On the Proactive and Interactive Visualization for Feature Evolution Comprehension: An Industrial Investigation

Renato Novais¹,², Camila Nunes³, Caio Lima¹, Elder Cirilo³, Francisco Dantas³, Alessandro Garcia³, Manoel Mendonça¹

¹Software Engineering Lab, Computer Science Department, Federal University of Bahia, Bahia, Brazil
²Information Technology Department, Federal Institute of Bahia, Campus Santo Amaro, Bahia, Brazil
³Opus Research Group, Software Engineering Lab, Informatics Department - PUC-Rio, Rio de Janeiro, Brazil

{renatoln, caiolima, mgmendonca}@dcc.ufba.br, {cnunes, ereioli, fneto, afgarcia}@inf.puc-rio.br

Abstract - Program comprehension is a key activity through maintenance and evolution of large-scale software systems. The understanding of a program often requires the evolution analysis of individual functionalities, so-called features. The comprehension of evolving features is not trivial as their implementations are often tangled and scattered through many modules. Even worse, existing techniques are limited in providing developers with direct means for visualizing the evolution of features’ code. This work presents a proactive and interactive visualization strategy to enable feature evolution analysis. It proactively identifies code elements of evolving features and provides multiple views to present their structure under different perspectives. The novel visualization strategy was compared to a lightweight visualization strategy based on a tree-structure. We ran a controlled experiment with industry developers, who performed feature evolution comprehension tasks on an industrial-strength software. The results showed that the use of the proposed strategy presented significant gains in terms of correctness and execution time for feature evolution comprehension tasks.

Keywords- program comprehension; feature evolution, software visualization; experimental evaluation

I. INTRODUCTION

It is widely recognized that program comprehension is a key aspect in software maintenance and evolution [1, 2]. The comprehensibility of a program largely depends on the representation of its functionalities into logical units called features. A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system [3]. The maintenance and evolution of industrial software projects often requires the analysis of how one or more features evolved in the code history [1, 2]. However, the comprehension of feature evolution is far from trivial as it relies on: (i) selecting versions and features of interest of a given application, (ii) understanding the implementation elements (e.g. methods and attributes) that realize a feature across multiple versions, and (iii) identifying feature dependencies that emerged during the application evolution.

Comprehension of features is primarily based on the identification of their implementation elements in the source code. As a result, tools have emerged to allow the mapping of the implementation elements realizing each feature. Some well-known examples of tools are: (i) FEAT [4] and ConcernMapper [5, 6] that provide a lightweight visualization based on graph-based representation, (ii) SoQueT [7] that supports sort-specific queries, and (iii) CIDE [8] that is based on an AST generic representation. Unfortunately, they provide from none to very limited support for feature evolution analysis.

There are also tools and approaches that aim to provide a graphical representation of the software evolution history, such as software visualization tools [9, 10, 11, 12, 13, 14] and diff tools [15, 16, 17]. Software visualization tools – such as Evolution Radar [9], Evolution Matrix [13], CodeCity [11], Moose [10], and CVSscan [12] – provide different views of the program modules’ history. Those tools usually aim at supporting software evolution comprehension on the basis of a single visualization paradigm. Diff tools aim at detecting the differences among the versions of an application in terms of added, modified and removed implementation elements. They have been around for quite some time and are usually supported by configuration management systems, such as Subversion (SVN) [15] and CVS [16]. However, current diff and visualization tools do not have support for feature evolution analysis. They are only able to capture and represent the evolution of the application modules (i.e. packages, classes, and methods).

To tackle this problem, we present and assess a proactive and interactive visualization strategy to assist developers in feature evolution comprehension. In order to realize this strategy, we combined a set of heuristics [26, 27] for proactively detecting feature evolution and extended our interactive differential visualization [18]. A tool, named SourceMiner Evolution (SME) [18] supports this strategy, which currently produces three different views of the analysed versions. We conducted a controlled experiment to evaluate the proposed visualization strategy. The experiment compares it against the strategy for feature comprehension supported by the ConcernMapper (CM) tool. CM provides a promising visualization strategy to support feature comprehension and is supported by a stable plug-in with extensive documentation. The strategies are compared with respect to execution time and correctness. These measures have also been used by other studies in the context of software visualization and comprehension [11, 19, 24].
experiment involved participants who are industry developers with different levels of experience. They had to perform program comprehension tasks on the top of five versions of a large-scale software project. The results showed that the proactive and interactive visualization strategy had a superior performance with respect to the correctness and execution time.

This paper is organized as follows. Section II discusses the challenges in feature comprehension. Section III presents the visualization strategy overview. Section IV describes the experiment. Section V through VII discusses the experiment’s results, lessons learned, and validity threats, respectively. Finally, Section VIII presents the conclusions.

II. CHALLENGES IN FEATURE EVOLUTION COMPREHENSION

In industrial scenarios, software developers need to continuously deal with the complexity of features in the source code. Feature comprehension is therefore a very important activity in large software systems. In order to comprehend features it is essential to identify and understand their implementation elements. In this manner, many tools and techniques have been proposed. They usually aim at supporting developers to identify and understand features’ implementation elements. They can be classified into three main categories: static, dynamic, and hybrid techniques. Developers use static analysis tools to interactively (manually) assign each feature to the implementation elements [4, 20, 21, 22]. Some examples of such tools are: ConcernMapper [5, 6], JRipples [23], and FEAT [4]. On the other hand, dynamic techniques try to automatically identify parts of the source code implementing a feature. Examples of such tools are EXTRAVIS [24] and FLAT [25]. There are also hybrid approaches that combine static and dynamic analysis techniques in different ways [21, 22].

However, such tools only take into consideration the analysis of feature realizations within a single application version. Therefore, they do not provide direct means for analysing feature evolution. With software systems being constantly evolved, developers spend a considerable part of their time in understanding how features evolved over time [1, 2]. Current tools for feature analysis make it difficult to discover, for instance, what implementation elements have been added or modified for realizing a given feature through the application evolution history.

Software visualization has successfully been used in several software comprehension activities [8, 9, 11]. Then, one may also expect this technique will be effective to support feature evolution comprehension. Software visualization researchers have proposed several approaches and tools to support the understanding of software evolution [9, 10, 11, 12, 13, 14, 19]. However, current visualization tools do not provide support for feature evolution analysis.

The proposed strategy is therefore a further development of the discussions aforementioned as we provide support for feature evolution comprehension using three different graphical representations. Furthermore, there is a complete lack of empirical studies in the literature evaluating the benefits of different visualization strategies in the context of feature evolution comprehension tasks.

III. PROPOSED VISUALIZATION STRATEGY OVERVIEW

This section presents the proposed proactive and interactive visualization to assist developers to comprehend the feature evolution. The proactive strategy utilizes a set of heuristics that are able to analyse the application evolution and to suggest what implementation elements are realizing each feature (Section III.A). The tool called SourcerMiner Evolution (SME) produces interactive visualizations that allow developers to select and compare any two versions of an application (Section III.B). The differences among any selected versions are exhibited through colors that highlight the removed, added and transferred features realized by each implementation elements. SME implements a software evolution visualization approach – SEVA [18] – that was extended to support feature evolution analysis. The result of this extension is a highly interactive visualization that helps developers in reengineering tasks by allowing: (i) to understand the evolution of features in terms of their implementation elements; (ii) to compare any versions of an evolving application; and (iii) the interaction of the developers with the views. SME uses multi-view visualization to portray feature evolution over several versions of the application. These views are produced directly from source code and feature mappings. Figure 1 illustrates the proposed visualization strategy.

A. Proactive Phase

The representation of the features in the SME tool is achieved by means of feature mappings. The feature mapping consists of assigning implementation elements in the source code to features [5, 21]. In this current version, SME uses the same feature representation provided by the CM tool [5, 6]. This means that the features contain names and the implementation elements that realize them. SME loads the features in order to visually represent their implementation elements in the views. It is required in the SME tool that all the feature mappings of each version of an evolving application are available. However, the process for producing the mappings of all the application versions is costly and time-consuming. As SME requires the feature mappings of each application version, it uses a set of heuristics to generate the mappings for all the versions (Vi in Figure 1). This process is named feature mapping expansion (Vi.xml in Figure 1) and is based on a seed mapping (Seed.xml in Figure 1) [26, 27].

The seed mapping contains an initial set of implementation elements realizing the analysed features. The heuristics analyse the seed mapping and all the versions’ source code of the application during the mapping expansion process. There are a set of five heuristics which are:
detecting omitted feature partitions, detecting communicative features, detecting code clone, detecting interfaces and super-classes, and detecting omitted attributes. The heuristics are decoupled from the SME tool and thus they can be used for any mapping and/or visualization tool. The heuristics have presented a good accuracy in the expanded mappings regarding precision (about 95%) and coverage ranging from 66% to 100% [27]. It is not the focus on this paper to analyse the effectiveness of the mapping expansion heuristics. More details about the rationale and strategy of the heuristics and their analyses can be found in [26, 27].

B. Interactive Phase

SME is an integrated, interactive and coordinated multiple views environment. It uses three views to address the feature evolution comprehension. Each view provides means for analysing the feature evolution under different perspectives: structure, inheritance and dependency. The first view is based on a Treemap, which is a hierarchical 2D visualization that maps a tree-structure into a set of nested rectangles [28]. This view addresses the structural perspective as it reveals how the software is organized into packages, classes and methods. The second one is the Polymetric view [14], which addresses the inheritance perspective. This view shows which classes extend others or implement certain interfaces. The inheritance tree is exhibited through rectangles. The width and length of the rectangles is used by SME to represent software attributes, such as the number of methods and size of a class. The third view aims to portray the dependency among the modules of an application. It uses interactive directed graphs (IDG) to describe coupling between software’s modules, in this case, software modules that depend on each other. The IDG view uses nodes to represent the modules and directed edges to represent the dependencies among them.

SME provides highly interactive views. This is a key issue in software visualization tools, especially when analysing large industrial software applications. Filters, zooms, and coupled navigation between views are provided by SME. The developer can, for example, zoom in a view to observe an element in detail. He can also use filters to search for specific application’s implementation elements or to select entities according to its properties. These filters are very useful in large applications as many elements are difficult to locate or cannot be drawn together in the same screen shot. Additionally, the developer can navigate between the views. For instance, if the developer identifies a class or interface of interest in Polymetric or IDG views, he can “shift right click” on the visual attribute that represents the implementation element and ask to open it in the Treemap view. The views are coordinated so that a change in some filter updates all the views at the same time.

SME uses the representation of features provided by the CM tool, as aforementioned. The developer can select any color and associate it to existing mapped feature. SME paints the implementation elements that realize a feature with the color selected by the developer. The developer uses a range bar slider to select any two sequential versions to observe the differences between them. We call this an evolution differential approach [18]. The three views will portray the elements of the most recent selected version and compare it with the other one. As an example, consider two sequential versions \(i\) and \(i + 1\). The elements of version \(i + 1\) are drawn in the views, and colors are used to portray their differences from the previous one. SME uses three different colors to do that. It paints in light blue color the implementation elements...
that realize features in version $i$, but not in $i+1$. On the same token, if an implementation element realizes a new feature in $i+1$, it is painted in dark blue. A purple color is used to represent an element dropped at least one feature and added at least another feature from version $i$ to $i+1$. The last case, many times involve elements that have been removed from one feature and added to another one, or vice-versa. For this reason, we name these elements as transferred. Figure 1 shows the views and the colors used.

**Figure 2.** Evolution of the feature Transaction in the Treemap view.

Figure 2 illustrates an example of the Treemap view. It shows the evolution of the class TransactionService from version $i$ to $i+1$. In this class, there are some methods that realize the feature Transaction. This example is part of the case study used in our experiment (see Section IV). We can observe that the method `shutdown()` was added to the version $i+1$ to realize the Transaction feature. On the other hand, the method `getProxy()` does not realize the Transaction feature anymore (version $i+1$). In this sense, the light blue color indicates the elements that do not realize a given feature anymore (method `getProxy()`); whereas the dark blue color indicates elements are now realizing the feature (method `shutdown()`).

An important aspect of this interactive phase is that it allows the developer to confirm or to reject the implementation elements mapped to a given feature by the heuristics. The proactive phase is thus seen as a suggestion that can be checked and validated by the developer. In spite of its importance, the effectiveness of this functionality was not assessed in the experimental study (Section IV).

**IV. EXPERIMENTAL EVALUATION**

The goal of this experiment is to evaluate quantitatively the effectiveness of the proposed visualization strategy when compared to a tree-structure strategy. Although diff tools [15, 16, 17] are more used than the CM tool in industry, we did choose the latter (as baseline) for the following reasons: (i) diff tools do not provide explicit visualization support for features, the main concept addressed in this work, and (ii) their purpose is completely different as they are intended to show the overall textual differences of the entire application. It is also important to remark that both the SME and CM tools are supported as plug-ins of the Eclipse IDE. The idea is to investigate if the use of feature evolution visualization is really a promising strategy to be adopted in an industrial scenario. In addition, the study will be useful to contrast a lightweight visualization (CM) with the proposed strategy (SME). Contrary to general understanding, software visualization tools also present drawbacks associated with their use [10, 11, 13, 14]. The effectiveness of the evaluated strategies is measured quantitatively in terms of time and correctness. These metrics were similarly adopted in experimental studies on software visualization and feature comprehension [19, 24]. The experimental evaluation is described from Section A to F.

**A. Study Hypothesis**

The time and correctness measures were gathered for the tasks on feature evolution comprehension. Both measures for the SME and CM tools are compared in order to reveal which strategy tends to be more beneficial. Table I presents the description of the hypotheses. The first hypothesis (H1) states that the SME strategy helps developers to faster comprehend the feature evolution. The second hypothesis (H2) states that the SME strategy improves the correctness of feature evolution comprehension tasks. The null hypotheses are the negation of the aforementioned ones.

**TABLE I. STUDY HYPOTHESES**

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>The SME strategy decreases the time spent in feature evolution comprehension tasks.</td>
</tr>
<tr>
<td>H2</td>
<td>The SME strategy improves the correctness of feature evolution comprehension tasks.</td>
</tr>
</tbody>
</table>

**B. Experimental Object**

The effectiveness of the visualization strategies was evaluated through five different versions of an industrial application. This experimental object is a logistic software application for the oil industry. It was chosen because: (i) it is a real world and evolving application that has been developed since 2006; (ii) it has a significant size, on average 120 KLOC, and complex modules; (iii) it underwent various forms of changes during their evolution; (iv) it contains a significant set of functionalities with different degrees of granularity; and (v) finally, its features have a large number of implementation elements realizing them. We have omitted the application name for confidentiality purposes.

Table II describes the analysed features in the application. We chose a few features from the application domain (Notification, Route and Report) as well as a well-known crosscutting feature whose concept is widely known (Transaction). Such features have undergone many changes across the five selected versions in terms of added, transferred and removed feature’s elements (Section III.B).
C. Participants

We run the experiment with 20 participants who work in industry as software developers in two venues in Brazil (Salvador – BA and Rio de Janeiro – RJ). It was selected 10 participants in each venue. The selection assured that the participants had both industry experience and the knowledge required by the experiment (see Figure 3). All participants were full or part-time developers in different companies. They have worked with software design and development of different domains, from web to real-time applications. The participants had similar expertise levels but different levels of experience (i.e. time in industry). We selected experienced (more than five years) and novice developers (less than five years) to assure that the strategies can be beneficial for both cases. During the selection we tried to assure that the participants: (i) had previously faced problems related to the evolution and reengineering of applications, and (ii) had knowledge on the concepts tackled in this paper, such as software evolution and development, and features, and on the instruments used in the experiment. It is important to highlight that such participants are not familiar with the experimental object. This was a requirement in this experiment as the goal is to observe how new developers in organizations can successfully perform feature comprehension tasks when using two different visualization strategies.

Before starting the experiment, we asked each participant to fill-in a characterization form [30], where they could attest their experience and level of technical expertise on the technology and concepts used in this study. This was used to allocate them in the two treatments. Figure 3 shows the average expertise of the participants by groups (i.e. those who used the CM tool and the SME tool) for the main concepts addressed in this work. The expertise level of each participant ranged from 0 (no knowledge) to 3 (Expert) on a certain subject. As shown in Figure 3, the participants of each group have similar expertise in the main topics of the experiment. The average industrial experience of the participants is about five years for each group. The participants were also divided into two groups considering his or her venue of origin, each one addressing a tool.

D. Task Design

We try to define representative tasks of feature evolution comprehension in the context of reengineering of applications. The definition of the tasks was based on demands identified during the maintenance and evolution processes of the experimental object. Therefore, they represent real world comprehension scenarios. We also considered these tasks because they may lead to potential mistakes performed by developers during the feature comprehension tasks [32]. Additionally, these tasks were selected based on their different levels of difficulty.

### Table III. Description of the Feature Comprehension Tasks

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Goals</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1-3</td>
<td>Feature Evolution Analysis</td>
<td>From version 1 to 2 what are the methods in the ValidationFrame class now realizing the feature Notification?</td>
</tr>
<tr>
<td>T4-5</td>
<td>Feature Tangling Analysis</td>
<td>Is the ServerConfig class realizing more than one feature through the application evolution? If yes, what are the features? For each feature, in each version, what are the methods realizing it?</td>
</tr>
<tr>
<td>T6</td>
<td>Feature Dependency Analysis</td>
<td>From version 1 to 2, what are the features of which the RemoveFolderCall class depends on? What are the features realized by the RemoveFolderCall class?</td>
</tr>
</tbody>
</table>
Feature tangling analysis aims to observe if there are modules that realize more than one feature and how they do it. Finally, the feature dependency analysis aims to observe how two features are communicating through their modules. On the description of each task, it is specified the feature name, implementation elements and versions under analysis. We followed this strategy for designing the tasks because: (i) the participants are not familiar with the application; (ii) the application is very complex and large; (iii) it was established a time limit for the experiment in order to avoid participant fatigue; and (iv) finally, it does not favour any of the tools for the execution of the tasks.

E. Experimental Procedures

The experiment was run in four sessions, two for each treatment (SME and CM) in each of the two venues (Salvador - BA and Rio de Janeiro - RJ). The sessions were supervised to avoid parallel conversations and provide support for eventual problems during the experiment execution (i.e. tools or computers). The tools were installed and tested as IDE plug-ins prior to the experiment execution. The source code of the experimental object was available to all participants during the experiment. The experiment was to be concluded within 80 minutes. This time was defined and adjusted after a pilot study (see Section F). The main experimental procedures were:

1. The CM and SME tools used the mappings generated by the heuristics (Section III.A). Thus, we guaranteed that all the application versions had the same mappings when executed on both tools;

2. The participants filled-in the characterization form (Section C) and were allocated to one treatment (strategy to be used during the comprehension tasks);

3. A training session of 30 minutes was conducted in order to explain and show how both tools work. Additionally, we also explained the main concepts and motivation behind the study. Two small tasks of a simple example were performed. The participants could observe how to use the tools by means of these small tasks. They were encouraged to address any doubts or concerns during the training session;

4. A practice homework was sent to the participants to make them familiar with the tool of their treatment. We used the MobileMedia system [29] as experimental object. Each participant received a set of tasks to practice on it. Screenshots were also sent to the participants in order to guide them during the homework tasks. All the answers were sent by email to the experimenters in order to guarantee that all participants accomplished the proposed tasks;

5. The experiment itself was run in university laboratories where the entire environment was set up. The time was measured by considering the difference between the final and initial time of each task. The time of any interruption during the task execution was deducted from the total execution time;

6. We analysed the results. Correctness was measured by comparing the correct answer model with each participant’s answer. We converted the answers into quantitative values by using the same criteria used in [19]. An entire task correctly answered was worth 1 point. Wrong answers counted negatively. Thus, if the task contained 5 answers, each correct one was worth 0.2 and each wrong one was worth -0.2 points. If the final score received in the task was 1 point, we classified it as correct. If it was between 0 and 1, not inclusive, we classified it as partially correct. Finally, if the final score is equal or less than 0 we classified it as incorrect;

7. Finally, we applied a feedback questionnaire. This questionnaire was applied on-line in order to guarantee anonymity and avoid intimidating the participants. They were asked about the experiment design, training, homework, execution and tasks.

F. Pilot Study

A pilot study was performed prior to the experiment with the intention of identifying certain problems in its procedures, or even in the tools, which are difficult to predict during its execution. Four participants were selected to perform the pilot study. Two of them used SME and two of them used CM. It is important to highlight that these four participants did not take part of the final experiment. The pilot study allowed us to improve the tasks’ description, goals and degree of difficulty. In addition, it was also essential to tune the experiment timing in 80 minutes.

V. RESULTS

This section discusses the experimental results. The statistical tests used are briefly described in Section A. Section B discusses the results in terms of time spent for accomplishing the tasks and, therefore, testing H1. Section C discusses the correctness of the tasks and tests H2.

A. Statistical test

The statistical tests were used to accept or reject the hypotheses listed in Section IV.A. The statistical analysis was composed of three phases. First, descriptive analysis was produced with the computation of measure’s mean, minimum value, maximum value and standard deviation. The data was also analysed to verify if the sample distributions were normal and had equal variances. To this end, we applied the Shapiro-Wilk and Levene tests, respectively. Second, as far as planning is concerned, we selected two tests, t-test and Mann-Whitney, to be applied depending if the sample distributions were normal and homoscedastic, or not. The sample distribution of time was normal and with equal variance. Hence, we used the parametric t-test to analyse the hypothesis H1. The sample distribution of correctness was not normalized and thus we select the non-parametric Mann-Whitney test to analyse the hypothesis H2. We used a confidence level of 95%
Figure 4 summarizes the time spent by the participants considering the tasks correctly concluded. It only takes into consideration the time of those who answered the entire task correctly (i.e. scored 1 point). Therefore, not all the 20 participants appear on the graphic for each task. The idea is also to analyse which tool tends to lead to the achievement of best correctness in the tasks. Our analysis revealed that the aspects consistently influencing the time for every successfully task execution were: tool configuration, feature analysis issues, and tool processing cost. We also discuss the statistical results regarding the hypothesis H1.

**Tool Configuration.** This factor refers to the time that the participants took to set some configuration properties of a tool before carrying out a given task. Using CM, the participants only need to open the mapping files of each version in its view and observe the differences. On the other hand, to perform a given task using the SME tool, the user needs to specify the versions to be compared, select colors for the respective features, and open a view to analyse them. We observed that when the tasks do not demand much effort, the configuration time significantly influences the final results. This occurs because some participants who used SME took most time choosing their preferred colors for representing each feature and also choosing the most appropriated view. As mentioned, SME only defines the colors for assigning the elements under evolution when comparing the versions (Section III.B); whereas the colors for the features are chosen by the user. An example of this situation occurred in T1 (Figure 4) where CM presented some better results than SME.

The choice of the most appropriated view is indicated by the type of task. The appropriateness of the SME views per task was illustrated in the training sessions. For instance, if the task is related to methods, it is common to use the Treemap view as it gives more details about them (Section III.B). However, tasks can be visualized in more than one view, which induce the participants to spend time deciding which view should be used.

**Feature Analysis Issues.** This factor is related to the time used for observing the differences among the features across the versions. These differences are related to the removed, transferred and added feature’s elements. The participants who used CM mainly focused on the use of set theory for identifying, for instance, which elements were removed. This analysis requires a meticulous observation of the feature mappings in CM. As a consequence, it impacts on the time spent for executing the tasks. On the other hand, when using the SME tool, the participants used colors to identify the feature’s implementation elements. The use of colors in the Treemap view can be observed in Figure 2. This scenario was evident in T2, T3 and T6 where SME users achieved better results than CM users. Even considering the time spent to set some properties for history analysis, the use of SME yielded faster performance of the participants while successfully accomplishing their tasks.

**Tool Processing Cost.** The processing cost refers to the time that a tool takes for executing a participant’s action. We observed that this cost was slightly higher for SME in T4 and T5. This occurred because when using CM, the participants only need to load the mappings and observe the evolution of the feature mappings. Consequently, the time is related to the analyses made by the participants. On the other hand, for processing T4 and T5, SME demanded more time due to the graphical representation of the large data. In fact, this is a current issue in many visualization tools and already reported by other works [9, 13]. However, we noticed that this problem is even more relevant in the context of software evolution as multiple versions of an application have to be considered in the analysis.

**Statistical Results.** Table IV shows the results of the time statistical analysis. Time here was computed for all tasks performed, including right, wrong and partial answers. Based on the results, we can see that SME had a lower mean considering all the tasks (8.95%). However, we could not reject the null hypothesis H1 (p-value = 0.3754). This can be explained by the limited total time for the experiment (80 min). Some participants that used the CM tool had their time expired before successfully accomplishing all the tasks.

Figure 4. Participants’ Time vs. Tasks.
adjust or confirm their answers from different perspectives. In addition, the use of colors also helped to clearly distinguish the removed, transferred and added actions performed on features’ elements (e.g. Figure 2). As a result the use of different colors seems to facilitate non-trivial analytical tasks on feature evolution comprehension. This is particularly interesting as the use of colors in feature comprehension tasks had only been evaluated in the context of individual application versions [8].

In fact, the higher number of incorrect answers is associated with the use of CM. This difference occurred because the functionalities provided by it are not as powerful for helping developers to understand the feature evolution. For example, CM does not provide an intuitive visual way to verify the differences among added, transferred and removed elements. Even though CM incorporates feature representation based on tree-structure, the feature evolution analyses tend to be more difficult as they are based on set theory. This occurs because the use of a single perspective tends to hinder the analyses of different feature properties.

Evolution Analysis. The number of incorrect answers increases when the feature evolution analysis becomes more complex. This complexity is directly associated with the identification of added and removed elements realizing a feature and analysed at the same time. This scenario was evident in T3. In this task, all participants who used SME provided correct answers while those who used CM completely failed in providing correct answers. The definition of T3 refers to find the transferred elements (Section III.B). The use of colors and “mouse over” the element to observe which features this element realizes were essential to correctly answer this task. Additionally, the use of the slider was also useful as the participants can quickly select and observe the differences among any versions of an evolving application. It is important to highlight that most participants who used CM gave up answering due to the difficulty in identifying the feature’s elements under evolution.

Proactive Mappings. As mentioned, both SME and CM relied on feature mappings generated by the mapping heuristics (Section IV.E). It is realistic to expect that such mappings may contain a few imperfections or mistakes as those ones performed by hand [32]. We analysed all the mappings in order to identify if there were major mistakes that would hamper the participants of completing their tasks (Section IV.D). This was not the case. For the minor mapping mistakes, we observed that the use of visualization can help to circumvent them. SME provides by means of colors a clear way to identify what has occurred during the feature evolution. Also, the user can evaluate the correctness of the mappings through the views. For example, let’s suppose that a new element is realizing a feature, which is modularized in only one module. If this new element does not belong to this module, it can be an example of mistake in feature mapping. The modularization of features can easily be observed using the Treemap view. The Dependency view can also be used to identify mapping mistakes through the coupling between elements realizing different features. For example, if one new element realizes a feature and has no dependency with others that realize it as well, this view helps to identify this mistake. Unfortunately, the single perspective view based on the CM tree-structure does not help the users to alleviate these problems. This is a key CM limitation as the accuracy of the best feature mining techniques tends to be 90% or lower [33].

Statistical Results. Table IV shows the results of the statistical analysis. The results show that SME entailed a significantly higher correctness when comparing it to CM. The SME mean was 26.94% higher than CM. The Mann-Whitney test p-value = 0.001192 supports the hypothesis H2. Therefore, we can conclude that the SME tool provides an improvement of the correctness in comprehension tasks involving feature evolution.

VI. Lessons Learned

This section presents some lessons learned derived from the experiment. We discuss the feature evolution comprehension by correlating time and correctness (Section A) and the user’s point of view (Section B).
A. Time and Correctness can go hand-in-hand

Regardless of the participants’ expertise, we could confirm that it is possible to reach a higher degree of correctness when there are distinct feature graphic representations. We could observe that users, who checked their solutions under different perspectives, tended to generate solutions with higher accuracy and precision. As the level of user confidence increases, the time spent to realize the most complex tasks is reduced. We could derive this lesson in our experiment by analyzing the performance of the participants using the SME tool.

The SME tool provides means to check solutions under different perspectives and thus improving the number of correct answers (Section III.B). To be precise, 90% of the given answers were completely correct; whereas only 58% of the participants who used the CM tool provided fully correct answers (Section V.C). In terms of time, the use of SME tool also presented gains. The time spent to execute tasks varied a lot, but the use of SME reduced the average task time in 8.95% when compared to CM. For this reason, if we compared time versus correctness, SME presented better results. Even considering that some participants took more time to provide answers, 90% of them successfully accomplished the tasks (Section V.B).

The correctness gains were largely caused by the use of colors to distinguish feature behaviour and the possibility of participants interacted with the SME views. The evolution of features can be easily visualized as it is represented with distinct colors. The participants can interact with the SME tool by selecting the target versions to be compared and checking their given answers under different views. It is important to mention that the level of participants’ expertise is not correlated to correctness or time. The SME tool could be satisfactorily used by experienced and novice participants [30].

B. User’s Point of View

We asked the participants about which characteristics they believe should be present in a tool designed to support the application evolution analysis. According to them, the tools should provide visual resources in order to facilitate smooth analyses of source code evolution. They want to be able to conduct faster the evolution analysis of the source code so that they can concentrate on the modification of the code itself.

The SME tool meets the software developers’ wishes as it provides a set of interactive mechanisms supported by different views (Section III.B). In our observation it was possible to register the massive use of the views (Section V.C). The possibility of having an overview of the source code under a fine-grained perspective (e.g. level of methods) facilitated the analysis of the implementation elements that realize a given evolving feature under different perspectives.

Another SME mechanism that caught the participants’ attention was the use of colors to identify the removed, transferred and added features’ implementation elements. It helped them to easily execute a wide range of required tasks. SME provides other mechanisms to identify feature evolution. For instance, users can move the mouse or use the menu-popup interactions over the target element in the views to identify features (Section III.B). Although the colouring was the preferred mechanism to identify feature evolution, the other options were frequently used by the participants to double check their answers.

VII. Threats to Validity

The threats to validity of our study and how they were mitigated during our experiment are described below [31].

Conclusion Validity. We identified three threats in this category: (i) design of the tasks: the tasks of feature evolution comprehension could have been too difficult or biased to a specific tool. To mitigate this threat, we extracted a set of recurring tasks from industrial scenarios. The difficulty of these tasks was analysed in the pilot study where the participants were able to perform them in both tools; (ii) time restrictions: the time allocated to the experiment could have influenced the participants’ answers. To circumvent this threat, the pilot study was used to estimate the suitable average time to perform all the tasks in both tools. In addition, according to the questionnaire answered by the participants after the experiment, the given time was enough to perform all the tasks [30]; and (iii) heterogeneity of the participants: it refers to the selection of the participants involved in the experiment. To alleviate this threat, we selected experienced and novice participants from the industry based on two different venues in Brazil. Although this heterogeneity can still be seen as a conclusion threat, it contributes, on the other hand, to minimize the external threats of the study.

Construct Validity. We identified two threats in this category: (i) operational procedures of the experiment: the participants may have not properly understood the experiment guidelines. To minimize this threat, we delivered a training session explaining the use of the tools and the experimental execution process. Simple tasks were performed in the training session in order to show how to use the functionalities of each tool. We also elaborated a practice homework to the participants so that they could familiarize with the tool functionalities; (ii) confounding constructs and levels of constructs: it refers mostly to the expertise level of the participants. To reduce this threat, we selected industry developers with different levels of experience.

Internal Validity. We detected one threat, which refers to the allocation of participants to the treatments. It is related to the partition of the participants in the groups (CM and SME). To minimize this threat, each participant filled in a form about their experience in industry and expertise in the concepts and techniques explored in the experiment. We partitioned the participants taking into consideration their answers in order to make the partition as fair as possible.
External Validity. The threat in this category is related to the representativeness of the artefact used in this experiment. To circumvent this threat, we selected five versions of an evolving industrial application, which is quite representative in terms of changes and size. We believe the results obtained from this artefact are representative of what one would expect from typical industrial applications.

VIII. CONCLUSIONS

This paper has presented a proactive and interactive visualization strategy for assisting developers in feature evolution comprehension. To evaluate this strategy, it was conducted an experiment comparing the SME and CM tools. In this experiment, we selected a representative set of professional developers and used a real evolving industrial application. A set of tasks that encompasses typical feature evolution comprehension was also selected. The experimental results revealed positive results related to the use of SME. According to the statistical tests, the hypothesis related to the time spent to accomplish the feature evolution comprehension tasks could not be accepted or rejected. However, we could observe that SME reduced the average task time in 8.95% when compared to CM. Nevertheless, the hypothesis related to the correctness was accepted, indicating that SME yielded better correctness when compared to CM. We also derived two lessons learned from the experiment results and observations. These initial results demonstrated the benefits of SME and its usefulness to help developers in feature evolution comprehension mainly regarding correctness but also presented gains related to the time.

ACKNOWLEDGMENT

This work has received full or partial funding from the following agencies and projects: for all the authors PROCAD-NF (grant 720/2010); for Renato and Manoel - INES funded by CNPq (grant 573964/2008-4); for Alessandro - FAPERJ (distinguished scientist grant E-26/102.211/2009); DANSis project (grant E-26/111.152/2011), CNPq (productivity grant 305526/2009-0), PUC-Rio (productivity grant), Universal Project (grants 483882/2009-7, 483699/2009-8, 485348/2011-0), CAPES: international collaboration scheme (grant 5688-09).

REFERENCES