A Diagnosis-Based Approach to Software Comprehension

Alexandre Perez
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
alexandre.perez@fe.up.pt

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
rui@computer.org

ABSTRACT

Program comprehension is a time-consuming task performed during the process of reusing, reengineering, and enhancing existing systems. Currently, there are tools to assist in program comprehension by means of dynamic analysis, but, e.g., most cannot identify the topology and the interactions of a certain functionality in need of change, especially when used in large, real-world software applications. We propose an approach, coined Spectrum-based Feature Comprehension (SFC), that borrows techniques used for automatic software-fault-localization, which were proven to be effective even when debugging large applications in resource-constrained environments. SFC analyses the program by exploiting run-time information from test case executions to compute the components that are important for a given feature (and whether a component is used to implement just one feature or more), helping software engineers to understand how a program is structured and what the functionality's dependencies are. We present a toolset, coined PANGOLIN, that implements SFC and displays its report to the user using an intuitive visualization. A user study with the open-source application Rhino is presented, demonstrating the efficiency of PANGOLIN in locating the components that should be inspected when changing a certain functionality.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms

Keywords

Software Evolution and Maintenance, Fault Diagnosis, Program Comprehension

ICPC '14, June 2-3, 2014, Hyderabad, India

1. INTRODUCTION

Software maintenance is a crucial part of software engineering. The need to add or change new features to existing software applications is becoming more and more prevalent. Furthermore, the ever increasing complexity of software systems and applications renders software maintenance even more challenging.

One of the most daunting tasks of software maintenance is to understand the application at hand [1]. In fact, recent studies point out that developers spend 60% to 80% of their time in comprehension tasks [2]. During this program understanding task, software engineers try to find a way to make both the source-code and the overall program functionality more intelligible. One of these ways is to create a "mental map" of the system structure, its functionality, and the relationships and dependencies between software components [3, 4].

To fully understand how a software application behaves, software engineers need to thoroughly study the source-code, documentation and any other available artifacts. Only then the engineer gains sufficient understanding of the application, enabling him/her to seek, gather, and make use of available information to efficiently conduct the desired maintenance or evolution tasks. This *program comprehension* (also known as *program understanding/software comprehension*) phase is thus resource and time consuming. In fact, studies show that up to 50% of the time needed to complete maintenance tasks is spent on understanding the software application and gaining sufficient knowledge to change the desired functionality [1]. Currently, there are several approaches that focus on dynamic analysis to provide visualizations of the software system, identifying their components and their relationships, *e.g.:* [5, 6, 7]. However, these approaches may not clearly show what code regions the developer needs to inspect in order to change a certain functionality. Another problem regarding dynamic analysis is the fact that program traces of sizable programs encompass large amounts of data [8]. This may lead to not only scalability issues when gathering/storing traces but also challenges concerning visualizing such a vast amount of information.

To address some of the issues of past approaches, we propose an approach, coined Spectrum-based Feature Comprehension (SFC), that exploits techniques used in the software fault-localization domain. Fault-localization techniques exploit coverage information of test cases to calculate the likelihood of each component being faulty, and were shown to be efficient, even for large, resource-constrained environments [9]. Our approach leverages these concepts to provide

an efficient dependency analysis and visualization for software comprehension that does not have the same scalability hindrances as other related work.

To support the effectiveness of our approach, we apply information-foraging-based theory. Information-foraging is a theory to explain and predict how people use environmental information to achieve their goals [10]. It builds its hypothesis upon optimal foraging theory, drawing from noticed similarities between users' information searching patterns and animal food-foraging strategies. Information-foraging theory assumes that human beings have the capability to efficiently filter irrelevant information out, so that they achieve their goal at minimum cost (e.g., time it takes to change a feature).

To assess the effectiveness of our approach, we have conducted a user study with the open software project Rhino. It demonstrated the effectiveness of the SFC approach and its visualizations in aiding users to pinpoint the components that need to be inspected when evolving/changing a certain feature in the application.

The paper makes the following contributions:

- We propose Spectrum-based Feature Comprehension (SFC), an approach that, similar to fault-localization techniques, exploits run-time information from system executions to identify dependencies between components, helping software engineers in understanding how a program is structured.
- We propose an information-foraging theory to support the effectiveness of our approach.
- We provide a toolset, PANGOLIN, providing a visualization of associated and dissociated components of an application functionality.
- A user study with a large, real-world, software project, demonstrating the effectiveness of our approach in locating the components that should be inspected when evolving/changing a certain functionality.

To the best of our knowledge, an approach that leverages spectrum-based fault-localization techniques to improve program understanding has not been described before.

The remainder of this paper is organized as follows: In Section 2 we introduce the concepts relevant to this paper, namely program spectra and fault-localization. Section 3 will present our spectrum-based feature comprehension approach. Section 4 introduces the PANGOLIN toolset that implements our approach as an Eclipse plugin. In Section 5 we describe the user study setup and present its results. We provide an overview of the related work and how it compares to PANGOLIN in Section 6. Finally, in Section 7, we conclude and discuss future work.

		Runs						
	largest() {	1	2	3	4	5	6	s_O
1:	int a,b,c,large;	●	●	●	●	●	●	0.41
2:	print("Enter 3 numbers");	●	●	●	●	●	●	0.41
3:	read(a, b, c);	●	●	●	●	●	●	0.41
4:	if (a>b) {	●	●	●	●	●	●	0.41
5:	if (a>c)	●			●	●	●	0.50
6:	large = a;				●		●	0.0
7:	else large = a; //BUG	●				●		0.71
8:	}	●			●	●	●	0.50
9:	else if (b>c)		●	●				0.0
10:	large = b ;		●					0.0
11:	else large = c;			●				0.0
12:	print("Largest: ",large);	●	●	●	●	●	●	0.41
	} Error vector:	✓	✓	✓	✓	✗	✓	

(a) Test coverage information.

Ranking	s_O	Statement
1	0.71	7: else large = a;
2	0.50	5: if (a>c)
3	0.50	8: }
4	0.41	1: int a,b,c,large;
5	0.41	2: print("Enter 3 numbers");
6	0.41	3: read(a, b, c);
7	0.41	4: if (a>b) {
8	0.41	12: print("Largest: ",large);

(b) Faulty statement ranking.

Figure 1: Example of SFL technique with Ochiai coefficient.

2. PRELIMINARIES

In this section, spectrum-based fault-localization is detailed. After that, an approach to visualize diagnostic reports is presented.

2.1 Spectrum-based Fault-Localization

Spectrum-based Fault-Localization (SFL) is a debugging technique that calculates the likelihood of a software component being faulty [11]. It exploits information from passed and failed system runs. A passed run is a program execution that is completed correctly (*i.e.*, the program behaves as expected), and a failed run is an execution where an error was detected [9]. The criteria for determining if a run has passed or failed can be from a variety of different sources, such as test-case results and program assertions, among others. The execution information gathered for each run is their program spectra.

A program spectrum is a characterization of a program's execution on an input collection [12]. This collection of data consists of counters of flags for each software component, and is gathered at runtime. Software components can be of several detail granularities, such as classes, methods, or statements. A program spectrum provides a view on the dynamic behavior of the system under test [13]. Many types of program spectra exist. This paper focuses on registering whether a component is touched or not during a certain execution, so binary flags can be used for each component, yielding a small memory footprint. This particular form of program spectra is also called hit spectra [13].

The hit spectra of N runs constitute a binary $N \times M$ matrix A, where M corresponds to the instrumented components of the program. Information of passed and failed runs is gathered in an N-length vector e, called the error vector. The pair (A, e) serves as input for the SFL technique.

With this input, the next step consists in identifying what columns of the matrix A (*i.e.*, the hit spectrum for each component) resemble the error vector the most. This is done by quantifying the resemblance between these two vectors by means of *similarity coefficients* [14].

One of the best performing similarity coefficients for fault-localization is the Ochiai coefficient [15]. This coefficient was initially used in the molecular biology domain [16], and is defined as follows:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \quad (1)$$

where $n_{pq}(j)$ is the number of runs in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{10}(j)$ counts the number of times component j has been involved in passing executions, whereas $n_{01}(j)$ counts the number of failing executions that do not exercise component j. Formally, $n_{pq}(j)$ is defined as:

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2)$$

Several other similarity coefficients do exist [17], namely the O'' coefficient, but these are comparable to Ochiai, and do not change how the approach works.

The calculated similarity coefficients rank the software components according to their likelihood of containing the fault. This is done under the assumption that a component with a high similarity to the error vector has a higher probability of being the cause of the observed failure. A list of the software components, sorted by their similarity coefficient, is then presented to the developer. This list is also called *diagnostic report*, and helps developers prioritize their inspection of software components to pinpoint the root cause of the observed failure.

We show Figure 1 as an example of the SFL technique. Under test is a function named `largest()` that reads three integer numbers and prints the largest value. This program contains a fault on line 7 – it should read `large = c;`. Figure 1a shows the function code, as well as the coverage trace and outcome of six test cases. With this information, similarity coefficients can be calculated for each line using the Ochiai coefficient (Equation (1)). The bigger the coefficient, the more likely it is that a line contains a fault.

Tools that use SFL for their diagnosis, such as Zoltar [18], rank these similarity coefficients to form an ordered list of the probable faulty statements (also referred to as diagnostic report), and present it to the user. Users can then start inspecting the statements located in the higher positions of the diagnostic report, until they reach the faulty statement. Figure 1b depicts the diagnostic report generated by these tools for our `largest()` example.

2.2 Diagnostic Report Visualization

To improve the intuitiveness of the diagnostic report generated by SFL, interactive visualization techniques have been proposed and are available within the GZoltar toolset[1] [19, 20]. GZoltar is a fault-localization plugin for the Eclipse[2] integrated development environment. Besides fault-localization, GZoltar also provides mechanisms for test-suite minimization and prioritization [21], but those are beyond the scope of this paper.

GZoltar provides developers with several different visualizations, namely sunburst, vertical partition, and bubble hierarchy. According to user feedback, the sunburst visualization was deemed the most intuitive [20]. In this visualization, each ring denotes a hierarchical level of the source-code organization, as depicted in Figure 2. From the inner to the outer circle, this visualization presents projects, packages, files, classes, methods, and lines of code. Navigation in this visualization is done by clicking on a component, which will display all the inner components of the selected one. As an example, if a user clicks in a class, all of the class methods will be displayed. Users may also zoom in/out and pan to analyze in detail a specific part of the system. Any inner component can also be set as the new root of the visualization, and only that component's sub-tree is displayed. This operation is called a *root change*. The color of each component in the visualization represents its likelihood of being faulty. Ranging from bright green if the similarity is close to zero; to yellow if the similarity is close to 0.5 and finally to red if the component's similarity to the error vector is close to 1.

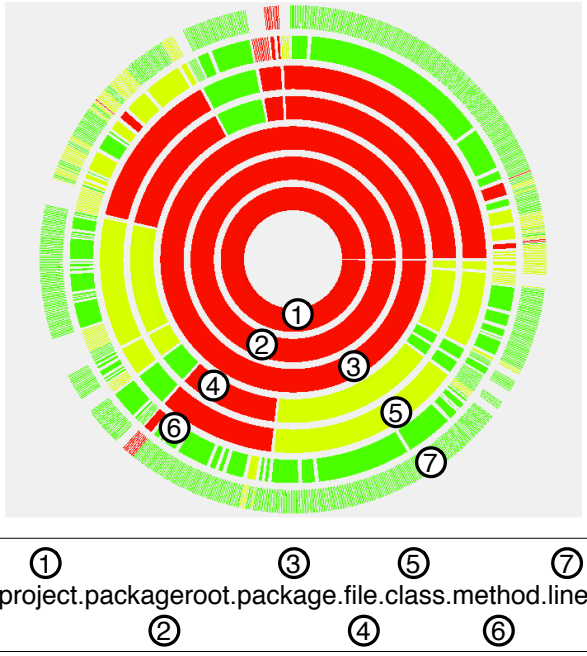


Figure 2: Sunburst Visualization Example.

The effectiveness of these fault-localization techniques and the hierarchical visualization was also demonstrated in a user study [20], where 40 participants, without any knowledge of the system under test, were asked to locate a fault in under 30 minutes. Everyone using the hierarchical visualization with diagnostic information was able to find the bug, whereas only 35% of the participants of the control group succeeded.

3. SPECTRUM-BASED FEATURE COMPREHENSION

This section details our approach coined Spectrum-based Feature Comprehension (SFC) that uses the techniques and code visualizations mentioned in the previous section in the context of program comprehension.

[1]Available at `http://www.gzoltar.com`
[2]Available at `http://www.eclipse.org`

3.1 Concepts & Definitions

To use these fault-localization techniques, its concepts and definitions need to be mapped into the program understanding domain. The first one is the notion of a feature.

Definition 1 *A **feature** is the source-code portion that implements a certain functionality. It can encompass one or more components.*

When trying to evolve/modify a certain feature f, we are interested in the relationships and interactions between f and other components in the source-code. As such, one should not use the error vector e to compute the similarity coefficient (cf. Section 2.1). Instead, an *evolution vector* ev_f should be used.

Definition 2 *The **evolution vector** ev_f is an N-length binary vector. In this vector, a given position i is true (*i.e.*, set as* 1*), if the i^{th} test run exercises feature f.*

When evolving a feature f, it is important to inspect its associated components because they may either call f or be called by f, and thus may need to be modified in accordance with the changes made to f.

Definition 3 *A component j is **associated** with f if its similarity coefficient with f is close to* 1*. This means that when f is executed, j is likely to be executed as well.*

In contrast to associated components, if a component is dissociated to f, it does not need to be inspected when f is modified.

Definition 4 *A component j is **dissociated** to f if its similarity coefficient with f is close to* 0*. This means that when f is executed, j is not likely to be touched by the execution.*

Components with a similarity coefficient of neither 0 nor 1 should be inspected, but only modified with great care. This is because these components are shared amongst features.

3.2 Approach

The SFC approach is depicted in Algorithm 1. The inputs for the algorithm are:

- $\mathcal{P}$ – the program under evaluation.
- $\mathcal{U}$ – set of system runs.
- $\mathcal{U}_f$ – set of runs exercising feature f.

The output is the report $\mathcal{R}$, which is a list of components, each containing its *association measure* (in this paper, measured in terms of the Ochiai similarity coefficient) to the feature under consideration.

First, the variables N and M are set, corresponding to the size of the test suite and the number of components that exist in program $\mathcal{P}$, respectively (Lines 1 and 2).

Subsequently, all system runs $\mathcal{U}$ of the program $\mathcal{P}$ are executed (Line 4), yielding the program spectra matrix A. This matrix contains the execution traces for every system runs in $\mathcal{U}$. Next, the evolution vector is calculated, by comparing each line of the matrix A to the coverage of the runs that exercise the feature U_f.

Algorithm 1 Spectrum-based Feature Comprehension.

Input:

Program $\mathcal{P}$

Set of runs $\mathcal{U}$

Set of runs that exercise the feature $\mathcal{U}_f$

Output:

Report $\mathcal{R}$

1: $N \leftarrow |\mathcal{U}|$
2: $M \leftarrow$ NumComponents($\mathcal{P}$)
3: $\mathcal{R} \leftarrow \varnothing$
4: $A \leftarrow$ Exec($\mathcal{P}$,$\mathcal{U}$)
5: $ev \leftarrow$ Update(A,$\mathcal{U}_f$)
6: $\forall_{j \in \{1...M\}} : n_{01}(j), n_{10}(j), n_{11}(j) \leftarrow 0$
7: $\forall_{j \in \{1...M\}, i \in \{1...N\}} : n_{01}(j) \leftarrow |\{i \mid a_{ij} = 0 \wedge ev_i = 1\}|$
8: $\forall_{j \in \{1...M\}, i \in \{1...N\}} : n_{10}(j) \leftarrow |\{i \mid a_{ij} = 1 \wedge ev_i = 0\}|$
9: $\forall_{j \in \{1...M\}, i \in \{1...N\}} : n_{11}(j) \leftarrow |\{i \mid a_{ij} = 1 \wedge ev_i = 1\}|$
10: $\forall_{j \in \{1...M\}} : \mathcal{R}[j] \leftarrow s_O(n_{01}(j), n_{10}(j), n_{11}(j))$
11: **return** $\mathcal{R}$

Following, in Lines 6 to 9 the occurrence variable n_{pq} (as described in Section 2.1) is calculated for each component of program $\mathcal{P}$.

Finally, for each component, the similarity is calculated with using the Ochiai coefficient (see Equation (1)), and stored in the report $\mathcal{R}$. Unlike what happens in SFL, the report $\mathcal{R}$ does not have to be sorted. This is because the report will not be inspected as a ranking by the user.

In this report $\mathcal{R}$, an association measure is attributed to each component. If a component has an association measure of 1, it means that the component is *associated* with the feature, and only is executed in runs where the feature is exercised. An association measure of 0 means that the component is *dissociated*, and is never called by the feature. Association measures in between 0 and 1 mean that the components can be executed even when the feature is not exercised, so they require further inspection by the developer before modifying them, as they can break other unrelated functionality.

3.3 Complexity Analysis

As for the space complexity, the generated program spectra matrix A has a complexity of $O(M \cdot N)$. The evolution vector and the report $\mathcal{R}$ complexities are $O(N)$ and $O(M)$, respectively. Therefore, the worst case space complexity is $O(M \cdot N + N + M)$.

The time complexity is as follows. Assuming that all system runs in set $\mathcal{U}$ are executed and take the same amount of time to execute, the complexity of this test execution step is $O(N)$. As for the evolution vector computation, its worst case time complexity is $O(M \cdot N)$. The computation of the n_{pq} occurrence function also has a complexity of $O(M \cdot N)$.

Finally, the Ochiai coefficient calculation to populate the report $\mathcal{R}$ is $O(M)$. The worst case time complexity is $O(M \cdot N + N + M)$.

3.4 Report Visualization

To visualize the report produced by our approach, we use the Sunburst visualization, just like the GZOLTAR framework. We apply information-foraging theory to explain the usefulness of the sunburst visualization for improving source-code understanding. Information-foraging theory aims to both *"explain and predict how people will best shape themselves for their information environments and how information environments can be shaped for people"*, as defined by Peter Pirolli [10].

This theory is itself based on optimal foraging theory, that tries to explain the behavior of *predators* and *preys*. *Predators* try to find *preys* by following their *scent*, and *preys* are more likely to be in places (or *patches*) where the *scent* is more intense. In the information-foraging context, the *predators* are the people in need of information and the *preys* are the information itself. The *scent* is the interpretation of the environment by the *predators*. Sjoberg *et al.* [22] suggests that a theory is best used to explain (at least one) of the following questions: what is, why, forecast future events, and guiding how to do something. Information-foraging theory can be used to answer all these questions.

In order to apply this information-foraging theory in the context of automatically generated diagnostic reports, a mapping between the theory constructs and this context must be established. Closely following the theory proposed by Lawrence *el al.* [23] for the context of debugging, in this paper we map the information-foraging theory constructs as follows:

- *Predator* is the person performing the maintenance task;
- *Prey* is what the *predator* seeks to know to pinpoint the code regions that need to be changed;
- *Information patches* are localities in the source-code that may contain the *prey*;
- *Proximal cues* are the runtime behaviors that suggest scent related to to the *prey*;
- *Information scent* is the *predator* interpretation of the report;
- *Topology* is the collection of paths through the source-code and report through which the programmer can navigate. In essence, it includes IDE features that help navigating the code.

The *topology* is a graph representing elements of the source-code (e.g., classes, methods) and the diagnostic report with navigable links between elements. The navigable links between the elements allow the programmer to traverse the connection at the cost of just one click. Information-foraging theory assumes that the developer's choices are an attempt to maximize the information gain per navigation interaction's cost. As in [24], this can be characterized as

$$choice = max\left(\frac{G}{C}\right)$$

where G is the information gain and C is the cost of the interaction (including both the visualization and the IDE features). Since the G and C values are not known to the developer *a priori*, his decisions will be based on the expected gain and cost.

When looking for the code regions that need to be changed, the developer relies on the cues to decide which *place* to inspect next. Those cues are used to estimate the trade-off between the navigation cost and the value to be gained. In an attempt to take the best decision, the developer will favor links whose cues will lead him to the location of code regions in need of change. Better cues are therefore more likely to lead to better information scent, hence reducing the cost incurred while maximizing the value gained.

By analyzing the Sunburst visualization described in Section 2.2 in regard to information-foraging, we may argue that its visualization of the system's topology and its interaction features can indeed reduce the cost of navigation through the various system components (be it packages, classes, methods, even statements) and thus C is reduced. The color coding of each component, which is obtained from the fault-localization ranking, can be regarded as a proximal cue, guiding the developer towards likely associated regions of the source-code and at the same time, notifying the developer about regions that should not be explored (where, e.g., relevant executions have not touched). Hence, a better information scent is conveyed to the developer, increasing the information gain.

4. PANGOLIN TOOLSET

In this Section, we introduce the PANGOLIN toolset[3], an Eclipse plugin that implements the SFC approach and displays its results with the aid of a sunburst visualization.

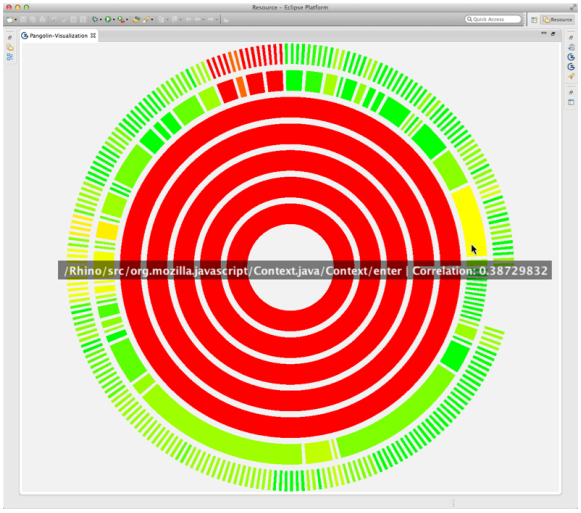


Figure 3: Pangolin's sunburst visualization.

The PANGOLIN plugin performs a dynamic analysis by instrumenting the project, so that the activity matrix is gathered during runtime. The plugin uses the project's JUnit test cases as the set of system runs. In order to perform the analysis, users must also identify in a specific view which system runs exercise the feature under consid-

[3]PANGOLIN is available online. To install the PANGOLIN toolset, users need to request a license at `www.gzoltar.com/pangolin`.

eration. After that, PANGOLIN plugin computes a feature-association-measure for every component in the project, and displays that information in a sunburst visualization, as shown in Figure 3. This sunburst visualization depicts the current project's topology in a hierarchic fashion, starting from the root component representing the whole project in the inner circle, up to individual lines of code in the outer circle. Each component is color coded with the corresponding association measure, ranging from bright green if the association measure is close to zero; to yellow if the association measure is close to 0.5 and to red it is close to 1. When a user hovers the mouse on a component, a label identifying that component and its association measure is shown, as depicted in Figure 3. If he/she clicks that component, Eclipse's code editor will open and the cursor is positioned on the start of the chosen component.

We also enhanced the sunburst visualization to show a summary of what each code class is responsible for. We rank every term used in each class file and apply term frequency-inverse document frequency (*tf-idf*) weighing, commonly used in the Information Retrieval domain [25]. The *tf-idf* value increases proportionally to the number of times a term appears in the document (in this case, a class file). However, it is also offset by the frequency of that term in the overall collection of documents, so that common terms have less weight. Top terms in the ranking are shown when hovering a class component in the visualization, with the intent of providing more cues to the developer, improving his understanding of the program.

5. USER STUDY

In this section, we evaluate the effectiveness of SFC when applied to a real-world application – the Rhino project. First, we describe the subject of our evaluation and the setup for our user study. Afterward, we present the results of the user study and potential threats to validity.

5.1 The Rhino Project

The software application under consideration for this case study is the open-source project Rhino[4]. Rhino is a Javascript engine written entirely in Java and is managed by the Mozilla Foundation. It is typically embedded into Java applications to provide scripting to end users and also allows JavaScript programs to leverage Java platform APIs. Rhino automatically handles the conversion of JavaScript primitives to Java primitives, and vice versa (*i.e.*: JavaScript scripts can set and query Java properties and invoke Java methods). Rhino is comprised by 28 packages, 433 classes and 75170 source lines of code. Furthermore, this project contains 448 unit tests, written for the JUnit framework.

5.2 User Study Setup

The user study was performed by 108 students enrolled in the Software-Engineering course of the Master in Informatics and Computing Engineering program from the Faculty of Engineering of University of Porto. All participants had at least three years of experience with the Java programming language and were familiar with both the Eclipse IDE and the JUnit testing framework, which are requirements to enroll in the Software Engineering course. None had, however, used Rhino before. Participants were grouped into pairs to perform the requested task, which was also a course requirement.

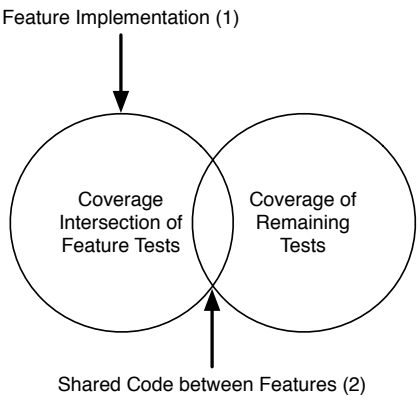


Figure 4: Feature analysis with a coverage tool.

The requested task was the following. Participants were requested to identify source-code regions that (1) exclusively implement a certain feature, and also regions where (2) that feature is being used (*i.e.*, code regions shared among different features). It is important to distinguish between these two kinds of regions when changing a feature. While developers can change regions labeled as (1) without many concerns, regions labeled as (2) require a more detailed inspection before changing the code, as the changes can break other functionalities. The feature under consideration for this user study was Rhino's context creation. This feature is responsible for creating a context, which contains the execution information needed to run Javascript code. One example of the information stored in a context is the call stack representation. A tutorial explaining the feature in detail was given to all participants[5]. The set of tests that exercise the creation of contexts was given to all participants and a time limit of 100 minutes was established to complete the task.

Participants were divided into two groups. One group comprised 26 pairs of participants was asked to use the PANGOLIN plugin to complete the task. As all participants were unfamiliar with PANGOLIN, a short tutorial explaining how to work with the tool (and how to interpret the results) was shown. In order for this group of participants to successfully complete the task, they need to use the tool to indicate the set of tests exercising the feature and run PANGOLIN's analysis. After the analysis is complete, the sunburst visualization appears in the corresponding Eclipse view. To identify the code regions that implement the feature (1), participants should look for components whose association measure is 1 (color coded as red). Code regions shared among several features (2) are components whose association measure is above 0 and below 1, and therefore color coded as different shades of yellow.

The other group of participants comprised by 28 pairs was the control group. Participants were asked to use the features from a standard version of the Eclipse IDE and its

[4]Available at `https://developer.mozilla.org/en-US/docs/Rhino`

[5]All tutorials produced for this user study are available online at `http://gzoltar.com/pangolin/tutorial`

code-coverage plugin EclEmma[6], that shows, for a set of tests, what statements were executed. A short tutorial on how to work with EclEmma plugin was given beforehand. For the task to be successfully completed, participants need to gather the code-coverage information of all tests that exercise the feature, and compute their intersection. The intersection between these tests denotes the code regions that were executed on every test. A set difference between this intersection and the coverage of remaining tests in the test suite allows us to identify the regions that (1) exclusively implement the feature and that (2) are shared among many features, as is depicted in Figure 4.

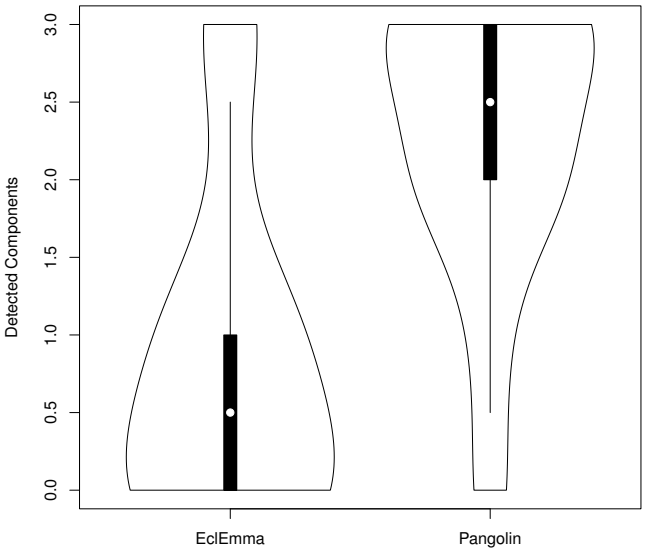
5.3 Results

For the particular feature considered in this user study, Rhino's context creation (described in the previous subsection), users had to identify code regions in 3 classes that exclusively implement the feature (1), and another 41 classes where that feature is used among many others (2).

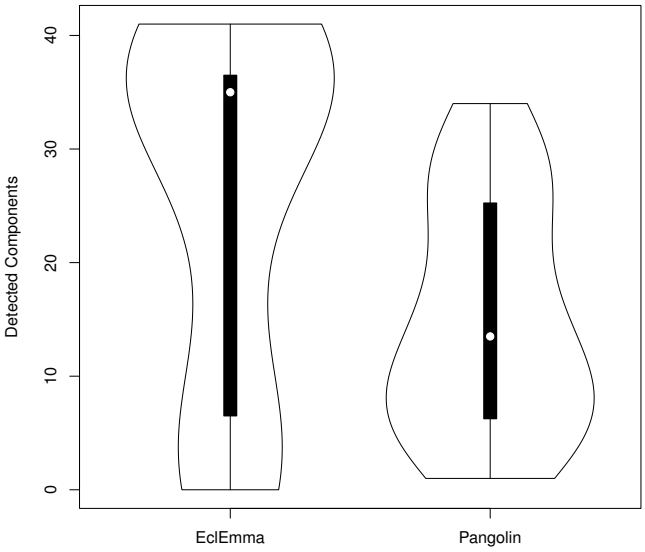
From the group that used the PANGOLIN plugin, participants were able to correctly identify a median of 2.5 classes from category (1) and 13.5 from category (2). In the group that used the code-coverage plugin EclEmma, participants identified a median of 0.5 classes from category (1) and 35 from category (2). Figure 5 shows violin plots[7] depicting the amount of correct components detected by participants. We can see that, for identifying components in category (1), participants working with PANGOLIN were able to achieve better results. In fact, over two thirds of the pairs of participants working with that plugin were able to find at least two correct components (out of three in total), as opposed to participants using EclEmma, where only 5 pairs identified at least two correct components. As for the identification of components from category (2), participants using EclEmma showed an increased overall accuracy when compared to PANGOLIN.

However, and along with the correct code regions, there were several false positives identified by participants. The group using EclEmma registered a median of 6 while identifying regions that exclusively implement the feature, while the group using PANGOLIN registered a median of 0 false positives. The second category, code regions with shared features, yielded a median of 53.5 false positives when using EclEmma versus only 1 false positive when PANGOLIN is used. The amount of false positives each pair of participants identified can be seen in the violin plots from Figure 6. Figure 6a concerns the false positives while identifying code regions for category (1), whereas Figure 6b shows the false positives while identifying category (2). In both categories, we see a substantial increase in the amount of false positives when the code-coverage EclEmma plugin is used to perform the requested task. This happens because the majority of participants using EclEmma, after gathering code-coverages for the indicted test cases, did not perform an intersection of the traces, as depicted in Figure 4. As a result, a considerable number of components were labeled incorrectly.

The last metric gathered was the time each pair of participants took to complete the task. Although a time limit of 100 minutes was established, only two pairs required that amount to submit their results. Figure 7 depicts the elapsed



(a) Detected implementation components.



(b) Detected shared components.

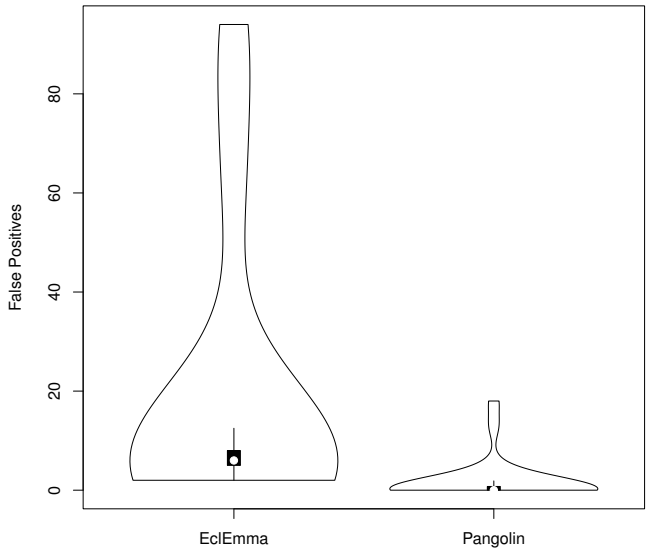
Figure 5: Detected components.

time for each pair of participants. In this plot, the pairs of participants were sorted by ascending order of time taken. This was done with the aim of increasing legibility. Overall, the group using PANGOLIN completed the task in less time compared to the group using EclEmma. Participants using PANGOLIN took a median of 50 minutes to complete, whereas participants working with the EclEmma plugin took 60 minutes. The main reason for participants using EclEmma taking longer to complete the task is the fact that, after gathering the coverage information, they needed to perform the coverage analysis as shown in Figure 4. Participants using PANGOLIN only needed to gather the information shown to them via the sunburst visualization. No extra analysis was required.

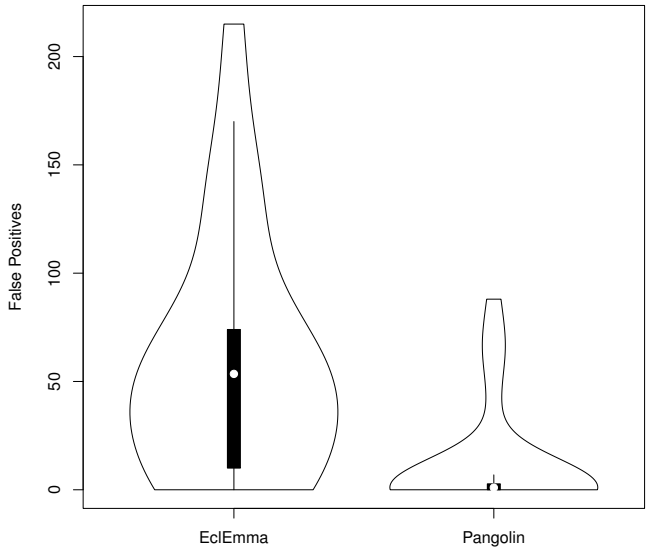
We also performed statistical tests to assess wether the gathered metrics yielded statistically significantly different results. The statistical test used is the Wilcoxon signed-

[6]Available at `http://www.eclemma.org/`

[7]A violin plot is the combination of a box plot and a kernel density plot.



(a) False positives labeled as implementation components.



(b) False positives labeled as shared components.

Figure 6: False positives while identifying system components, sorted by ascending order.

rank [26]. The reason we use Wilcoxon instead of, *e.g.*, Student's t-test is because it does not assume that the data is normally distributed. According to these statistical tests, the two groups of participants can be considered significantly different with 97% confidence.

Results show that the information about the program provided by the SFC analysis and the sunburst visualization is more accurate than requiring users to inspect and compare several traces with a code-coverage tool. Although in category (2), users working with EclEmma were able to detect more components, this approach yielded a large number of false positives, which will most likely increase the comprehension effort, as users will need to inspect those components and deem then dissociated from the feature. From an information-foraging standpoint, we argue that due to the intuitiveness of the visualization and accuracy of SFC,

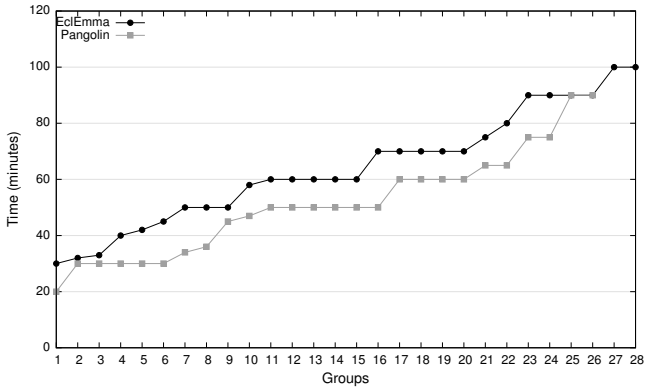


Figure 7: Time required by each pair of participants to complete the task, sorted by ascending order.

Pangolin provides better cues than the EclEmma and, ultimately, increases the perceived information gained about the system being inspected.

5.4 Threats to Validity

The main threat to the external validity of these results is the fact that participants were given the set of tests that exercise the feature under consideration. Information about what features each test is exercising may not be available or difficult to obtain. In fact, software projects may not even have tests for every feature.

Another threat to external validity is the fact that only one feature of one open-source application was used in the study. It is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. Also, all participants in the user study were software engineering students, and the study was performed in an academic setting, so it may not correctly reproduce the problems that the industry deals with. However, we argue that our setting closely resembles an important challenge faced in the software industry regarding program comprehension: introducing junior programmers into well established projects.

6. RELATED WORK

Various techniques and tools were developed as a result of several years of research into trace visualization and feature localization [27, 28]. This section provides an overview – not meant to be exhaustive – of the related work in this area.

6.1 Trace Visualization

De Pauw *et al.* [29, 5] have developed a tool - termed Jinsight - for visually exploring a program's runtime behavior. Although this tool was shown to be useful for program comprehension, scalability concerns render the tool impractical for use in large applications. Reiss [6] states that execution traces are typically too large to be visualized and understood by the user. As such, Reiss proposed a way to select and compact trace data to improve the visualization's intelligibility. Live run-time visualizations have also been proposed as a way to reduce overheads [30], but make it harder to visualize entire executions.

Ducasse *et al.* [31] propose a way of representing condensed runtime metrics (such as attribute-usage frequencies,

object allocation frequencies, object lifetime, among others) with the use of polymetric views. Greevy *et al.* [7] proposes a 3D visualization of the run-time traces of a software system. Greevy displays the amount of information about a component as a tower whose height is influenced by the amount of instances created. The main objective of this technique is to determine which system regions are involved in the execution of a certain feature, but the visualization may not be trivial to grasp.

Cornelissen *et al.* [32, 33] developed a tool - Extravis - that visualizes execution traces by employing two synchronized views: a circular bundle view for structural elements and an interactive overview via a sequence view. Its effectiveness was also demonstrated for three reverse engineering contexts: exploratory program comprehension, feature detection and feature comprehension. Pinzger *et al.* [34] proposes DA4Java, a tool that represents the source-code as a nested graph. Vertices in the graph represent code components, such as packages, classes and methods, and edges represent dependencies (*e.g.*, inheritance or method calls). Graph representations are also used in other works. Such is that of Yazdanshenas *et al.* [35], which like PANGOLIN, was abe to visualize information flow at various abstraction levels. Ishio *et al.* [36] also used graphs to generate interprocedural data-flow paths. However, this analysis can also generate infeasible paths.

Trümper *et al.* [37] implemented the TraceDiff tool, to ease the comparison of large-scale system traces. The tool provides visualizations featuring a modified hierarchical edge-bundling layout and icicle plot-node aggregation, so that the scalability of large traces is addressed. Maletic *et al.* [38] proposes the MosaiCode tool, that uses a 2D metaphor to support the visualization and understanding of various aspects of large scale software systems. It supports multiple coordinated views of these systems and leverages a mosaic visualization to map their characteristics so that it is easy to understand by programmers, managers, and architects. Color and pixel maps are used to represent these characteristics such as lines of code, functions, files, and subsystems. MosaiCode is available as a stand-alone tool and is not integrated with a development environment.

Stengel *et al.* [39] developed the View Infinity tool. It provides a zoomable interface of *software product lines* (SPL). In SPL, software is implemented in terms of reusable user-visible characteristics, and is difficult to understand due to its variability. Like PANGOLIN, this tool offered a customizable granularity visualization, as well as a navigable interface.

The SFC approach proposed in this paper differs from the related work because of the low overhead necessary to compute the *association measures* for all the application's components. Another advantage is that it can not only pinpoint what components should be inspected and what components can be completely disregarded when evolving a feature, but also can warn about the existence of functionality that is not properly modularized.

6.2 Feature Localization and Information-Foraging

Work related to locating features in code includes the software-reconnaissance approach proposed by Wilde *et al.* [40, 41]. This approach tries to answer the question *"In which parts of this program is functionality X implemented?"* using only dynamic information, namely execution traces. Similarly to SFC, the Software Reconnaissance approach distinguishes two sets of test cases (or scenarios): scenarios that activate the feature, and scenarios that do not activate the feature. The former are used to locate the portions of code that implement the feature and the latter are used to reduce the size of those code portions. The SFC approach differs from this approach as it uses similarity coefficients, such as Ochiai, to assert the association of each line of code to the feature.

Information-foraging-based theories to explain information-seeking strategies have been used in the context of program comprehension and software engineering before. Relevant works include those of Ko *et al.* [42] in the context of software maintenance; Romero *et al.* [43], Lawrance *et al.* [23], Flemming *et al.* [24] and Piorkowski *et al.* [44] in software debugging; Chi *et al.* [45] and Spool *et al.* [46] for website design and evaluation.

7. CONCLUSIONS

This paper proposes an automated dynamic method to reduce the effort required by developers in identifying dependencies between features to be evolved (or maintained) and the rest of the program. This approach, coined Spectrum-based Feature Comprehension (SFC), is based on statistics-based methods used in software-fault-localization. These methods exploit run-time information (*program spectra*) of test cases to calculate the likelihood of each component being faulty. In the field of software evolution, the resemblance to the code being evolved is calculated for all components. This can identify associated components, that need to be changed, because they call or are called by the evolving component, and the dissociated components, that can be disregarded by the developer as they do not touch the functionality under consideration.

A theory based on information-foraging to support the effectiveness of our method is also described. This theory draws similarities between users' searching patterns and animal food-foraging strategies. Also, a user study with the open software project Rhino was carried out. It demonstrated the effectiveness of the PANGOLIN approach in pinpointing the components that need to be inspected when evolving/changing a certain feature and those that can safely be disregarded.

For future work, as discussed in Section 5.4, we plan to enhance the PANGOLIN tool so that users can, instead of using test cases, capture manual executions of the application and label them as associated or dissociated with the feature. This way, the SFC analysis can be applied without requiring the project to have an ample test suite or requiring users to peruse extensive test documentation. Users would only need to first run the application and exercise the feature, and then use other dissociated features so that the relevant associated code slice becomes smaller. Second, we plan to extend the amount of information provided to users in the visualization, so that the perceived information gain of exploring a given component is more accurate. One way to do this is by enhancing components when users are hovering them with summaries of what each component is responsible for. We plan to use code summarization-techniques that are based on stereotypes [47, 48].

8. ACKNOWLEDGEMENTS

We would like to thank the students from the Faculty of Engineering of University of Porto for participating in our user study. This work is partially funded by the ERDF through the Programme COMPETE, the Portuguese Government through FCT - Foundation for Science and Technology, project reference FCOMP-01-0124-FEDER-020484.

9. REFERENCES

[1] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, 1989.

[2] R. Tiarks. What programmers really do - an observational study. *Softwaretechnik-Trends*, 31(2), 2011.

[3] D.B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, 1997.

[4] M. Renieris and S. P. Reiss. Almost: exploring program traces. In *Proceedings of Workshop on New Paradigms in Information Visualization and Manipulation (NPIVM'99)*, pages 70–77, 1999.

[5] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 219–234, 1998.

[6] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 221–230, 2001.

[7] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3D. In *Proceedings of ACM symposium on Software Visualization (SoftVis '06)*, pages 47–56, 2006.

[8] A. Zaidman. Scalability solutions for program comprehension through dynamic analysis. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 327–330, 2006.

[9] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[10] P. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, Inc., New York, NY, USA, 1 edition, 2007.

[11] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques – Mutation (Mutation'07)*, pages 89–98, 2007.

[12] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of ACM SIGSOFT international symposium on Foundations of software engineering (ESEC'97/FSE'5)*, pages 432–449, 1997.

[13] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *STVR Journal of Software Testing, Verification, and Reliability*, (3):171–194, 2000.

[14] A.K. Jain and R.C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[15] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, 2006.

[16] A. da Silva Meyer, A.A.F. Garcia, A.P. de Souza, and C.L. de Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l.). *Genetics and Molecular Biology*, 27:83–91, 2004.

[17] L. Naish, H. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology*, 20(3):11, 2011.

[18] T. Janssen, R. Abreu, and A.J.C. van Gemund. Zoltar: A Toolset for Automatic Fault Localization. In *Proceedings of International Conference on Automated Software Engineering (ASE'09)*, pages 662–664, 2009.

[19] J. Campos, A. Riboira, A. Perez, and R. Abreu. GZoltar: An eclipse plug-in for testing and debugging. In *Proceedings of International Conference on Automated Software Engineering (ASE'12)*, pages 378–381, 2012.

[20] C. Gouveia, J. Campos, and R. Abreu. Using HTML5 Visualizations in Software Fault Localization. In *Proceedings of IEEE Working Conference on Software Visualization (VISSOFT'13)*, pages 1–10, 2013.

[21] Jose Campos and Rui Abreu. Leveraging a Constraint Solver for Minimizing Test Suites. In *Proceedings of the 13th International Conference on Quality Software (QSIC '13)*, pages 253–259, 2013.

[22] D. Sjøberg, T. Dybå, B. Anda, and J. Hannay. Building theories in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer London, 2008.

[23] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[24] S. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):14:1–14:41, 2013.

[25] C. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[26] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[27] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[28] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[29] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 326–337, 1993.

[30] Steven P. Reiss. Visualizing java in action. In *Proceedings of ACM Symposium on Software Visualization (SoftVis'03)*, pages 57–65, 2003.

[31] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 309–318, 2004.

[32] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of International Conference on Program Comprehension (ICPC'07)*, pages 49–58, 2007.

[33] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[34] M. Pinzger, K. Grafenhain, P. Knab, and H.C. Gall. A tool for visual understanding of source code dependencies. In *Proceedings of International Conference on Program Comprehension (ICPC'08)*, pages 254–259, 2008.

[35] A.R. Yazdanshenas and L. Moonen. Tracking and visualizing information flow in component-based systems. In *Proceedings of International Conference on Program Comprehension (ICPC'12)*, pages 143–152, 2012.

[36] T. Ishio, S. Etsuda, and K. Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading. In *Proceedings of International Conference on Program Comprehension (ICPC'12)*, pages 37–46, 2012.

[37] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *Proceedings of IEEE 21st International Conference on Program Comprehension (ICPC'13)*, pages 53–62, 2013.

[38] J. Maletic, D. Mosora, C. Newman, M. Collard, A.Sutton, and B. Robinson. Mosaicode: Visualizing large scale software: A tool demonstration. In *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2011*, pages 1–4, 2011.

[39] M. Stengel, Mathias Frisch, S. Apel, J. Feigenspan, C. Kästner, and Raimund Dachselt. View infinity: A zoomable interface for feature-oriented software development. In *Proceedings of International Conference on Software Engineering (ICSE'11)*, pages 1031–1033, 2011.

[40] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of Conference on Software Maintenance*, pages 200–205, 1992.

[41] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[42] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[43] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Man-Machine Studies*, 65(12):992–1009, 2007.

[44] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. In *Proceedings of Conference on Human Factors in Computing Systems (CHI'12)*, pages 1471–1480, 2012.

[45] E. Chi, P. Pirolli, K. Chen, and J. Pitkow. Using information scent to model user information needs and actions and the web. In *Proceedings of Conference on Human Factors in Computing Systems (CHI'01)*, pages 490–497, 2001.

[46] J. Spool, C. Perfetti, and D. Brittan. *Designing for the scent of information*. User Interface Engineering, 2004.

[47] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of IEEE 21st International Conference on Program Comprehension (ICPC'13)*, pages 23–32, 2013.

[48] N. Alhindawi, N. Dragan, M. Collard, and J. Maletic. Improving feature location by enhancing source code with stereotypes. In *Proceedings of International Conference on Software Maintenance, Eindhoven (ICSM'13)*, pages 300–309, 2013.