2005, pp. 112–121.
23. F. Nielson, The typed lambda-calculus with first-class processes, in: E. Odijk, J.-C. Syre (Eds.), Proceedings PARLE '89, Vol II, LNCS 366, Springer-Verlag, Eindhoven, 1989, pp. 357–373.
24. J.-B. Chen, S. C. Lee, Generation and Reorganization of Subtype hierarchies, Journal of Object Oriented Programming 8 (8) (1996) 26–35.
25. K. Brade, M. Guzdial, M. Steckel, E. Soloway, Whorf: A visualization tool for software maintenance, in: Proceedings of IEEE Workshop on Visual Languages, IEEE Society Press, 1992, pp. 148–154.
26. O. Greevy, M. Lanza, C. Wysseier, Visualizing live software systems in 3D, in: Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization), 2006.
27. M. F. Kleyn, P. 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), Vol. 23, ACM Press, 1988, pp. 191–205.
28. S. P. Reiss, Visualizing Java in action, in: Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization), 2003, pp. 57–66.
29. D. Lange, Y. 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), ACM Press, New York NY, 1995, pp. 342–357.
30. W. De Pauw, R. Helm, D. Kimelman, J. Vlissides, Visualizing the behavior of object-oriented systems, in: Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), 1993, pp. 326–337.