On the impact of trace-based feature location in the performance of software maintainers

Marcelo de Almeida Maia, Raquel Fialho Lafetá

Faculty of Computing
Federal University of Uberlândia
Av. João Naves de Ávila, 2121. Bloco B. Uberlândia, MG, 38400-902, Brasil

Abstract

Software maintainers frequently strive to locate source code related to specific software features. This situation is mostly observable when features are scattered in the code. Considering this problem, several approaches for feature location using execution traces have been developed. Nonetheless, the practice of post-mortem analysis based on execution traces is not fully incorporated in the daily practice of software maintainers. Empirical studies that reveal strengths and weaknesses on the use of execution traces in maintenance activities could better explain the role of execution traces in software maintenance. This study reports on a controlled experiment conducted with maintainers performing actual maintenance activities on systems of different sizes unknown to them. There are benefits from systematic use of execution traces: the reduction of the maintenance activity time and greater accuracy of the activity outcome. Other qualitative observations were the lower level of activity difficulty perceived by the participants that used execution trace information and that this kind of information seems to be less useful in maintenance activities where the problem of feature scattering does not occur clearly.

Keywords: empirical assessment, execution traces, feature location, software maintenance.

Email addresses: marcmaia@facom.ufu.br (Marcelo de Almeida Maia), raquel.rafialho@gmail.com.br (Raquel Fialho Lafetá)

1. Introduction

Maintenance requests are usually triggered by users and can be associated with software features. The term *feature* has been used in several contexts. For example, in the context of feature-oriented programming, features are usually understood as functional increments in software product lines and are modularized with special constructions of the programming language [1, 2, 3]. In the context of software maintenance, the term *feature* is mainly associated with the problem of feature location. In this case, the term *feature* is commonly related to a bug, an enhancement, a patch, or a functionality that can be described from the user point of view. The seminal work in this area is that by Wilde and Scully [4], who defined a feature as a concept that can be related to the source code. Recent work recognized that features can be functional or non-functional, but in the context of feature location, functional features are predominant in the relevant literature and we will consider feature as an observable behavior of the system that can be triggered by the user [5, 6].

The feature location problem is not trivial because features are not usually modularized in the source code. The implementation of a feature can be scattered across several modules and a module can contain code that implements several features. As a matter of fact, feature-oriented software development approaches address the modularization problem [1, 7], but they are not widely adopted. Other kind of solution for feature location would be the use of traceability information previously documented, which returns the source code fragments directly related to some features. However, in general, traceability links are not available in the software documentation, or even when they are available, they are not necessarily up-to-date. To alleviate this problem, some automated solutions to traceability recovery have been proposed, but still there is no widely adopted solution [8, 9]. Generally, these solutions are based on a post-mortem dynamic analysis that relates a feature to fragments of source code using the information retrieved from traces generated during the execution of the feature [4, 5, 10, 11]. However, the use of dynamic analysis is not a widespread practice of software maintainers. We could raise several hypotheses to explain this situation, for example, the lack of formal training with these techniques in programming courses or even the existence of technical difficulties, such as the excessive information that execution traces produce or the dependency on maintainers choosing appropriate execution scenarios.

The goal of this following study is to provide additional understanding on the impact of using execution trace information in the performance of maintainers during software maintenance activities. As already reported by Cornelissen et al. [11], research conducted in the area of program comprehension using dynamic analysis has been poorly supported by empirical evidence based on human performance. They showed that controlled experiments with human subjects are rarely reported in the literature.

Our study is based on an empirical evaluation of approaches that use execution trace information to help maintainers perform software maintenance tasks. The evaluation is supported by a controlled experiment aimed at assessing the effort required to complete maintenance tasks and the correctness of the results provided the maintainer. Our hypothesis is that the use of execution traces could enhance the performance of maintainers because execution traces seem to support feature location, which is a major challenge during maintenance activities. The experiment was conducted with software maintainers executing real maintenance tasks on different software systems. In the experiment, data was collected during the execution of the tasks and also a questionnaire was filled by each participant to acquire qualitative information about their task experience.

The paper is organized as follows. In Section 2, we present the related work. In Section 3, we present the approaches based on the use of execution trace information that will be object of evaluation in this work. In Section 4 we provide details on the experimental setting. In Section 5, the results are presented and in Section 6 they are discussed. In Section 7, the threats to validity of this study are discussed, and finally, in Section 8, the conclusions are presented.

2. Related Work

One of the first studies of feature location using dynamic analysis is Software Reconnaissance [4], which compares traces from different execution scenarios to map features to code elements. The execution scenarios are defined with and without the considered feature. This method focuses on one feature at a time and does not intend to analyze a set of features. The subsequent work following the Software Reconnaissance approach was mainly focused on the approach's evaluation with legacy code in Fortran and comparison with other approaches for feature location, such as search with program dependency graph and text search with *grep* [12, 13, 14].

Concept analysis has been used to enhance a feature location approach based on dynamic and static analyses [5, 15]. The approach distinguishes computational units that are general from those that are specific to the analyzed features. The approach consists in the definition of a scenario set, where each scenario executes one or more features and one feature can be executed in one or more scenario. Formal concept analysis is used to produce a feature-unit mapping. Eisenbarth et al. [5] also have shown that the combination of static and dynamic analysis are important to produce better quality mapping that reduces the search space for maintainers.

A technique named *Dynamic Feature Traces – DFTs* used an heuristic ranking to determine the relevance of code elements to features [10]. DFTs are created in three steps: i) developer partitions a test suite, which is required to be large, to be comprehensive, and to provide a correlation between features and test cases; ii) the tool performs the trace extraction and analysis; and iii) the tool generates the call sets and the ranks. The rank denotes how relevant is a code element to a feature. The rank is defined by a heuristic that considers three criteria: 1) the number of occurrences of a method in the test case of a feature; 2) the participation of a method in test cases of different features; 3) the depth of the call, i.e., the deeper the relative depth, the lower the respective rank. Eisenberg and Volder observed that the technique produces better results when applied to a large number of scenarios and that incomplete scenarios can produce poor results. We will adopt, in the approach evaluated in this work, a similar ranking based on the level of participation of classes in features.

Greevy et al. [16] combined static models of source code with dynamic models to map features to source code elements. They proposed views that help to analyze the evolution of systems, describing changes in a way that shows how many and which features are impacted by them. They classified code elements based on their participation level in the feature implementation, which has some similarity with the rank criteria of Eisenberg and Volder [10] and is the basis of the class classification schema that will be shown in the next section.

Simmons et al. [17] conducted an exploratory study to establish the steps necessary to feature location using known industrial tools. They concluded that three steps are necessary: dynamic analysis, trace differentiation, and static analysis. They combined two industrial tools, Metrowerks CodeTEST and Klocwork inSight, with an academic tool for trace comparison, TraceGraph [18]. They reported on problems in the operational details integrating

the tools but they have shown that the tool combination was effective. The data were obtained from only two human subjects working on the Apache system, producing eight data points for analysis.

ConcernMapper [19] was proposed to map manually concerns to the related source code. In our work, we use *ConcernMapper* to map features to the related source code elements (classes and methods), and to enable further investigation of these elements using the native functionalities of the IDE. A complementary work named Concern Graphs [20] was proposed to locate and document concerns. The approach does not use dynamic analysis but works on the idea of enhancing the comprehension process providing a manual mapping between concerns and source code elements.

Revelle and Poshyvanyk [21] presented an exploratory study of 10 feature location techniques that use several combinations of textual, dynamic, and static analyses. They observed that there is no approach that outperforms all others but observed that combining analyses generally improves results, reinforcing the integrated use of dynamic and static approaches.

Wang et al. [22] investigated feature location techniques based on information retrieval and static/dynamic analyses. They executed an exploratory study consisting of two experiments in which developers were given unfamiliar systems and asked to complete feature location tasks. Their results provided a framework for a feature location process based on phases, which consists of collections of concrete actions oriented to the purpose of the phase. They identified 11 types of physical actions and six types of mental actions. Also, they identified that the actions are not independent of each other, i.e., they are organized in patterns.

Quante [23, 24] introduced a technique to extract architectural information based on *Dynamic Object Process Graphs (DOPGs)*, which describe the application control flow from the perspective of an object. In fact, DOPGs are projections of well-known interprocedural control flow graphs. This technique was evaluated empirically, with 25 Computer Science students, on the response time to finish a comprehension task and on the correctness of the solutions [25]. They concluded that DOPGs do not support program comprehension in general because it depends on the subject systems, on the tasks and on the choice of DOPG objects. Although Quante's comprehension approach is fundamentally different from ours, the proposed controlled experiment has a similar framework, except that they did not establish an upper-bound time limit to execute each experimental task and the tasks were only the comprehension of pre-generated graphs, without having neither to

perform any code modification, nor generating the dynamic information for further use. Their session time lasted for two hours.

Cornelissen et al. [26] claimed that execution trace visualization is an important approach for software understanding and created EXTRAVIS [27], a tool for the visualization of large traces. EXTRAVIS provides a massive sequence view, which is essentially a large-scale UML sequence diagram and a circular bundle view that hierarchically projects the structural entities (packages, classes, methods) of a program on a circle and show their interrelationships in a bundled fashion. They conducted an empirical evaluation with 34 human subjects to assess EXTRAVIS added-value in maintenance contexts, measuring how the tool affects the time that is needed for typical comprehension tasks and the correctness of the given solutions. Their results provided initial evidences on the benefits of EXTRAVIS, showing a decrease in needed time and an increase in correctness. Similarly to Quante's work, the tasks were only for comprehension of using pre-built EXTRAVIS views, without having neither to collect traces and generate the views nor to perform any code modification.

There are other approaches for feature location that are not specifically based on dynamic informatic. Some approaches uses static analysis or information retrieval methods. Robillard and Murphy [28] propose Concern Graphs that encapsulates a subset of program elements and a set of relations between them. The relations are based on static dependencies between program elements. Robillard [29] introduced an approach for analyzing the topology of structural dependencies in a program. From this topology, relevant elements are proposed for the developer to investigate. The relevance criteria are based on previous input set provided by the developer. Marcus et al. [30] use Latent Semantic Indexing to map feature descriptions expressed in natural language to source code elements. Zhao et al. [31] present the SNIAFL approach to feature location, which is based on a information retrieval technique and also on static branch-reserving call graph. Dit et al. [32] has produced a taxonomy and survey about methods for feature location, where the reader can find a comprehensive coverage of the feature location area.

3. An Analysis Approach Based on Execution Traces for Software Maintenance

To evaluate if execution traces plays an important role during maintenance tasks, we had to choose a dynamic analysis approach based on execution traces to assess the performance of the maintainer using such approach. Although there are some possibilities in the literature that could be chosen for this purpose, we decided not to use an approach as-it-is. Instead, the choice was to adapt tools and concepts used in other work [33, 19, 10, 16]. Moreover, we do not advocate that this is a novel approach, because in fact, feature location approaches usually have three common steps: dynamic analysis, trace differentiation and static analysis [17]. Our approach also has these steps that are similar to others already shown in the literature [4, 5, 10, 17].

The approach aims at helping maintenance activities using dynamic and static analyses to facilitate feature location. It is designed for object-oriented systems and is implemented for Java programs. We decided to separate the independent variables with three values according the approach used by the subjects. Below, we briefly show these values, which will be further explained in Section 4:

- a control approach, which provides only the conventional IDE functionality;
- a simple approach, which provides one view for feature location that associates classes and methods with features, and
- an enhanced approach, which has the same main view as the simple approach and three additional secondary views: 1) class classification for a feature group, 2) mapping of features to source code elements with class classification, and 3) a filtered method call view for a selected method, containing only methods executed within the specific feature scenario.

The motivation for this separation is that we could assess if the central idea (simple approach) based on using the information on which classes and methods were executed for a specific feature plays an effective role in the maintainer performance and also we could analyze if the use of more views related to feature location (enhanced approach) would significantly enhance the maintainers' performance.

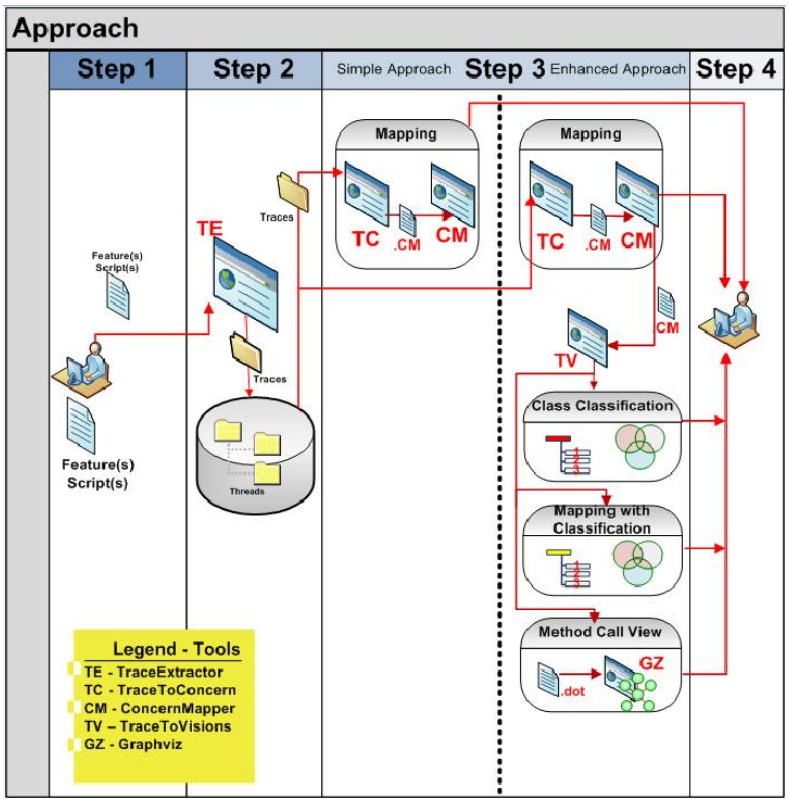

Figure 1: Overview of the dynamic and static analysis approach.

The whole approach is organized in several steps, shown in the four swimlanes of Figure 1: execution scenario planning (Step 1), execution trace extraction (Step 2), generation of the aforementioned views (Step 3), and view-driven static analysis (Step 4).

3.1. Execution Scenario Planning

Post-mortem dynamic analysis requires execution traces related to the feature being analyzed. These traces are extracted during the execution of a scenario that represents adequately the feature. The simple approach and the enhanced approach requires planning a set of feature's scenarios that

should be analyzed in group. It is possible that the set of scenarios can be a singleton.

Scenarios should be carefully specified, otherwise false positive and false negative source code elements may be mapped to features. The maintainers should consider a comprehensive set of tasks and input values that should be defined in the scenarios that are strictly related to the features. However, it must be considered that, independently of well-designed scenarios, false positive and false negative elements are expected to occur in some degree.

3.2. Trace Extraction

Every approach based on dynamic analysis needs some mechanisms to extract data from program executions. Sobreira and Maia [33] proposed a tool to analyze feature scattering across the code with matrix visualization. The evaluated approach uses their trace extractor, which instruments the system with an AspectJ module to capture method-call events generated during execution. During the extraction process, trace files for each triggered thread are generated. The method-call event data contains the fully qualified method name, the system time of the call entry, the stack level of the call, and the identifier of the associated object or class. In the approach proposed in this work, only the qualified method name will be used from the collected traces to be shown in the views. The traces for each thread of a specific execution scenario are combined using the union operation from set theory.

3.3. View Generation

Four views can be generated after trace extraction. These views will be described below. In the experiment setting that will be described in Section 4, the maintainers will be responsible to extract the traces and generate the views according to their corresponding approach group.

Mapping features to code. The mapping of features to their corresponding implementing classes and methods is the main view of this approach because it is the initial search space for feature location. This view is generated from traces, extracting method-call events that occurred during the execution of the respective feature. The result of the extraction is an input file that can be imported by the *ConcernMapper* tool [19]. Each feature will be interpreted as a concern in this tool. Figure 2(a) (Area 1) shows an example of a feature and some corresponding classes and methods that were mapped to that feature. For example, the figure shows that for the feature *"Change the size of the comment"*, the class *"DefaultUndoManager"* (among others)

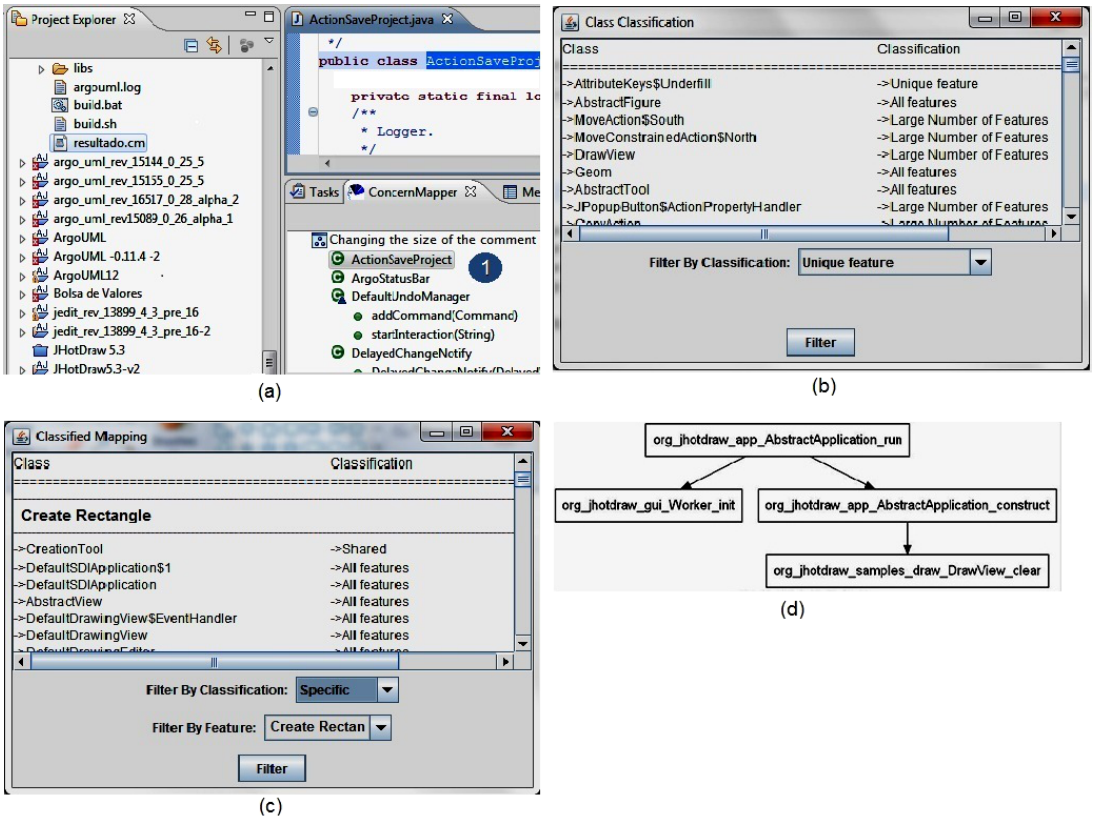

Figure 2: Views produced with execution trace information

had the methods *"addCommand(Command)"* and *"startInteraction(String)"* executed. Maintainers can have access to further static information of those classes and methods because *ConcernMapper* is smoothly integrated in the Eclipse IDE.

Class classification. This view classifies the classes that implement features based on the *level of participation* of each class in the implementation of the set of features being analyzed. :The classification is relative to participation of a class in that set of different features. The criteria for class classification are based on previous work [10, 16]. There are four distinct participation levels and they are calculated from the number of analyzed features (NOF) and the number of those features implemented by the class (NOFC).

- *Unique feature* is a class that participates in the implementation of one

feature in the set (NOFC = 1). This class is very specific to one feature implementation.

- *Small number of features* is a class that participates in more than one feature but in less than half of the features in the set (NOFC > 1) ∧ (NOFC < NOF / 2).
- *Large number of features* is a class that participates in half or more of the features in the set but not in all of them (NOFC > 1) ∧ (NOFC ≥ NOF / 2) ∧ (NOFC ≠ NOF).
- *All features* is a class that participates in the implementation of all features in the set (NOFC = NOF). This class usually corresponds to an utility class.

Figure 2(b) shows an example of the class classification view. Each line shows the class name on the left side and its corresponding classification on the right side. It is possible to filter classes according the level of participation.

Mapping with classification. In the previous view, we have a list of all classes related to the set of features. In this view, the presentation is organized by features, i.e., for each feature, their related classes and corresponding classification are shown. Below, we define the criteria for classification.

- *Specific* is a class the implements only the feature in the context.
- *Shared* is a class that implements the feature in the context and other features, but not all.
- *All features* is a class that implements all analyzed features.

Figure 2(c) shows an example of mapping with classification. At the top, we have the feature name and below a list of classes that implement that feature with the respective classification. It is possible to filter the lines by a specific classification and for a specific feature.

These views show code elements associated to executed features, thus reducing the search space for maintainers. Instead of searching the whole code, the search will be driven by the views, which have a lower number of code elements compared to the whole system source code.

Filtered method call view For each method shown in the mapping view of the Concern Mapper, the maintainer can generate the filtered call

view for these methods. This call view contains both the methods called from the selected method and the methods that call it, considering **only the executed methods** that appeared in the execution trace of a selected feature. Currently, this view is not integrated with the Eclipse IDE. Figure 2(d) shows an example of the call view generated for the method `AbstractApplication.run()` of the JHotdraw system. There are no methods calling this method because it is top-level in the trace.

3.4. Static Analysis Driven from Dynamic Views

During the comprehension process, maintainers may need static information about the system. The integration with the Eclipse IDE provides an additional support for searching information that may be driven by the provided dynamic views. Previous work reported on the importance of the combination of static and dynamic analysis to enhance the comprehension process [5, 17, 34, 35], because the combination of different types of analyses possibly reduces the search space.

4. Study Setting

This section describes the experiment that included four main maintenance tasks on different systems. This experiment was conducted with groups of maintainers, which were not involved in the definition and implementation of the previously-presented approach. These participants were volunteers. Qualitative and quantitative data were collected for subsequent analyses. The process of experimentation was based on Juristo and Moreno's textbook [36]. The experiment was developed to answer the research questions described below.

4.1. Research Questions

We pose the following research questions to better understand if execution trace information hinders or enhances maintenance activities:

- RQ1) Does the use of the proposed approach reduces the maintenance effort of a maintainer compared to a free approach without traces?
- RQ2) Does the use of the proposed approach enhance the correctness of the result provided by a maintainer compared to a free approach without traces?

- RQ3) Does the use of the proposed enhanced approach reduces the maintenance effort of a maintainer compared to the proposed simple approach?

- RQ4) Does the use of the proposed enhanced approach enhance the correctness of the result provided by a maintainer compared to the proposed simple approach?

Next, we describe our null and alternative hypotheses associated with this study for each research question.

- $H0_{RQ1}$ - Developers using the proposed approach do not require less effort to perform the maintenance tasks compared to maintainers using a control approach without traces on unknown systems for them.

- $H1_{RQ1}$ - Developers using the proposed approach require less effort to perform the maintenance tasks compared to maintainers using a control approach without traces on unknown systems for them.

- $H0_{RQ2}$ - Developers using the proposed approach would not produce more correct results during maintenance tasks compared to maintainers using a control approach without traces.

- $H1_{RQ2}$ - Developers using the proposed approach would produce more correct results during maintenance tasks compared to maintainers using a control approach without traces.

- $H0_{RQ3}$ - Developers using the enhanced approach would not require less effort compared to maintainers using the simple approach.

- $H1_{RQ3}$ - Developers using the enhanced approach would require less effort compared to maintainers using the simple approach.

- $H0_{RQ4}$ - Developers using the enhanced approach would not produce more correct results compared to maintainers using the simple approach.

- $H1_{RQ4}$ - Developers using the enhanced approach would produce more correct results compared to maintainers using the simple approach.

4.2. Experimental Setting

This section presents four experimental tasks designed to trigger maintainers' behavior to be able to answer the previous research questions and evaluate our hypothesis. No specific criterion for defining the execution order of the tasks and sessions was defined because the participation in one experiment should not interfere in participation in the other, because the tasks are different. In the first task, maintainers had to implement an improvement, necessarily reusing part of the code of a small *Board Games* suite. In the second task, maintainers had to fix a bug in the *Undo* and *Redo* functionality of JHotDraw 7.1, which is a medium-size graphics editing application framework. In the third task, the maintainers had to answer a question involving the location of code elements related to the evolution of features *Undo* and *Redo* of JHotDraw from versions 5.3 to 7.1. In the fourth task, maintainers had to implement an improvement in a reasonably large system, ArgoUML 0.26.1.

The dependent and independent variables are the same for all tasks, allowing analyzing the data in the same way, providing a larger number of observations on the dependent variables and consequently offering a more robust result than when analyzing each task separately.

The independent variable proposed to answer RQ1, RQ2, RQ3 and RQ4 is the approach used during the maintenance task, which values are one the following three groups. The data analysis will verify if there are differences between the results of these three groups.

1. *Control Group.* This group uses none of the four views. Instead, the subject are free to use all native functionalities of the IDE Eclipse Ganymede.
2. *Simple Approach Group.* This group uses the the native functionalities of the IDE Eclipse Ganymede and also the *mapping* view for feature location. Moreover, the use of the approach includes the execution of the corresponding four steps defined in Figure 1. These steps are an overhead compared to the Control Group. The answer to the research questions RQ1 and RQ3 will say if the provided dynamic information can make up for this overhead.
3. *Enhanced Approach Group.* This group uses the native functionalities of the IDE Eclipse Ganymede and the four views (the same *mapping* view from the simple approach, class classification, mapping with classification and filtered method call view) defined in the proposed ap-

proach. As in the simple group, the use of the approach includes the execution of the corresponding four steps defined in Figure 1.

We will also analyze the *Simple* and *Enhanced* group together as one group named *Traces* group, in order to reinforce the result of the influence of the availability of dynamic views in general.

The following dependent variables were defined for each research question:

- For RQ1 and RQ3, the elapsed time to execute the maintenance task.
- For RQ2 and RQ4, the rate of correct results provided by maintainers for the maintenance task. In fact, we should interpret the "correctness" in the sense of "correctness in the available time frame", similarly to school exams.

Group Definition. The selection of human subjects for the experiment was based on a list of personal and professional contacts of the experimenter, who is the second author of this work. A number of 27 developers attended the experimenter's invitation and voluntarily participated in the experiment. 26 out of 27 participants were professional developers from eight different companies at Uberlândia-Brazil and one participant is graduate student of the Post-Graduate Program in Computer Science at Federal University of Uberlândia. 78% were graduate and 22% were final year undergraduate students. It is possible that students were hired before getting their bachelor degree. In our study, only some of the subjects were professionals that were also still students in the final year or semester. The volunteers were interviewed and filled a form about their specific knowledge. The criteria for distributing participants in the groups were randomness, their level of knowledge in Java development, and their software maintenance experience. They were classified as beginner, mid-level, or advanced. The classification was based on the answers to a questionnaire that included questions on the professional experience and an examination on object-oriented programming with Java. Each group (*Control, Simple,* or *Enhanced*) contained three subjects: one beginner, one mid-level, and one advanced, randomly selected, providing a fair balancing. So, as each task is conducted with three groups, nine subjects performed each task. This organization has resulted in 36 individual observations. Because we had 27 subjects available, some of them have participated in more than one observation.

Target Systems. To have more representativeness in the object systems, we decided to study systems of different sizes. We ran the experiment with

Table 1: Target systems.

System	#Classes	#Methods	Activity
Board Games	16	72	Reuse
JHotDraw 7.1	466	4078	Fixing and Location
JHotDraw 5.3	215	1793	Location
ArgoUML 0.26.1	2.022	14.038	Improvement

the following systems: *Board Games*, *JHotDraw*, and *ArgoUML*. Some of the criteria to choose these systems were: implementation in Java with open-source code, reasonably well-documented, and with different versions. Also, they have different sizes: the smaller is the Board Games and the larger is ArgoUML, as shown in Table 1. Moreover, JHotdraw and ArgoUML were selected because they have been extensively used in other studies and this choice is interesting because the results could be further used to establish comparisons with other studies. Also, these systems have non-trivial size and have active source-code repositories. We decided to use a home-made system, the Board Games, to have a small system representative. In fact, we established the choice of those systems when we could define maintenance activities that could be adequately performed in a reasonable time frame.

Maintenance Tasks. Each task was conducted with different activities aiming at covering a broad spectrum of common practices in software maintenance, which are: code reuse to implement new requirements, bug fixing, location of code that implement the same feature in different versions, indication of the changes that has occurred to a feature in different versions, and improvement of an existent feature.

The activities were jointly chosen with the systems. We needed representative systems that would provide representative activities. There are reports pointing out that roughly half of the maintenance effort is related to program comprehension and also that one of the most important task during program comprehension is feature location [37, 38]. Moreover, in a study with 487 organizations, Lientz and Swanson reported that 70% of the type of maintenance activities are corrective and perfective ones [39]. So, we decided to define tasks that included bug localization, improvement, and reuse-based improvement. These tasks are detailed below.

Task 1 - Reuse. The target system was *Board Games*. There are two games in the suite: *Connect* and *Tic-Tac-Toe*. The game *Connect* has a side

menu with the following functionalities: *Count Games, Count Points, Main Menu, New Game*, but the game *Tic-Tac-Toe* does not have such functionalities. The activity consists in reusing the *Connect* code to introduce the same side menu in *Tic-Tac-Toe*. A correct result should provide a new program running according to a predefined test case.

Task 2 - Bug Fixing. The target system was *JHotDraw*. A bug in version 7.1 was selected from the list of fixed bugs in the official site. This bug was *Text Area Tabs* and is related with *Undo* and *Redo* that does not work for the *TextTool* functionality. The task consisted in fixing this bug. A correct result should provide a new version of the program running without the respective failure and pass in the test case predefined by the experimenter. The choice of the bug used in the experiment was not completely random. In fact, the criterion used to choose the bug was that the bug should not be too easy and not too difficult to find, so it could be corrected by the subjects during the time frame defined in the experiment with possibly some variation in time. Moreover, the bug was a real bug (ID 639124), already corrected, extracted from the bug repository at Sourceforge.

Task 3 - Evolution Location. The target system was *JHotDraw*, versions 5.3 and 7.1. Substantial changes have occurred in version 7.1, including the *Undo* and *Redo* features. The subjects had to answer the following questions: *(Q1)* Which classes are related to *Undo* and *Redo* in JHotdraw 7.1? A correct answer to this question should have a correct list of classes. The subjects would receive a partial grade relative to a partial correct list of classes. *(Q2)* Which classes are related to *Undo* and *Redo* in JHotdraw 5.3? A correct answer to this question should have a correct list of classes. *(Q3)* Describe the overall implementation change that occurred with *Undo* and *Redo* from version 5.3 to version 7.1. We considered a correct answer to this question any answer that mentioned that the package structure of the system was restructured to simplify the design.

Task 4 - Improvement. The target system was *ArgoUML*. The activity is the improvement of the feature *Comment* in diagram drawing. In the selected version, when the user inserts a comment in a diagram, the object automatically adapts its size to the size of the text. If the text is too large, this comment may increase indefinitely. The requested *improvement* is to limit the size of the comment object, so that the text should have at most 160 characters. A correct answer to this question should provide a new version of the program running with the respective functionality according to a predefined test case.

Experiment Activities. To get a reasonable control of the experiments, the participants were instructed and trained, the laboratory was prepared to offer the same condition for everyone, the solutions were checked and a questionnaire was applied to acquire qualitative information about their task experience.

Preparatory Training and Session Instruction. The participants of the groups *Simple* and *Enhanced* were trained to use correctly their respective dynamic approach for maintenance. The maintainers were able to use the trace extractor, the Concern Mapper and the other views because of this previous special training session. All maintainers in the same group had the same corresponding training.Before each experiment session, the participants were instructed about tasks that they should perform and about the target system. They received the same printed descriptive material that was explained by the experimenter, which was the same in all experiment sessions. The participants could also warm-up with their corresponding approach.

Infrastructure. The experiment was conducted in a specific laboratory of the Faculty of Computing at Federal University of Uberlândia. The computers were previously configured and tested with the same hardware and software configuration. The access to the Internet was blocked.

Experiment execution. The experiment session was executed in a fixed time. The activities of tasks 1, 2, and 3 were executed in a limit of four hours, including 50 minutes used for instruction and adaptation. The activity of task 4 was executed in a limit of four hours too, but the participants were instructed and warmed-up with the tools in 30 minutes. The instruction time reduction for task 4 was necessary because the effort for this activity was known to be greater than for the others. After finishing the execution of the tasks, the participants showed the solutions to the experimenter, who recorded the data and evaluated the results. After that, all participants answered a questionnaire. The questionnaire of the groups *Simple* and *Enhanced* had some specific questions concerning the use of the approach.

5. Results

In this section, the results directly related to the research questions are presented: the time used by subjects in the execution of tasks and the correctness of their results. Other qualitative results, such as utility of the approach, search space, and difficulty level, are presented in the following

section to provide a deeper understanding of the quantitative results. The results of time and correctness will be presented in two steps: the first will show the global results considering all tasks together and the second will consider each task individually.

5.1. Time to Execute the Tasks

This section shows the results obtained for the dependent variable: *the elapsed time to execute the maintenance tasks*. To produce a global sample with data points from all tasks, the data points were normalized: for each task, the greatest execution time among all groups was normalized to the value *one* (1), and the other execution times within the task received a proportional value R, where $0 \leq R \leq 1$. Because there were four tasks, there should be at least four values *one* (1) for the relative time R. All data points R were in the range $0 \leq R \leq 1$. The normalization procedure enabled the comparison of times between different tasks.

To have more confidence in the results, a statistical test was conducted to assess the support to answer if the task execution time of maintainers using trace information was significantly lower than those not using this kind of information. The samples were tested to know if they follow a normal distribution. The relative time of the *Control* group did not follow a normal distribution, either with Kolmogorov-Smirnov test or with the Shapiro-Wilk test because the p-values were respectively, 0.0033 and 0.0015. So, we chose the non-parametric Mann-Whitney test. Table 2 shows the results of this test. The results of the test Mann-Whitney show that the groups *Simple*, *Enhanced*, and the union of these two groups, *Traces*, have a lower median than the *Control* group. There is no significant difference between the *Simple* and *Enhanced* groups.

Figure 3 shows the mean of relative times to execute all tasks with 95% confidence interval. Although the mean time of the *Control* group was greater than the other groups, the confidence interval does not strictly support that the mean relative time of the *Control* group is greater than the other groups. So, we can see a stronger difference for the median time than for the mean time.

Figure 4 shows the boxplot for the absolute time (in seconds) for the 36 data points (nine in each graph). The boxplot connects the medians in the graph. Except, for the *Localization* graph, the connecting line follows a downward trend. In principle, this result seems surprising because feature

Table 2: Mann-Whitney Tests for Relative Task Times R

	Control x Traces	Control x Simple	Control x Enhanced	Simple x Enhanced
H_0	$\tilde{R_C} = \tilde{R_T}$	$\tilde{R_C} = \tilde{R_S}$	$\tilde{R_C} = \tilde{R_E}$	$\tilde{R_S} = \tilde{R_E}$
H_1	$\tilde{R_C} > \tilde{R_T}$	$\tilde{R_C} > \tilde{R_S}$	$\tilde{R_C} > \tilde{R_E}$	$\tilde{R_S} \neq \tilde{R_E}$
P value	0.0158	0.0302	0.0364	0.7290
Exact or approx. p?	Gauss Appr.	Gauss Appr.	Gauss Appr.	Gauss Appr.
Reject H_0 ($p < 0.05$)?	Yes	Yes	Yes	No
One- or two-tailed P?	One-tailed	One-tailed	One-tailed	Two-tailed
Sum of ranks	286.5, 379.5	183, 117	181.5, 118.5	156.5, 143.5
Mann-Whitney U	79.50	39.00	40.50	65.50

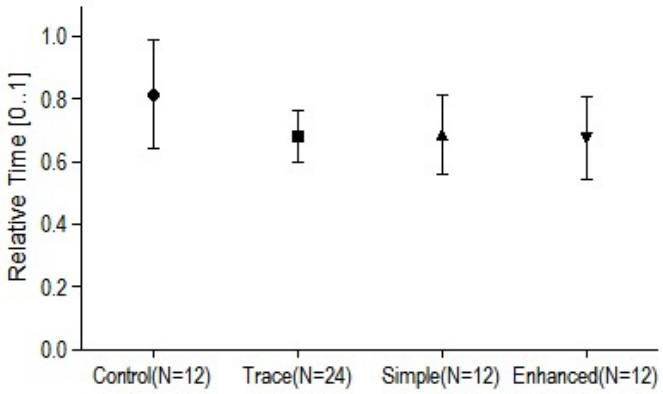

Figure 3: Means of relative time with 95% confidence interval

location approaches did not performed so well in the specific *Localization* task. We will discuss in more detail this result in the next section.

In the first task (*Reuse*), the groups that used the approach presented a lower mean time compared to the group *Control* with a better performance for the *Enhanced* group. In the second task (*Bug Fixing*), the groups that used traces also presented a lower mean time compared to the group *Control*. The mean time of group *Simple* was 34.21% better than group *Control*. Considering the third task (*Location*), the mean time of the groups that used traces was higher compared to the group *Control*. However, considering the variation of the data there is little support to confirm that the group *Control* had significantly better performance, either in Bug Fixing or in Location tasks. In the fourth task (*Improvement*), both groups that used the approach had a lower mean time compared to the group *Control*. Both groups

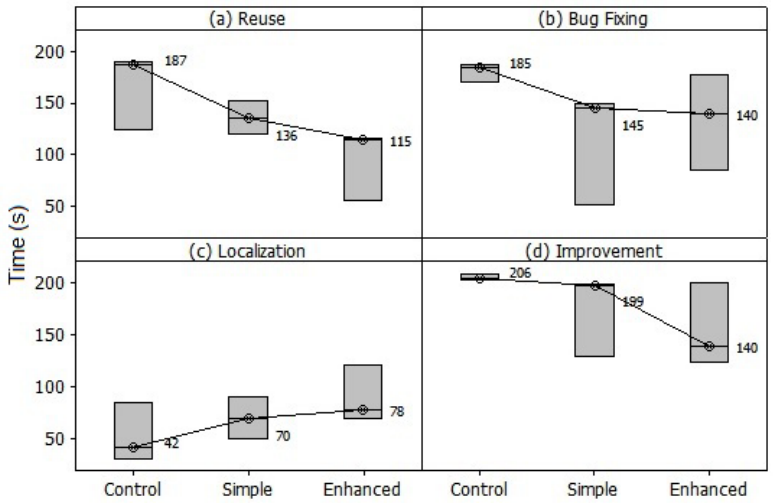

Figure 4: Time results per Task.

that used traces had significantly better performance than group *Control*. It is important to consider that all participants in group *Control* reached the upper-bound time limit allowed for the task without concluding the activity.

5.2. On the Rate of Correct Results

This section shows the results obtained for the dependent variable: *the rate of correct results provided by maintainers for the maintenance activity.*

A statistical test was conducted to know if there is support to answer if the correctness of the tasks outcomes using trace information was significantly greater than those not using this kind of information. The samples were tested to know if they follow a normal distribution. The samples of correctness values of all groups did not follow a normal distribution with the Shapiro-Wilk test because the p-values for groups *Control*, *Traces*, *Simple*, *Enhanced*, were respectively, 0.013, <0.0001, 0.0003 and <0.0001. So, we chose the non-parametric Mann-Whitney test. Table 3 shows the results of this test. The results have shown that the groups *Simple*, *Enhanced*, and *Traces* had significant greater correctness median values than the *Control* group. Similarly to the time evaluation, there is no significant difference between the *Simple* and *Enhanced* groups.

Figure 5 shows the means of correctness values of all tasks with 95% confidence interval. The shown confidence interval clearly supports that the mean

Table 3: Mann-Whitney Tests for Correctness C

	Control x Trace	Control x Simple	Control x Enhanced	Simple x Enhanced
H_0	$\tilde{C_C} = \tilde{C_T}$	$\tilde{C_C} = \tilde{C_S}$	$\tilde{C_C} = \tilde{C_E}$	$\tilde{C_S} = \tilde{C_E}$
H_1	$\tilde{C_C} > \tilde{C_T}$	$\tilde{C_C} > \tilde{C_S}$	$\tilde{C_C} > \tilde{C_E}$	$\tilde{C_S} \neq \tilde{C_E}$
P value	0.0005	0.0034	0.0031	0.9726
Exact or approx. p?	Gauss Appr.	Gauss Appr.	Gauss Appr.	Gauss Appr.
Reject H_0 ($p < 0.05$)?	Yes	Yes	Yes	No
One- or two-tailed p?	One-tailed	One-tailed	One-tailed	Two-tailed
Sum of ranks	130.5, 535.5	104.5, 195.5	104, 196	149, 151
Mann-Whitney U	52,50	26,50	26,00	71,00

of correctness of the *Control* group is lower than other groups, supporting the results of the non-parametric test.

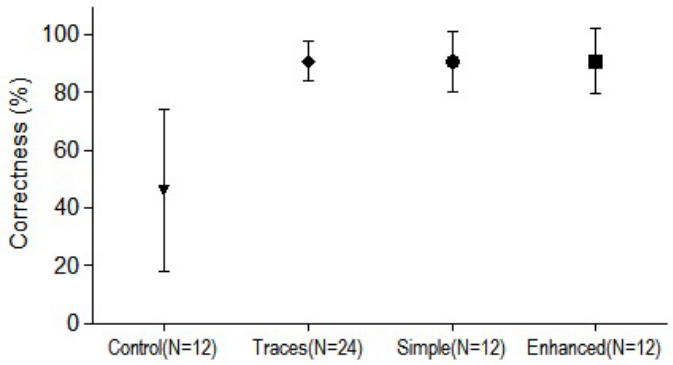

Figure 5: Means of Correctness with 95% confidence interval

The results of the four tasks are presented in boxplot shown in Figure 6. Considering the first task (*Reuse*), in the groups that used traces, all participants have concluded the activity with success while in the group *Control* one out of three participants did not conclude the activity.

Considering the second task (*Bug Fixing*), in the group *Control*, nobody had completed the activity with success while all participants of both groups *Simple* and *Enhanced* had completed with success. This is an interesting result. However, it is important to clarify that we considered that a task was successful when the bug was completely fixed. In the post questionnaire, the participants could show the code elements that could be fixed, when they did not conclude successfully the task. In fact, all participants of group *Control*

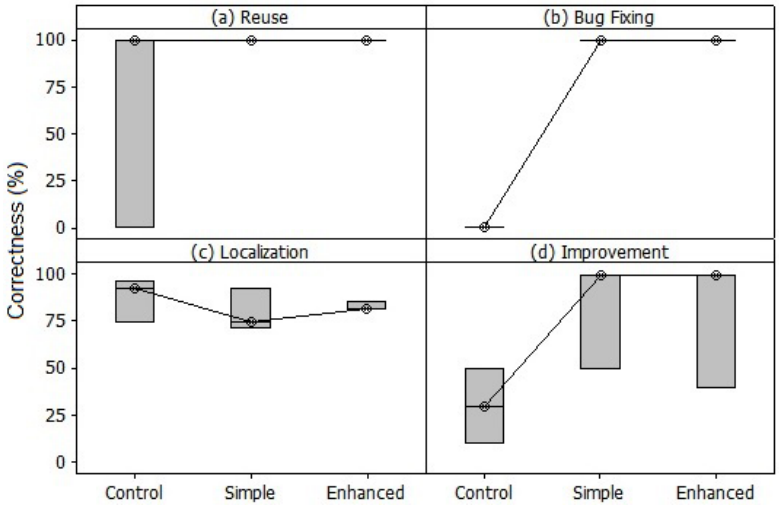

Figure 6: Correctness results per task

answered the post questionnaire indicating correctly which code elements should be fixed, but they were not sure about their answer. It is possible that if they had had more time, they could have completed the task, but this would impact the dependent variable *elapsed time*.

Considering the third task (*Location*), the participants did not have to change the code, but should answer questions about the corresponding feature. The evaluation of the participant answers has shown that *(Q3)* was correctly answered by all participants in the study. All participants that answered that a reorganization in the package structure has occurred to simplify that structure have acquired the maximum grade in this task. Some have provided more details, but we required just to describe the overall implementation change and we accepted more general answers. For *(Q1)* and *(Q2)*, the results did not provide a significant difference among the groups. However, the groups that used the approach had a slightly better result, especially the group *Simple*.

Considering the fourth task (*Improvement*), the mean rate of correct results for groups that used the dynamic approach was better, compared to the group *Control*. On the rate of correct results, we must clarify that nobody in the group *Control* had completely finished the activity. However, they produced some results that received a relative grade that corresponds to the median rate of 30% shown in the plot. In groups *Simple* and *Enhanced*,

66.7% of the participants in each group had completely finished the activity. For those that had not completed, but produced some results, they also received a relative grade with the same criteria of the group *Control*. The group *Simple* had a reasonably significant better result, compared to the group *Control*. The groups *Simple* and *Enhanced* had better results than group *Control*.

5.3. On the Utility of Dynamic Views

This section presents the results obtained for answers on *the utility of the dynamic views perceived by maintainers.* This section and the following provide qualitative analysis to reinforce the understanding of the quantitative results. The results are based on post-questionnaire answers. 95.3% of the participants that used traces answered they were satisfied with it. 100% answered that the approach was useful for the tasks because it has driven the location process. On the utility of the several views, 100% answered that the main feature location view, *Mapping View*, was useful for the tasks. The other three views of the *Enhanced* approach were considered useful by only 25% of the participants in that group.

5.4. On the Difficulty Degree of Maintenance Activities

The difficulty degree of the maintenance activities perceived by participants was answered after performing the tasks. Figure 7 shows the results. The plot in the top is the result of all groups *Control*, and the plot in the bottom shows the result for groups *Simple* and *Enhanced* (*Traces*).

The groups that used the approach (*Traces*) mostly classified the performed activity as *median* or *easy*. Only 17% of the participants of this group classified the activity as *hard* for tasks 2, 3 and 4. For the group *Control*, except in task 1, nobody has classified the performed activity as *easy*. Moreover, in tasks 2 and 4, most of participants classified it as *hard*, 100% and 67%, respectively. This is an interesting finding because in tasks 2 and 4, the group *Control* had worse performance in correctness, suggesting that their poor performance is related to the difficult nature of the task and not to the insufficient time.

5.5. On the Search Space

We analyzed the impact of the initial search space reduction provided by dynamic approaches. Figure 8 presents the mean number of methods in the views of the groups *Simple* and *Enhanced*, which the subjects have used as

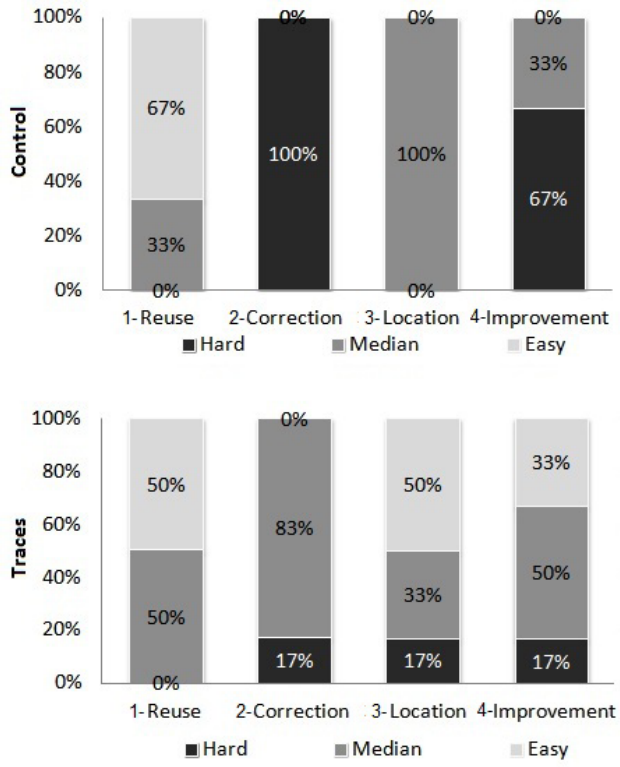

Figure 7: Difficulty level of activities as perceived by participants.

starting point. Because there is no previous view prepared to group *Control*, we will not consider the percentage of the search space that they had to search.

For example, in task 1 (*Reuse*), 45.83% of all methods were shown for the group *Simple* and 50.93% for the group *Enhanced*. This task had the lowest search space reduction. This can be explained because the target system is small and the defined scenario executed nearly half of the existent methods.

Tasks *2 - Bug Fixing* and *4 - Improvement* had higher search space reduction. It is interesting to see that the elapsed time and the rate of correction of these tasks were correlated with this observation. The search space reduction in the mean number of classes of task *3 - Location* was not significant as 2 and 4 and the elapsed time was slightly worse for the participants in groups *Simple* and *Enhanced*.

The quality of the search space concerning false negatives were analyzed to understand if they had interferred in the effort and correctness of results for

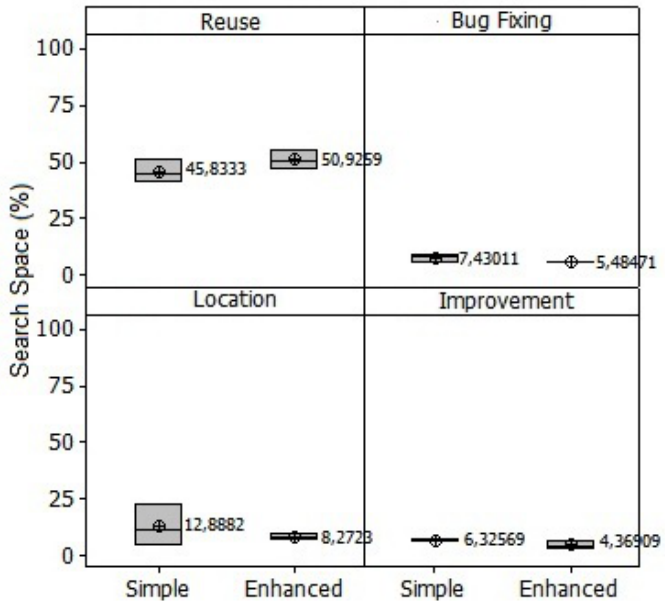

Figure 8: Search space (mean number of methods)

those maintainers that use the approach. In Figure 9, we show the number of false negatives classes, which were calculated by the experimenter. The task *Location* presented the higher rate of false negatives, either in the *Simple* or in the *Enhanced* approach. The tasks *Bug Fixing* and *Improvement* did not present false negatives, indicating that the subjects produced representative execution scenarios that captured all relevant method calls. This result can also be related to the fact that in these tasks, the *Traces* group had a better performance compared to the *Control* group.

6. Discussion

6.1. On the Time to Execute the Tasks

Our hypothesis was that when the proposed approach, either simple or enhanced, is applied to the maintenance of unknown systems, feature location activities would be faster because they are partially automated by the use of execution trace information and, consequently, the response to a task that requires the feature location would also be faster. Feature location using dynamic analysis would reduce and organize the search space for maintainers, especially for those features that are scattered across the code.

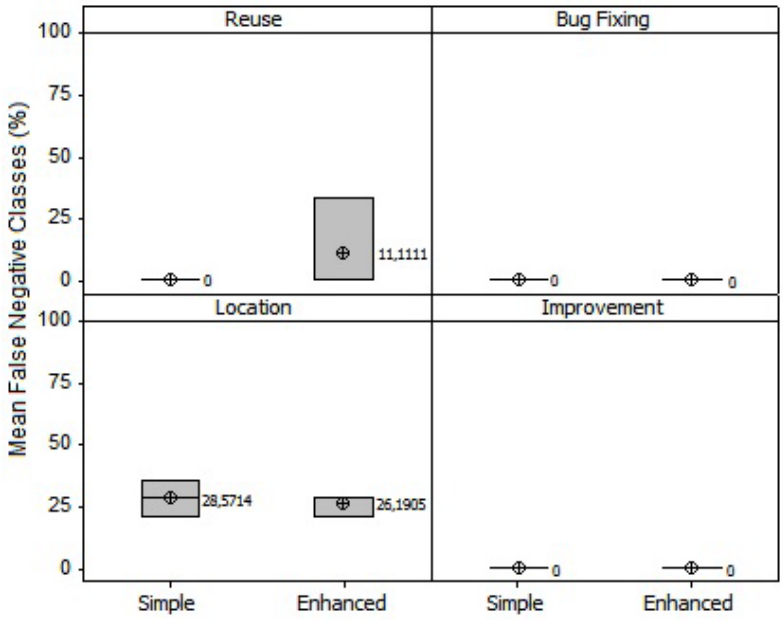

Figure 9: False negative classes.

On the dependent variable related to RQ1, the elapsed time to execute the maintenance task has shown a significant gain for the execution traces approaches concerning the median time, but concerning the mean time the results were not conclusive, denoting an important variation between subjects.

There is an interesting fact to consider in *Location* task. Considering all groups, the mean time of this task is systematically lower than that of the others, suggesting that this task is the easiest one, at least concerning effort. Moreover, another point that we must consider is that, differently from other studies [26, 25], the time of the groups using execution traces also include the time to plan and extract the execution traces. So, when we have a small task time, the influence of the time to plan and to extract the traces plays a major role, and would hinder the performance of the *Trace* groups. For medium-sized or large-sized tasks, the benefits of trace-based approaches is likely to make up for the spent time in executing the necessary steps.

The *Location* task was to identify classes that implement the selected features and to describe changes that occurred from one version to another. Unlike the other tasks, this task did not require changes to source code. The

mean time for groups *Simple* and *Enhanced* were 70 and 90 minutes, respectively, including the time to plan the execution scenarios, extract the traces and analyze the generated views. The participants in the group *Control* initiated the activity directly with code analysis. An important fact is that *JHotDraw* has a good modularization for the analyzed features *Undo* and *Redo*, because they are explicitly modularized into specific packages, which facilitated feature location. Also, the search (on Eclipse) for terms `undo` and `redo` returned good results, since 92.85% of the classes that implement these features have these terms in the suffix or prefix of their names.

Considering the tasks *Bug Fixing* and *Location*, the results were different, despite the functionality and target system being the same in both tasks. Feature location in the *Location* task was easier than *Bug Fixing* for the *Control* group because of the adequate modularization and naming of classes present in the *Location* task. However, this point seemed not to help in the *Bug Fixing* task because the classes and methods that should be corrected did not have names directly related to the feature and were not part of its representative package. The subjects that used dynamic analysis approach could find straightforwardly the problematic methods using trace differentiation.

Concerning RQ1, we can answer that, in most of the analyzed situations, the use of the approach (*Traces* group = *Simple* + *Enhanced*) significantly reduces the median maintenance effort. The approach did not reduce the effort in cases where the feature location is not a challenging task for maintainers. These cases occur when the feature is well-modularized in the implementation and has intuitive code elements names, facilitating textual search. We believe that we have used a more realistic approach in our experiment, compared to other works in the literature [25, 26], when we included the time to plan and extract the traces in the data used in our analyses, despite producing a less significative gain, because there was no significative difference in the mean time of the groups. This observation that trace-based approaches had major impact in the median than in the mean could indicate that the use of trace-based approaches produce better result for the individuals than considering the whole maintenance time in organization.

Considering the RQ3 that compares the *Simple* and the *Enhanced* approaches, no one outperformed the other because one approach had better performance in two tasks and the other approach had better performance in the other two tasks. Moreover, the medians and means shown in the previous section had no significant difference.

In our experimental setting, we defined an upper-bound limit for executing the tasks. Although, this restriction was not strictly necessarily, an unbound session time would impose practical difficulties when running the sessions. So, we also need to carefully interpret the task execution time when the maintainer reached that upper-bound limit. So, when the task could not be completed, which occurred for the *Control* group, we must carefully compare the times because the probable time to complete the task would be longer. However, as we mentioned in Section 5.2, during the correction of the task results, the experimenter could analyse the partial work produced by maintainers and verify that for some of those tasks that could not be completed, they had almost been completed. Moreover, most of the uncompleted tasks occurred in the control group in task 4, so having a larger accounted time for this group would not modify our conclusions, but indeed would reinforce the conclusion.

Another point concerns the absolute time to complete the tasks. Other experimental studies on the effectiveness of dynamic information for comprehension [25, 26] have only accounted for the comprehension time, not including the time to generate the dynamic information views, neither the time to complete a maintenance task. In Quante's [25] work, the worst time to answer the considered question was less than 25 minutes and in Cornelissen's work [26] the limit time was 90 minutes to complete all eight comprehension tasks. These absolute experimental times are much lower than ours because they included only comprehension activities. Moreover, our experiment is different from others [25, 26], where the independent variable is the availability of the view, because they consider neither the time to obtain the view nor the inherent variability of the human subjects obtaining those views. So, our study would be much susceptible to individual variations. In fact, we have adopted a more realistic approach concerning the independent variable and the results concerning alternative hypothesis can be considered even stronger.

6.2. On the Rate of Correct Results

Our hypothesis was that the provided views induce maintainers to find the appropriate classes and methods that should be considered in the maintenance tasks and, consequently, they would produce all the appropriate modifications while maintainers using traditional approach may not consider all necessary modifications if they miss related code elements.

The results on the dependent variable *correctness* clearly answer RQ2. Considering all tasks together the groups *Simple* and *Enhanced* provided bet-

ter correctness results than the group *Control*, confirmed by Mann-Whitney significance test and confidence intervals of the means. Moreover, in the tasks *Bug Fixing* and *Improvement*, the correctness was even better for the *Traces* group.

Similarly to RQ1, the approach did not provide better results in cases where the feature location is not a challenging task for maintainers, which can occur in situations where the maintenance tasks are easy or the features are well-modularized in the implementation.

We suggest that the better correctness results of the *Traces* groups was influenced by the inherent good recall of dynamic analysis, which have retrieved the classes and methods involved in each executed feature. Because the maintainers were driven to investigate those classes, they did not incur in errors related to missing some important class/method of the analysis. These results are coherent with those observed in [26].

Concerning the task 4 that could not be completed by none of the participants in the *Control* group and by 33% of the *Traces* groups, we observed that the main difficulty of maintainers to conclude the task was concerned with locating and understanding of where should the modifications take place, in other words, a feature location problem. This information was obtained from a qualitative analysis of the post-questionnaires answered by the subjects. We can clearly see a better accuracy of the group *Traces* in the task 4, which can be explained by the support offered by execution trace in the feature location problem.

Considering the RQ4, similarly to the time results, the groups *Simple* and *Enhanced* did not present significatively different correctness performance.

6.3. On the Utility of Dynamic Views

The results on *utility of the view* help to answer if information provided by the dynamic views were considered useful during the maintenance tasks. We can answer that the *Mapping* view was effectively considered useful, but there is no support to say the other views were effectively useful. The subjects answered that the mapping has driven the comprehension process because they intuitively studied the elements related to the feature implementation.

The views *Class Classification*, *Method Call*, and *Mapping with Classification* were considered useful only by 25% of the participants in group *Enhanced*. We suggest that either this information did not have an important impact to contribute with the task execution or the maintainers did not use it because they did not grasp entirely the purpose of that information.

It is important to mention that the group *Enhanced* also had access to the *Mapping* view, which already had useful available information that would suffice for them.

Another point is that the functionalities of the IDE Eclipse were useful to refine and complement the information obtained by the dynamic view. Moreover, it was observed that only the *Mapping view* is not sufficient to complete the activities, so the navigation in the source code provided by the IDE static analyses was considered essential.

6.4. On the Difficulty Degree of Maintenance Activities

The results on the difficulty degree perceived by the users have clearly shown that the users in the group *Control* encountered more difficulty than the other groups, particularly in tasks *Bug Fixing* and *Improvement*. This result is important even to validate the representativeness of the proposed tasks in this study. If the perceived difficulty were similar for every group, maybe the tasks were all very difficult or very easy.

Another point is that the task *Reuse* seems to be the easiest. One probable reason would be that the target system (*Board Games*) was the smallest, facilitating the analysis of whole system. Nonetheless, this task still had provided significant information to the analysis of the execution time.

Other interesting finding is that results on the difficulty perceived by the maintainers are consistent with the results on the correctness variable.

6.5. On the Search Space

6.5.1. Space Reduction

Our initial hypothesis was that dynamic analysis information drives the comprehension process because there is some search-space reduction. As we can observe in Figure 8, in the four tasks the search-space considering the whole system has been reduced. In fact, in the tasks *Reuse* and *Location*, the reduction was not large as in the other tasks. Moreover, this smaller reduction in the *Reuse* task would be expected because the system is the smaller and proportionally a larger part of the system was necessary for the task completion. In the case of *Location*, this smaller reduction also helps to explain the weaker time performance of the *Traces* group in this task. The other two tasks *Bug Fixing* and *Improvement*, which had greater reduction, also had compatible performance.

In the case of the task *Location*, the *Control* group used the package names as an effective hint for search-space reduction, because the features to

be analyzed were well located in specific packages.This situation can explain the good performance of the *Control* group in this task.

Finally, we suggest that if the space reduction provided by dynamic analysis is significant and the feature is not well-modularized in the system, then that space reduction has a significant impact.

6.5.2. False Negatives

The false positives (code elements that should not be in the views but are) and false negatives (code elements that should be in the views but are not) may occur in a dynamic analysis approach when scenarios are poorly defined or invisible events to users may be triggered during the execution. False positive elements usually can be recognized by the users, but false negative elements cannot. We tried to investigate if the false negative elements interfere in the performance of maintainers. The false negatives were measured in the post-questionnaire.

Interestingly, in the tasks *Bug Fixing* and *Improvement*, there was no occurrence of false negatives, which helps to explain a better time and correctness performance of the *Traces* group. In the tasks *Reuse* and *Location*, even with some rate of false negatives, it seems that it did not affect significatively the correctness but impacted the execution time, which can be consistently explained by the static analyses that follow the provided true positive elements. Another point is that, as we could expect, the number of false negatives were highly sensitive to the quality of execution scenarios planned by maintainer.

7. Threats to Validity

Even with the careful planning and formal procedures applied during the execution of the experiments, some threats should be considered in the evaluation of the results validity.

7.1. Internal Validity

Some threats are important to consider when analyzing the answers for our research questions because they could interfere in the time and/or correctness of the tasks.

- Individual differences among the participants. The actions to minimize these differences were the analysis and classification of the capability of

each participant to generate fairly balanced groups and also a random selection of the group in which the participant should be included.

- The participants' understanding about the dynamic approach may vary during the training. We tried to minimize this using the same printed material for everybody and practicing with tools and approach before the experiment session.

- Subjectivity degree in the definition of which code elements are relevant for an activity. The result of some maintenance activities does not have a unique and correct answer. Developers may propose different, but correct results for the same activity. Our criteria for changes in source code was that the changed program should pass in a predefined test case.

- Inexperienced participants concerning the proposed approach. The comparison of the performance of the *Control* group, which followed a traditional Eclipse-based approach with the performance of groups that do not have the long term experience of using a new approach may not be totally fair. We can consider that the results could be even better for *Traces* groups if these groups had more experience with the tools.

- Human-related issues such as, the participant enthusiasm in the experiment day, the participant expectation on the experiment.

7.2. External Validity

Some other factors limit the generalization of the results like:

- The number of participants in each group for an activity. As in any statistical study, the larger and more representative is the sample, the stronger are the evidences. In fact, if the results were only analyzed separately for each activity, then this threat would be more sensitive. However, the results were analyzed considering the set of four tasks with a normalized metric. This setting accounts for 12 observations for each group and 36 observations for the whole study. Moreover, the impact of the use of trace information (*Simple* or *Enhanced*) were analyzed with 24 observations. These numbers are significant considering related work. Cornelissen et al. [11] showed that only six out of

the 114 selected papers in the above area were published using empirical evalution based on human subjects. From these six papers, only Quante's work has used more than nine human subjects in the experiment [25]. In Quante's work, 25 students have participated in the main experiment. In our work, 27 subjects have participated in the main experiment.

- The representativeness of target systems. Although the system used in the first task is a home-made system, JHotdraw and ArgoUML are well-known systems widely used in other studies. The number and domain of the used system limit the generalization of our results.

- The selected maintenance tasks. Although the maintenance tasks were chosen to be quite different from each other, it is not possible to generalize for any possible maintenance task. Nonetheless, our results have shown a reasonable level of variability in the tasks results that enabled the qualitative discussion of important points.

- Only the Java programming language and Eclipse development environment were considered in this study. Some of our results would be different if other languages and environments were used. For example, different languages may support different types of dependencies; different development environments may summarize and present code differently.

7.3. Construct Validity

Concerning our measurement framework there some issues to be pointed:

- The experimenter also has defined the dynamic approach being evaluated. However, the experimenter had the care to adopt an impartial behavior during the experiment, being as formal as possible during the execution of the planned experimental activities.

- The use of a time frame to execute the maintenance tasks. Because we are measuring the time elapsed during the execution of a maintenance task, maybe the measured time can be greater than it could be in reality, because maintainers could manage to use all the available timeframe. To minimize this effect, the subjects were instructed to use only the necessary time to complete the task and not to perform unnecessary activities during task execution. Moreover, the time frame

implies in an upper-bound time limit for the execution of the task. When maintainers reached this time, we have to consider that the real execution time could be greater than what was measured.

- The use of an internally adapted execution trace approach. Although, the tool suite has been tested and verified, it still could remain some undetected bug just as occurs with any software. There is no widely recognized and adopted tool support for this kind of approach, so any other adopted solution would incur a similar threat. We tried to minimized this issue using well-known tools and concepts during the definition of our the studied approaches.

8. Conclusions

This paper presented a controlled experiment to evaluate an approach to help maintenance tasks using dynamic and static analysis for the feature location problem. An experiment with real maintenance tasks performed by human subjects was conducted aiming at assessing the impact of this approach on the subjects' performance during maintenance activities. The controlled experiment was conducted with three distinct groups (*Control*, *Simple* approach and *Enhanced* approach), adding up 36 individual observations.

From the study, we have concluded that:

- The use of either *Simple* or *Enhanced* approaches led to a reduction in the time to complete a task compared to a traditional approach confirmed by a Mann-Whitney test for the median times, but not confirmed with a confidence interval analysis for the mean times. This observation on the major impact in the median than in the mean could indicate that the use of trace-based approaches produce better result for the individuals than when considering the whole maintenance time in organization. Moreover, this reduction occurs more clearly in situations where the analyzed features are not completely modularized in specific units and do not have intuitive names for code elements. Even in situations where the system is well packaged and named, dynamic analysis is useful when the desired target is not located in the representative package of the feature.

- The rate of correct results achieved by participants using either *Simple* or *Enhanced* approach is better than traditional approach supported by statistical tests.

- In tasks where the search-space reduction is higher or the activity is considered harder, the use of *Simple* or *Enhanced* approach produced better results than the use of the *Control* approach.

- The low difficulty perceived by the participants of the *Simple* or *Enhanced* groups are related with their better performance, indicating that even if the task is difficult, the approach facilitates maintainers' work.

- The results suggested that there is no quantitative evidence about the difference of the *Simple* and *Enhanced* approaches. Moreover, the participants qualitatively suggest that the *Mapping* view is the most important, reinforcing this lack of difference.

Concerning our hypothesis, this study has shown that the use of a dynamic approach for feature location does not necessarily reduce the effort and enhances the correctness of results in all cases. The approach was best suited when the maintainers perceived the maintenance tasks as being difficult. A factor that can help maintainers to identify if the maintenance task will be hard or not is feature scattering across the code. The results also indicated that the usefulness of the approach for feature location probably occurs because of an organized reduction of the initial search space. Furthermore, participants who used either the simple or the enhanced approach indicated a decreased level of difficulty in the proposed activities, when compared to group Control, for tasks with the scattering problem of source code. The questionnaire results also reported that the approach should be seamlessly integrated into the IDE and this is part of the future work that has to be done.

Concerning the possibility of wide adoption of the use of execution traces in daily practice of maintainers, our results suggest that the adoption would require an additional preliminary effort to use this kind of approach. In situations where the system is well-known, or the activity seems to be easy, or the features are well-modularized, then this effort seems not to provide significative gain. If the previous situations do not appear in the software

maintenance context then the use of execution trace information proved to be effective.

Some future work still remains to be done. Since our results indicated that trace-based feature location produces better results when maintenance tasks are considered harder or the selected feature is scattered across the code, an useful approach should help developers to predict in which maintenance tasks, the execution trace information would provide better performance results. Our experimental setting did not consider an industrial-scale environment, which could reduce several threats identified in this work. Although, it is a challenging task conducting these kind of experiment in such environment, the results that were shown in this work indicates a feasible opportunity for the adoption of trace-based feature location techniques in an industrial setting.

Acknowlegments

We are grateful for the valuable comments of the anonymous referees that helped to improve the quality of this paper.

References

[1] D. Batory, Feature-oriented programming and the AHEAD tool suite, in: Proc. of the Intl. Conf. on Software Engineering, 2004, pp. 702–703.

[2] D. Batory, J. Sarvela, A. Rauschmayer, Scaling step-wise refinement, Software Engineering, IEEE Transactions on 30 (6) (2004) 355 – 371.

[3] S. Trujillo, D. Batory, O. Diaz, Feature refactoring a multi-representation program into a product line, in: Proceedings of the 5th international conference on Generative programming and component engineering, GPCE '06, ACM, 2006, pp. 191–200.

[4] N. Wilde, M. C. Scully, Software Reconnaissance: mapping program features to code, Journal of Software Maintenance 7 (1995) 49–62.

[5] T. Eisenbarth, R. Koschke, D. Simon, Locating features in source code, IEEE Transactions on Software Engineering 29 (3) (2003) 210–224.

[6] G. Antoniol, Y.-G. Guéhéneuc, Feature identification: An epidemiological metaphor, IEEE Transactions on Software Engineering 32 (9) (2006) 627–641.

[7] S. Apel, T. Leich, M. Rosenmller, G. Saake, FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming, in: R. Glck, M. Lowry (Eds.), Generative Programming and Component Engineering, Vol. 3676 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 125–140.

[8] B. Ramesh, Factors influencing requirements traceability practice, Communications of the ACM 41 (1998) 37–44.

[9] D. Cuddeback, A. Dekhtyar, J. Hayes, Automated requirements traceability: The study of human analysts, in: 18th IEEE Intl. Requirements Engineering Conf., 2010, pp. 231 –240.

[10] A. D. Eisenberg, K. D. Volder, Dynamic feature traces: finding features in unfamiliar code, in: Proc. of the Intl. Conf. on Software Maintenance, 2005, pp. 337–346.

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

[12] N. Wilde, C. Casey, Early field experience with the Software Reconnaissance technique for program comprehension, in: Proc. Int. Conf. on Software Maintenance (ICSM), IEEE C.S., 1996, pp. 312–318.

[13] N. Wilde, M. Buckellew, H. Page, V. Rajlich, A case study of feature location in unstructured legacy Fortran code, in: Proc. 5th European Conf. on Software Maintenance and Reengineering (CSMR), IEEE C.S., 2001, pp. 68–76.

[14] N. Wilde, M. Buckellew, H. Page, V. Rajlich, L. Pounds, A comparison of methods for locating features in legacy software, J. Syst. Software 65 (2) (2003) 105–114.

[15] T. Eisenbarth, R. Koschke, D. Simon, Incremental location of combined features for large-scale programs, in: Proc. of the Intl. Conf. on Software Maintenance, 2002, pp. 273 – 282.

[16] O. Greevy, S. Ducasse, T. Gîrba, Analyzing software evolution through feature views, Journal of Software Maintenance and Evolution: Research and Practice 18 (6) (2006) 425–456.

[17] S. Simmons, D. Edwards, N. Wilde, J. Homan, M. Groble, Industrial tools for the feature location problem: an exploratory study: Practice articles, Journal of Software Maintenance and Evolution 18 (2006) 457–474.

[18] K. Lukoit, N. Wilde, S. Stowell, T. Hennessey, TraceGraph: Immediate visual location of software features, in: Proc. 16nd Int. Conf. on Software Maintenance (ICSM), IEEE C.S., 2000, pp. 33–39.

[19] M. P. Robillard, F. Weigand-Warr, ConcernMapper: simple view-based separation of scattered concerns, in: Proc. of the OOPSLA Workshop on Eclipse technology eXchange, ACM, 2005, pp. 65–69.

[20] M. P. Robillard, G. C. Murphy, Representing concerns in source code, ACM Trans. Softw. Eng. Methodol. 16 (1) (2007) 3:1–38.

[21] M. Revelle, D. Poshyvanyk, An exploratory study on assessing feature location techniques, in: Proc. of ICPC'2009, 2009, pp. 218–222.

[22] J. Wang, X. Peng, Z. Xing, W. Zhao, An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions, in: 27th IEEE Intl. Conf. on Software Maintenance (ICSM'2011), 2011, pp. 213 –222.

[23] J. Quante, R. Koschke, Dynamic object process graphs, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, 2006, pp. 10 pp. –90.

[24] J. Quante, Online construction of dynamic object process graphs, in: Proc. of the 11th European Conference on Software Maintenance and Reengineering, 2007, pp. 113 –122.

[25] J. Quante, Do dynamic object process graphs support program understanding? – a controlled experiment, in: Proc. the Intl. Conf. on Program Comprehension, 2008, pp. 73–82.

[26] B. Cornelissen, A. Zaidman, A. van Deursen, A controlled experiment for program comprehension through trace visualization, IEEE Transactions on Software Engineering, 37 (3) (2011) 341 –355.

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

[28] M. P. Robillard, G. C. Murphy, Representing concerns in source code, ACM Trans. Softw. Eng. Methodol. 16 (1).

[29] M. P. Robillard, Topology analysis of software dependencies, ACM Trans. Softw. Eng. Methodol. 17 (4) (2008) 18:1–18:36.

[30] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, An information retrieval approach to concept location in source code, in: Proc. of the 11th Working Conference on Reverse Engineering, 2004, pp. 214 – 223.

[31] W. Zhao, L. Zhang, Y. Liu, J. Sun, F. Yang, Sniafl: towards a static non-interactive approach to feature location, in: Proc. of the 26th International Conference on Software Engineering, 2004, pp. 293 – 303.

[32] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process - published online.

[33] V. Sobreira, M. A. Maia, A visual trace analysis tool for understanding feature scattering, in: 15th Working Conf. on Reverse Engineering WCRE'08., 2008, pp. 337 –338.

[34] G. Antoniol, Y.-G. Gueheneuc, Feature identification: a novel approach and a case study, in: Proc. of the Intl.Conf.on Software Maintenance, 2005, pp. 357 – 366.

[35] T. Eisenbarth, R. Koschke, D. Simon, Aiding program comprehension by static and dynamic feature analysis, in: Proc. of the Intl. Conf. on Software Maintenance, 2001, pp. 602 –611.

[36] N. Juristo, A. M. Moreno, Basics of Software Engineering Experimentation, Springer, 2001.

[37] A. De Lucia, A. Fasolino, M. Munro, Understanding function behaviors through program slicing, in: Program Comprehension, 1996, Proceedings., Fourth Workshop on, 1996, pp. 9 –18.

[38] A. von Mayrhauser, A. Vans, From program comprehension to tool requirements for an industrial environment, in: Program Comprehension, 1993. Proceedings., IEEE Second Workshop on, 1993, pp. 78 –86.

[39] B. Lientz, E. Swanson, Software Maintenance Management, Addison-Wesley, Reading, MA, 1980.