Noname manuscript No.
(will be inserted by the editor)

Comparing the Influence of Using Feature-Oriented Programming and Conditional Compilation on Comprehending Feature-Oriented Software

Alcemir Rodrigues Santos · Ivan do Carmo Machado · Eduardo Santana de Almeida · Janet Siegmund · Sven Apel

the date of receipt and acceptance should be inserted later

Abstract Several variability representations have been proposed over the years. Software maintenance in the presence of variability is known to be hard. One of the reasons is that maintenance tasks require a large amount of cognitive effort for program comprehension. In fact, the different ways of representing variability in source code might influence the comprehension process in different ways. Despite the differences, there is little evidence about how these variability representations – such as conditional-compilation directives or feature-oriented programming – influence program comprehension. Existing research has focused primarily on either understanding how code using modern paradigms evolves compared to the traditional way of realizing variability, namely conditional compilation, or on the aspects influencing the comprehension of conditional compilation only. We used two different programs implemented in `Java` and each of these variability representations. As Java does not support conditional compilation natively, we relied on the mimicking (*i.e.*, preprocessing annotations in comments) that has been used in the literature. Our results show no significant statistical differences regarding the evaluated measures (correctness, understanding, or response time) in the tasks. Our heterogeneous sample allowed us to produce evidence about the influence of using CC and FOP variability representations on the aspects involved in the comprehension of feature-oriented software, while addressing bug-finding tasks.

Alcemir Rodrigues Santos
Federal University of Bahia, Brazil, E-mail: alcemirsantos@dcc.ufba.br

Ivan do Carmo Machado
Federal University of Bahia, Brazil, E-mail: ivanmachado@dcc.ufba.br

Eduardo Santana de Almeida
Federal University of Bahia, Brazil, E-mail: esa@dcc.ufba.br

Janet Siegmund
University of Passau, Germany, E-mail: janet.siegmund@uni-passau.de

Sven Apel
University of Passau, Germany, E-mail: apel@uni-passau.de

1 Introduction

Several variability representations have been proposed over the years to handle variability in source code, and they have been increasingly applied in the development of large and complex software systems [3]. Some of them have reached high levels of popularity in industry, such as Conditional Compilation (CC) [17], whereas others are still mostly known in academia, such as Feature-Oriented Programming (FOP) [5].

The research community has attempted to bridge existing gaps from both paradigms to improve their adoption and make them easier to understand. Nevertheless, software maintenance in the presence of variability can still be considered hard [17]. The maintenance of source code in the presence of variability might require a large amount of cognitive work for comprehension, for a set of different factors, such as the need of language processing or the visual perception of the source code [28]. However, apart from the studies addressing *(i)* the differences in the effort to evolve software systems using different variability representations [12,14] and *(ii)* the work focused on program comprehension from the perspective of a given particular variability implementation mechanism - (CC) - [16, 19, 26], there is still a lack of understanding about how such representations might influence program comprehension, in particular to carry out bug-finding tasks. To the best of our knowledge, only three studies specifically addressed the influence of variability representation differences on the topic [23, 24, 29]. First, Siegmund et al. [29] conducted a pilot of a *(quasi-)*experiment with `Java` systems implemented with FOP and CC. Later, Santos et al. [23] carried out a study with `JavaScript` systems implemented with the "standard" way to implement variability and the RiPLE-HC approach. Santos et al. [24] stressed different aspects from which software engineers could either benefit or be confused with while addressing bug-finding tasks in code using FOP and CC variability representations. Indeed, it is important to address such differences from the perspective of program comprehension because of two main aspects: *(i)* the ever increasing complexity of software systems with variability in their code; and *(ii)* the difficulty to assess and select one among the plethora of currently available variability representations.

Shull et al. [27] highlighted benefits of replications in Empirical Software Engineering. This fact motivated us to carry out two replications of Siegmund et al.'s [29] pilot to add to our earlier efforts [23, 24], as a means to contribute to a better understanding of the influence of different variability representations on feature-oriented software comprehension. Their pilot targeted software developed with two different variability representations: FOP and CC. We believe these two variability representations are representative of the emerging and solid approaches to handle variability. While FOP is language-independent, CC could be considered the most widely used variability representation (*e.g.*, the `C` and `Java` preprocessors, such as Antenna, Munge, and JavaPP) [17]. Therefore, likewise Siegmund et al. [29], we recruited 33 graduate students as participants and designed a number of bug-finding tasks in two versions (FOP and CC) of two open-source systems. We assessed three dependent variables: understanding, correctness, and response time. We believe these variables could capture important aspects of the influence of variability representations on the feature-oriented software comprehension.

Regarding the two replications, we performed one *dependent* (or *exact*) replication using the same design and artifacts, and one *independent* (or *conceptual*) replication, in which we extended the original design with a second round using a system from a different domain to strengthen validity.

In summary, our hypotheses testing did not allow us to reject our null hypotheses on no difference regarding the dependent variables (understanding, correctness, and response time) while using the the varaiability representations used in this study. In summary, we make the following contributions:

- A more robust design for experimentation with FOP and CC variability representations if compared with previous ones;
- Indicators of no statistical difference regarding *(i)* the effort to understand, *(ii)* the effort to find errors in source code, and *(iii)* the time required to finish bug-finding tasks while using CC and FOP;
- A supplementary Web site with the experimentation artifacts of the carried experiments[1], which may provide researchers with a replication package.

The remainder of this article is organized as follows: Section 2 presents underlying concepts about Feature-Oriented Software Development. Section 3 describes the planning, preparation, and execution of the (*quasi-*)experiments, including the addressed research questions and hypotheses. Section 4 presents the results of the empirical evaluation. Section 5 discusses the influence of the participants' motivation and difficulty perception concerning their results in our replications comparing the current with the original study, as well as the answers to the stated research questions. Section 6 discusses the threats to validity and Section 7 discusses existing related work. Finally, Section 8 draws concluding remarks and pinpoints opportunities for further investigations.

2 Feature-Oriented Software Development (FOSD)

FOSD is a paradigm used for the construction, customization, and synthesis of large-scale software systems relying on features [4]. A *feature* satisfies a requirement while performing design decisions, which in other words means an increment in the program functionality [6]. In fact, a software manufacturer can generate a software product based on the requirements of a specific customer based on a set of reusable parts [3]. The ability of handling features in FOSD is known as variability management and is accomplished at the implementation level by a variability representation. Apel et al. [3] built an extensive catalog of variability representations, which might be used to handle such reusable parts.

In this study, we addressed two variability representations: FOP and CC. Figure 1 illustrates the main differences between (a) (CC) and (b) (FOP) approaches to FOSD. The latter emphasizes the modularization. FeatureHouse [5] was selected as the tool implementing FOP, whereas Antenna and JavaPP are tools implementing CC. In addition, FeatureHouse is representative of the group of techniques that *physically* separate the implemented features in the code base, whereas Antenna and JavaPP are representative of the group of approaches that *virtually* separate the implemented features. We use Antenna and JavaPP instead of `C`-preprocessor, because we needed subject systems implemented in both FOP and CC. Next, we present these tools.

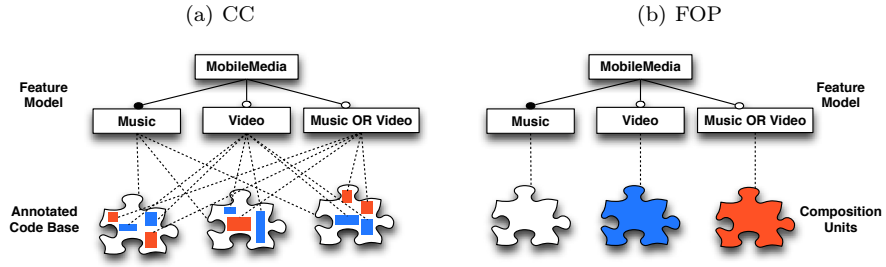


Fig. 1 CC and FOP approaches of Feature-Oriented Software Development. Adapted from Apel *et al.* [3].

2.1 FeatureHouse

FOP supports the so-called *positive variability* – when composition units are added on demand – and aims at keeping a traceable mapping between features and composition units [3]. More specifically, FeatureHouse [5] is a programming-language-independent FOP technique to implement variability, which provides mechanisms to compose artifacts to derive products in a composition-based approach. Moreover, feature modules are represented by file-system directories – called *containment hierarchies*, in which the classes and their refinements (*i.e.*, feature-specific lines of code of a class) are stored in files inside the corresponding containment hierarchies [3].

Figure 2 shows three code snippets with the refinement of a given method by different features in the `MobileMedia` application [13] with FeatureHouse [29]. `MobileMedia` is one of our subject systems, and it is further discussed in Section 3. Each code snippet concerns a different feature implementation located in a `Java` file with the same filename, class name, and method signature, but a different feature code container, namely "Music_OR_Video", "Music", and "Video". These different snippets are supposed to be composed depending on the product configuration and the order in which they were listed for binding. Indeed, FeatureHouse preserves most of the language syntax (`Java`, in this case), but it introduces mechanisms to compose pieces of refinement code as defined in these snippets by using the original() method call and Feature Structure Tree [5] for composition. This method is the link among the existing refinements in these three implementations. It guides the execution of the instructions sequence, depending on which features are selected for binding and their order of composition.

2.2 Antenna and JavaPP

CC is a technique well known from the `C/C++` languages and later implemented in other languages, either natively or supported by third-party tools. Antenna[2] and JavaPP[3] are examples of these third-party tools that enable the implementation of `C`-preprocessor directives (*e.g.*, #if, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`) in `Java`

[1] http://rise.com.br/riselabs/vicc/
[2] Available at `http://antenna.sourceforge.net/`
[3] Available at `http://www.slashdev.ca/javapp/`

(a) Music_OR_Video

```
1  public class MusicMediaUtil extends MediaUtil {
2     private boolean isSupportedMediaType(MediaData ii){
3          return false;
4     }
5
6     public byte[] getBytesFromMediaInfo(MediaData ii)
7                throws InvalidImageDataException{
8          try {
9               byte[] mediadata = super.getBytesFromMediaInfo(ii);
10              if (ii.getTypeMedia() != null) {
11                 if (isSupportedMediaType(ii)){
12                   ...}
13              }
14              return mediadata;
15         } catch (Exception e) {...}
16    }
17 ...
18 }
```

(b) Music

```
19 public class MusicMediaUtil extends MediaUtil {
20      private boolean isSupportedMediaType(MediaData ii){
21          return original(ii) ||
22                ii.getTypeMedia().equals(MediaData.MUSIC);
23     }
24 }
```

(c) Video

```
25 public class MusicMediaUtil extends MediaUtil {
26     private boolean isSupportedMediaType(MediaData ii){
27         return original(ii) ||
28                ii.getTypeMedia().equals(MediaData.VIDEO);
29     }
30 }
```

Fig. 2 `Java` with FeatureHouse code example extracted from `MobileMedia` [29].

programs. They allow software developers to control the inclusion of the code that belongs to the selected features or the exclusion of the code from the deselected ones. The CC version of our subject systems use Antenna and JavaPP to represent their variability. Several studies have considered preprocessors in `Java` while investigating the effects of existing differences among variability representations [12, 14].

CC supports "negative variability" – when parts of the code are removed from the final product on demand. The drawbacks of the annotative approaches, such as the lack of modularity and poor readability, have been criticized in the past [10]. However, the ease of use and its flexibility made it the most used variability-implementation mechanism in practice [17].

Figure 3 shows one code snippet with the refinement of a given method by different features in the `MobileMedia` [13] with Antenna. JavaPP uses the same syntax. All the

```
//#if includeMusic || includevideo
...
public class MusicMediaUtil extends MediaUtil {
    public byte[] getBytesFromMediaInfo(MediaData ii)
             throws InvalidImageDataException{
        try {
            byte[] mediadata = super.getBytesFromMediaInfo(ii);
            if (ii.getTypeMedia() != null) {
//#if (includeMusic && includevideo)
                if ((ii.getTypeMedia().equals(MediaData.MUSIC)) ||
                     (ii.getTypeMedia().equals(MediaData.VIDEO)))
//#elif includeMusic
                if (ii.getTypeMedia().equals(MediaData.MUSIC))
//#elif includevideo
                if (ii.getTypeMedia().equals(MediaData.VIDEO))
//#endif
                 {...}
            }
            return mediadata;
        } catch (Exception e) {...}
    }
...
}
```

Fig. 3 `Java` with ANTENNA code example extracted from `MobileMedia` [13].

code is in a single file, which is going to be preprocessed to remove the code of deselected features before the actual compilation, depending on the product configuration. The snippets shown in Figures 2 and 3 are equivalent in the sense of the final software product the variable code would produce.

3 Study Settings

In this section, we present the planning, research questions, hypotheses, and the execution of our two (*quasi*-)experiments [33] replications.

3.1 Planning

The planning of an experiment concerns the experiment design, the selection of the participants, the tasks performed, and the supporting material available to the participants during the experiment session. Next, we detail each of them.

3.1.1 Design

We carried out the quasi-experiments in the form of two replications, consisting of three rounds. All rounds were inspired by the pilot of Siegmund et al. [29]. The participants were computer science graduate students from Federal University of Bahia, Brazil. Figure 4 shows the design of the replications. In fact, the *first* replication was executed in

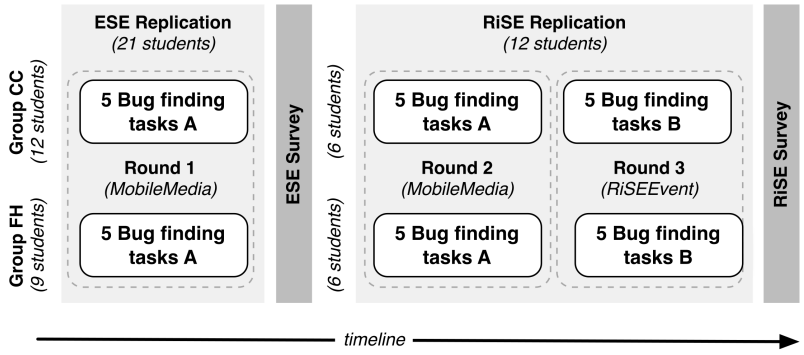


Fig. 4 Experiment design. Group CC means the group using CC and FH means the group using FOP.

one round (Round 1) as an *exact replication* of Siegmund et al.'s pilot [29] – when using the exact same experiment design with a different sample. The *second* replication is a *conceptual replication* – although investigating the same or related research questions, one is doing it by using a different experiment design – [27], executed in two rounds (Round 2 and 3). Round 2 is exactly the same as Round 1; in Round 3, we elaborated the tasks trying to mimic the level of difficulty of Siegmund's pilot in a second system, which was executed one week after Round 2.

In total, 33 students participated in our experiments. We recruited 21 from an "Empirical Software Engineering" (ESE) course to participate in Round 1. The other 12 were recruited from the members of the "Reuse in Software Engineering"[4] (RiSE) research group to participate in Rounds 2 and 3. Each participant worked at an individual workstation and participants were not allowed to communicate with each other during the experiment session.

In both replications (ESE and RiSE), we split the participants in two groups, each addressing a different variability representation, which we are going to call hereinafter as "Group CC" (*i.e.*, the participants using CC) and the "Group FH" (*i.e.*, the participants using FOP). We attempted to balance the groups by considering their programming experience (*i.e.*, each group would have similar number of experienced programmers – in terms of their programming years, working in industry, and courses taken – in each group, regardless of the addressed variability representation). In the ESE replication, the grouping is unbalanced by three students in the group FH due their absence in the experiment session. In the RiSE replication, we used a cross-over design where the participants of the FH group in Round 2 switched to the CC group, to perform the tasks of Round 3 and vice-versa. We split the groups in the same way as we did in ESE replication.

[4] `www.rise.com.br/`

3.1.2 Subject Systems

We used two open-source systems as subjects: `MobileMedia`[5] [13] and `RiSEEvent`[6] [8]. We selected them because they are from different domains, and both are available in two equivalent versions: one in `Java` CC directives based on the ANTENNA and JAVAPP support and its respective refactored version in `Java` with FEATUREHOUSE [5].

`MobileMedia` is a small mobile phone application that serves the purpose of management of media on mobile phones, initially developed by Figueiredo et al. [13] in two versions: `Java ME` with conditional compilation and ASPECTJ. Later, Siegmund et al. [29] built a refactored version with FOP.

`RiSEEvent` is a software product line of a scientific event management system implemented with conditional compilation by Silveira Neto et al. [8] and refactored into a FOP version by the first author of this paper.

Table 1 shows metrics of packages (as in the FOP version there are several package duplications, we decided to add the number of feature-code containers), classes, and lines of code for each version.

Table 1 Subject systems characterization.

	`MobileMedia`		`RiSEEvent`	
	CC	FOP	CC	FOP
Packages	9	35*	8	40*
Classes	52	52	496	559
Lines of Code	∼3000	3823	26457	28771

CC: Conditional Compilation; **FOP:** Feature-Oriented Programming; *: Number of feature code containers.

3.1.3 Tasks

In their pilot study, Siegmund et al. [29] reported that the tasks were well balanced and had a feasible difficulty level for one experiment session, for which the participants took, on average, one hour per variability representation. Therefore, we reused the tasks in the first and second round of our replications, *i.e.*, we used the same bugs injected by the authors of the original study. Each round consisted of *five* bug-finding tasks. We introduced five similar bugs in the second subject system, relying on the task design of the original study. The first author introduced the bugs for the second replication tasks and the second author reviewed to avoid bias.

Tables 2 and 3 describe the tasks defined for each subject system used in the experiment and Table 4 elaborates on the experimental design, by mapping each task to the groups defined. In Table 4, the highlighted FH and CC marks indicate the used variability representation code version of the subject systems. In addition, Figure 5 shows a code snippet from the `RiSEEvent` subject system from which the Task 1 was

[5] Both `MobileMedia` versions are available at: `http://fosd.net/experiments`.

[6] Both `RiSEEvent` versions are available at: `https://github.com/riselabs-ufba/RiSEEventSPL-FH`.

created. The error was introduced in Line 17 by using the variable statement instead of idActivity in the delete SQL query. Table 5 provides an example of a correct answer.

```
1  package rise.splcc.repository;
2
3  // <several imports>
4
5  public class ActivityRepositoryBDR
6          implements ActivityRepository {
7  //  <70 more lines>
8      @Override
9      public void remove(int idActivity)
10             throws ActivityNotFoundException,
11             RepositoryException {
12         try{
13             Statement statement =
14                         (Statement) pm.getCommunicationChannel();
15             int i = statement.executeUpdate(
16                         "DELETE FROM Activity "+
17                         "WHERE idActivity = '"+ idActivity+"'");
18 //  <3 more lines>
19         } catch(PersistenceMechanismException e){
20 //  <8 more lines>
21 }
22 //  <300 more lines>
23 }
```

Fig. 5 Original `RiSEEvent` code snippet used in the Task 1.

Table 2 Experiment tasks defined for `MobileMedia`.

Task ID	Description
1	Instead of setting the counter to the actual value, it is set to 0.
2	A false identifier is used (`SHOWPHOTO` instead of `PLAYVIDEO`).
3	Instead of showing a list of favorite items when requested by clicking in the "View Favorites" menu, the application shows nothing.
4	Instead of showing a list of pictures ordered by the number of visualizations, they appear unordered.
5	A wrong label for deleting a picture is used, such that the check of user rights provided by the access control feature is never executed, and a user can delete a picture without according rights.

Table 3 Experiment tasks defined for `RiSEEvent`.

Task ID	Description
6	The code uses the wrong activity ID variable in the delete query.
7	The code calls `gerarCarne(...)` instead of `gerarBoleto(...)`.
8	The notification was not implemented yet.
9	The Register menu was added a second time instead of adding the Reports menu.
10	The option `JFrame.DO_NOTHING_ON_CLOSE` was used instead of option `JFrame.EXIT_ON_CLOSE`.

Table 4 Experimental designs revisited.

Subject System	Group FH	Group CC	Round
ESE Replication			
`MobileMedia`	Tasks 1–5 (FH)	Tasks 1–5 (CC)	#1
ESE Replication			
`MobileMedia`	Tasks 1–5 (FH)	Tasks 1–5 (CC)	#2
`RiSEEvent`	Tasks 6–10 (CC)	Tasks 6–10 (FH)	#3

Table 5 `RiSEEvent` Task 1 correct answer.

Question	Answer
Feature folder	ActivityMainTrack
Class	`rise.splcc.repository.ActivityRepositoryBDR`
Line of code	75
Problem	A wrong variable was used instead of correct one.
Solution	Change SQL query to use variable idActivity instead of variable statement.

3.1.4 Support Material

No additional support material was provided during the experiment sessions. The participants only had access to the task descriptions, the questions to answer, and the source code of the subject system. All these items were provided through the Prophet[7] experimental workbench [11].

Figure 6 shows the two screens of the Prophet workbench. Figure 6(a) shows on the left the tasks description window, which guided participants throughout the experiment by displaying the tasks descriptions, the appropriate place to hold the participant's answers, including those regarding the feedback form. In the other side, Figure 6(b) shows the source code inspector, which the participants used to browse the source code. In this window, the participant could use the functionalities of "local search" (①) and "global search" (②). In addition, the participant could navigate through the source project tree in the left part of the window (③) and open multiple source code files at same time, which the window shows on the right tabbed panel (④).

3.2 Preparation and Execution

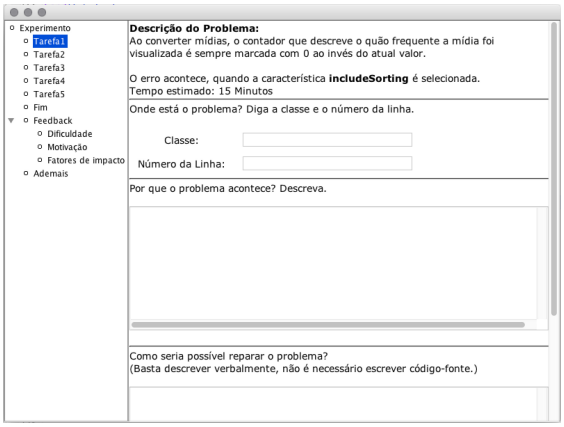
Next, we present details on the preparation and the execution of the experiments. For the purpose of preparation, we conducted a number of training sessions. For the purpose of execution, we gathered data for the characterization of participants, and asked them to perform a warm-up task, so that they could get familiar with the experiment tasks.

[7] Prophet is free and open-source and available at `https://github.com/feigensp/Prophet/`

3.2.1 Training

We carried out different training sessions to harmonize with the different levels of knowledge of the sample in each replication, ESE and RiSE. In the first replication, the participants attended a seminar about variability, followed by a practice session, including differences and peculiarities about each of the variability representations, FOP and CC. In the second replication, given that the participants were already familiar with variability representations, we developed and provided them with written material for training prior to the actual experiment session for the purpose of familiarization with the experiments object of study.

(a) Tasks description window



(b) Source code inspector

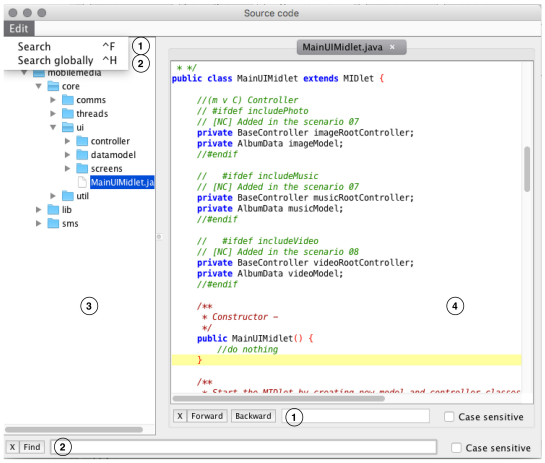


Fig. 6 PROPHET workbench: the two screens used by the participants to solve the assigned tasks.

Table 6 Participants' experience summary.

	ESE	**RiSE**
Gender	11 male and 10 female.	6 male and 6 female.
Age	Median is 33.	Median is 29.
Degree	13 *Master* students and 8 *Ph.D.* students	3 *Master* students and 9 *Ph.D.* students.
OO experience (5-point scale; 1=no and 5-high)	16 have at least mediocre level (3)	All have at least mediocre level (3).
Working experience (at least 1 year)	11 out of 21	11 out of 12

3.2.2 Participants

We used the programming-experience questionnaire developed by Siegmund et al. [30] in both replications, except for few specific questions regarding the participants' knowledge of FOP and CC for the RiSE participants. This differing treatment is justified, because the RiSE group of participants have a solid knowledge about variability, which includes the addressed variability representations.

The questionnaire serves the purpose of gathering basic information and details on the programming experience of each participant. In particular, we asked about their experience with industry-based projects, as well as their background knowledge on different programming paradigms and languages. Appendix A shows the questionnaire. Table 6 summarizes the characterization of the participants. Half of them were female in both replications. The median age was 33 in the ESE replication and 29 in the RiSE replication. In the ESE, 13 were *Master* students and 8 *Ph.D.* students, whereas in the RiSE, 3 were *Master* students and 9 *Ph.D.* students. On a 5-point Likert scale, 16 of the ESE group and all the RiSE group have, at least, a mediocre level (3 out of 5 points) of experience with Object-Oriented programming. Lastly, 52% (11 out of 21) from the ESE group and 92% (11 out of 12) from the RiSE group have, at least, one year of experience working in industry.

In line with Siegmund et al. [30], we asked the participants to compare themselves against their classmates and professional developers with 20 years of experience. Figure 7 shows the answers for both questions in both replications. The comparison with their classmates are identified by the key "Students", whereas the comparison with the developers with the key "Professionals". Figure 7(a) shows that most participants from the ESE replication see themselves, at least, as experienced as their classmates and, at most, as experienced as the professionals. Figure 7(b) shows a similar pattern in the RiSE replication, except that no one judged himself/herself as less experienced than their classmates or more experienced than professionals. The RiSE answers are not split by groups (CC or FH), because they all took place in both of them. The experience of the participants picture them as two heterogeneous groups with enough experience for the replications.

3.2.3 Example Task

All participants completed a warm-up task just before the actual experiment session in each round with two main purposes: *(i)* familiarization with the experiment environment (Prophet workbench); *(ii)* an initial contact with the source code of the subject

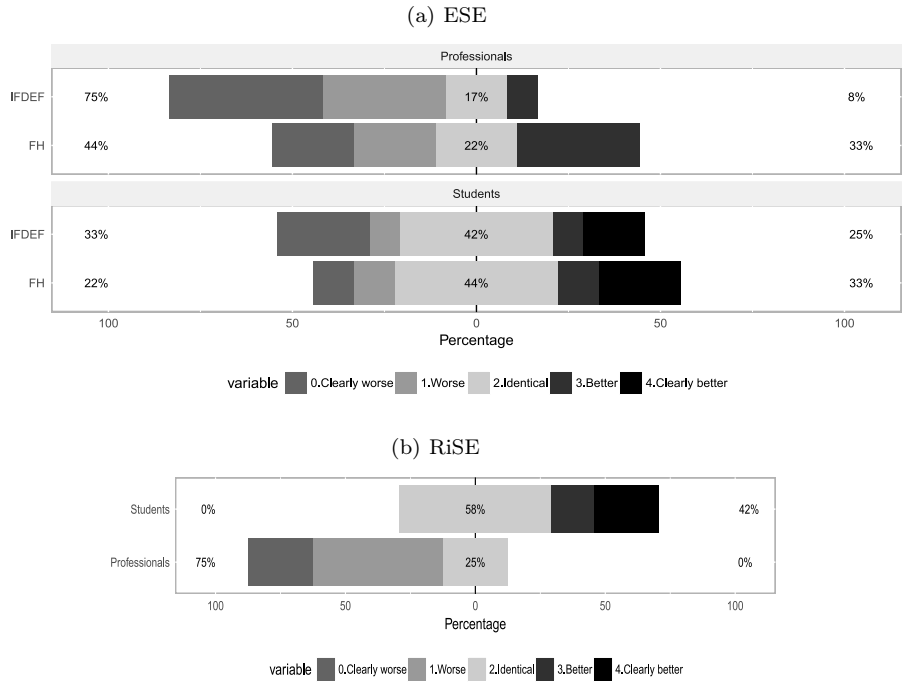


Fig. 7 Participants' programming experience (self assessment) compared to their classmates and professional developers with 20 years of experience.

system. The task required the participants (FH group) to identify across how many features a class ("PhotoViewScreen") had been refined; or (CC group) to identify across how many files a feature ("includeFavourites") had been implemented. Neither in the warm-up tasks nor in the actual experiment session the participants could execute the code.

3.3 Research Questions, Hypotheses, and Variables

In this section, we present our research questions, hypotheses, and variables. While the research questions guide our study, the hypotheses state specifically what we are going to test against the gathered data. In this sense, we associated each hypothesis with a measure to be able to test it.

Next, we describe the main research question, the sub-questions, the associated hypotheses, and how we proceeded to gather data to test them.

The main question of this investigation could be stated as follows:

RQ: How could the variability representation affect bug-finding tasks in a feature-oriented program?

This RQ was split into three sub-questions, as discussed next.

The main argument in favor of physically separated variability representations, such as FOP, is their supposed benefits in terms of improved modularity [3]. However,

there is a lack of evidence in the literature about how feature-oriented software developed with FOP could facilitate the *understanding* of problems triggering software bad functioning. We firstly attempted to investigate the following research question.

RQ$_1$: Does the variability representation affect the understanding of bug-finding tasks in a feature-oriented program?

Based on such a research question, we state the following *null hypothesis*:

H1$_0$ *The variability representation does not affect the* ***problem understanding*** *of bug-finding tasks in a feature-oriented program.*

Regarding hypothesis **H1$_0$**, we asked the participants to answer "why the problem was occurring" and "how to solve the problem" in every task. The tasks did not require the participants to write any source code, but they had to provide a textual description for each of these questions. We used a three-point scale: *complete understanding* (2), *partial understanding* (1), and *no understanding* (0). Complete understanding occurred when participants answered both questions correctly, partial understanding when either the problem or the solution was described correctly, and no understanding, when no question was answered correctly. We call this measure understanding.

In fact, it makes sense to believe that a good understanding of an issue being addressed is more likely to provide developers with means to improve the functionality of a software system. Besides, the widespread use of CC in both academia and industry [17] proves its value when discussing variability implementation. There is evidence regarding optional features that FOP (FeatureHouse included) adheres more closely to the Open-Closed principle [12, 20].

However, is it more likely that FOP yields more *correct* answers in comparison with CC directives? In order to investigate such an issue, we stated the following research question:

RQ$_2$: Does the variability representation affect the correctness of answers of bug-finding tasks in a feature-oriented program?

In order to answer RQ2, we stated the following *null hypothesis*:

H2$_0$ *The variability representation does not affect the* ***correctness of answers*** *of bug-finding tasks in a feature-oriented program.*

Regarding hypothesis **H2$_0$**, we asked the participants in each task to describe the "class", the "line of code", and, in case of a FOP task, also the "feature folder" where the error occurs. Again, we coded the answers along a three-point scale: *complete correctness* (2), *partial correctness* (1), and *no correctness* (0). Complete correctness occurred when the participants correctly answered both questions. In the case a participant only described the class (and the feature folder in the case of a FOP task) correctly, we took this as a partial correctness. Finally, no correctness was used when neither of these questions were correctly answered, or even when only the line of code was answered correctly. We call this measure correctness.

Time is a scarce resource in software development. Although some developers could produce the same correct answers with both variability representations, the differences in the understanding of an issue addressed during a maintenance task could lead teams to undesired costs. This fact makes the *response time* a factor worth to investigate.

Therefore, we attempted to investigate such an issue through the following research question:

RQ$_3$: Does the variability representation affect the time developers need to carry out bug-findings tasks in a feature-oriented program?

We suspect – by considering the arguments in favor of the benefits of modularity – that the response time needed to accomplish tasks is likely to be different, which leads us to the next *null hypothesis*:

H3$_0$ *The variability representation does not affect the* ***time that developers need to address bug-finding tasks*** *in feature-oriented software.*

Regarding hypothesis **H3$_0$**, we set up the PROPHET tool [11] to record the time spent by the participant. We then used time measured in minutes to ease the reading and the analysis.

Table 7 summarizes all measures, their descriptions, as well as their association with the addressed hypotheses.

Table 7 Measures, their descriptions, and the associated hypotheses.

Measure	Description	Hypotheses
	Independent variable	
`variability representation`	The type of variability representation used by a participant in a given task	**H1$_0$**-**H3$_0$**
	Dependent variables: Tasks measures	
`understanding`	Describes to what extent a participant understood a given task	**H1$_0$**
`correctness`	Describes to what extent a participant answered a given task correctly	**H2$_0$**
`response time`	Describes how long it took to finish a given task	**H3$_0$**

4 Results

This section discusses and analyzes the results achieved by the participants during the assigned tasks. The section is organized according to the three hypotheses associated to the research questions. For each hypothesis, we present raw data obtained from each of the three rounds of the experiment – ESE replication (Round 1) and RiSE replication (Rounds 2 and 3).

4.1 Problem Understanding

To complete the task surrounding the investigation of the first issue - *problem understanding of bug-finding tasks* -, the participants had to describe why a particular problem occurred and how this problem could be solved. Figure 8 shows raw data for the three rounds. Columns T1 to T5 show the results for each task of a given round:

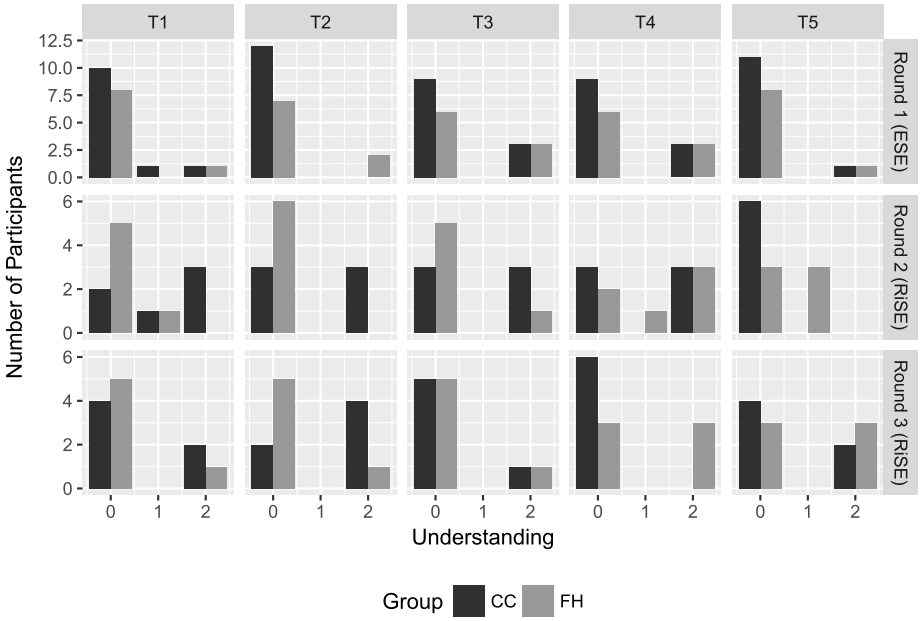


Fig. 8 Overall participants understanding of "why the problem happened" and "how to solve" in each round of the experiments. Scale: *(0)* no understanding; *(1)* partial understanding; *(2)* complete understanding.

Round 1 and 2 – first and second rows – correspond to the tasks 1 to 5 described in the Table 2; Round 3 – the last row – actually corresponds to the tasks 6 to 10 described in Table 3. This pattern repeats in Figures 9 and 10, which present the result of the $\mathbf{H2_0}$ and $\mathbf{H3_0}$. The scale ranges from *(0)* no understanding, *(1)* partial understanding to *(2)* complete understanding. In all three rounds, most participants failed to locate the problem and propose a solution.

However, in some tasks of the RiSE experiment, half the group managed to reach a complete understanding. When comparing data from Rounds 1 and 2, in which groups with different background performed the same tasks, we could state that the CC group from the RiSE replication had slightly better results, whereas the results are inconclusive for the FH groups. When comparing Rounds 2 and 3, in which the same group switched the variability representation between rounds, the results of FH groups are similar, whereas the results from Round 3 are slightly worse than Round 2 in terms of understanding.

We tested $\mathbf{H1_0}$ regarding understanding.[8] We used the *Shapiro-Wilk Test* to check whether the raw data sample had a non-normal distribution. Since we have one independent variable (`variability representation`) with two levels ("FH" and "CC") and one ordinal dependent variable (understanding), we used the two-tailed *Mann-Whitney Test* to test $\mathbf{H1_0}$. To test $\mathbf{H1_0}$, we considered all observations (*i.e.*, from the three rounds) regarding understanding from each group ("FH" and "CC") as our sample. The p-value of the test was 0.9947, that is, we cannot reject the *null hypothesis*.

[8] Data available at: `http://rise.com.br/riselabs/vicc/data/vicc3/H01_H02-data.pdf`

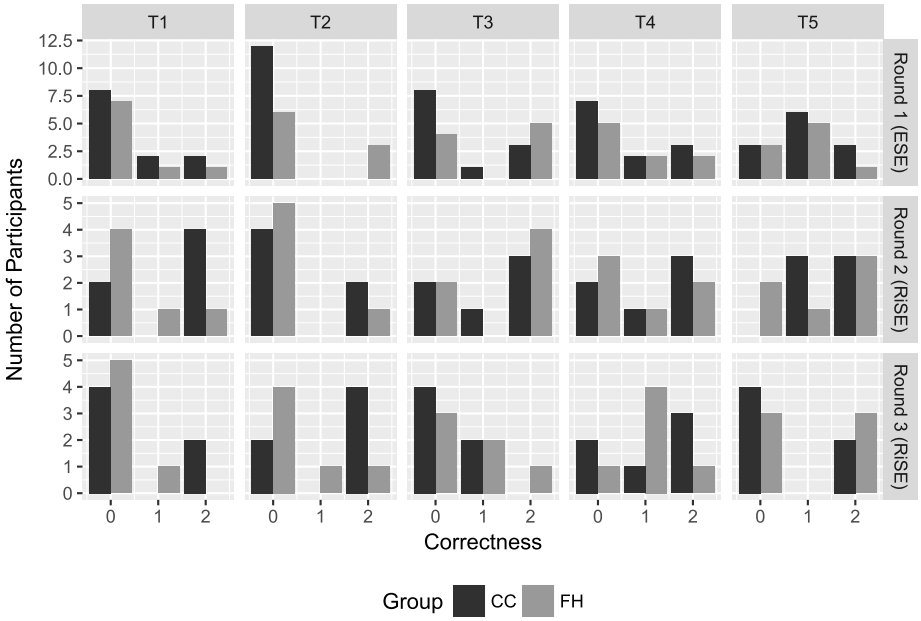


Fig. 9 Overall correctness of the description of what class, line of code, and feature folder in each round. Scale: *(0)* no correctness; *(1)* partial correctness; *(2)* complete correctness.

$H1_0$: Accepted. The evidence yielded identical populations for both groups and as such it does not support a conclusion.

4.2 Correctness of Answers

To complete the task surrounding the second issue under analysis - *correctness of answers of bug-finding tasks* -, the participants were supposed to describe the *class, line of code* and *feature folder* where they located the error. Figure 9 shows raw data obtained from the three replication rounds. The scale range comprised the following values: *(0)* no correctness; *(1)* partial correctness; *(2)* complete correctness. As expected, with a flawed understanding of the problem in all three rounds, many participants failed to locate both the class and the line containing the wrong piece of code. Nevertheless, on several occasions, participants managed to locate, at least, the class in which the problem occurred.

When comparing Rounds 1 and 2, we might observe that the participants of both groups (FH and CC) from the RiSE replication achieved proportionally better results than those who performed the same tasks in the ESE replication.

We tested $\mathbf{H2_0}$ regarding correctness. [9] We then used the *Shapiro-Wilk Test* to test whether the raw data sample follows a non-normal distribution. Besides, since we have one independent variable (`variability representation`) with two levels ("FH" and "CC") and one ordinal dependent variable for each hypothesis (correctness), we used the two-tailed *Mann-Whitney Test*. To test $\mathbf{H2_0}$, we also considered all observations

[9] Data available at: `http://rise.com.br/riselabs/vicc/data/vicc3/H03-data.pdf`

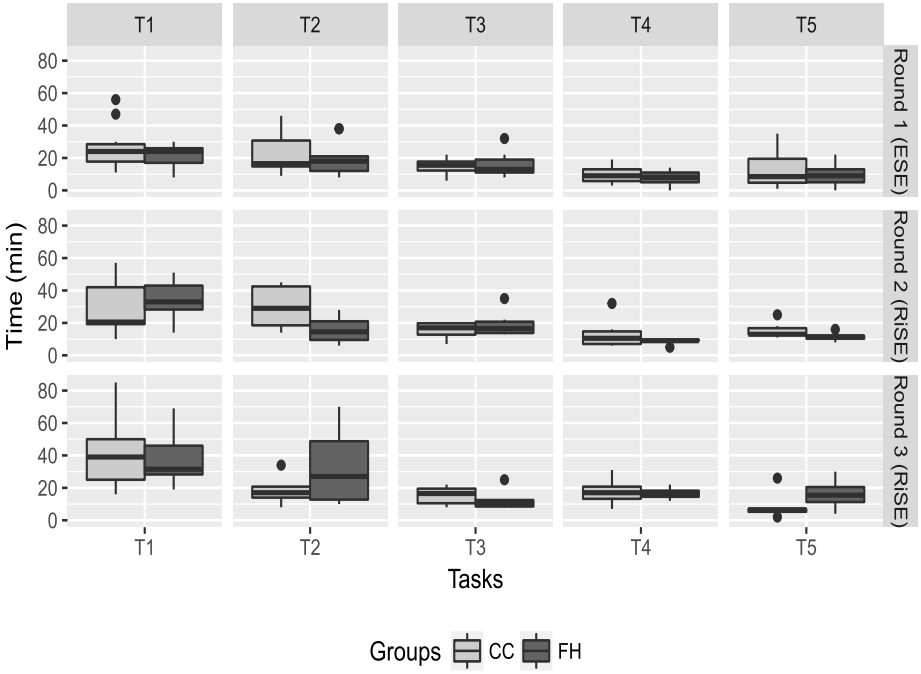


Fig. 10 Overall response time of the participants in each round.

(*i.e.*, from the three rounds) regarding understanding from each group ("FH" and "CC") as our sample. The p-value was 0.7579, we cannot reject the *null hypothesis*.

$H2_0$: Accepted. The evidence yielded identical populations for both groups and as such it might not support a conclusion.

4.3 Required time to accomplish bug-finding tasks

Figure 10 shows the raw results for the response time of the participants. All tasks accomplished rather similar time ranges for both variability representations, regardless of the experience with variability of the group (ESE and RiSE) or the addressed subject system. The response times range (considering the 1*st* and the 3*rd* quartiles) from 20 to 30 minutes in Round 1, and from 20 to 50 minutes in Rounds 2 and 3 for Tasks 1 and 2. The other tasks were performed in a smaller amount of time.

Finally, we tested $\mathbf{H3_0}$ with regard to the `response time`[10]. We used the *Shapiro-Wilk Test* as well. As we have one independent variable (`variability representation`) with two levels ("FH" and "CC") and one interval dependent variable for each hypothesis (response time), we used the two-tailed *Mann-Whitney Test* to test the $\mathbf{H3_0}$. To test $\mathbf{H3_0}$, we also considered all observations (*i.e.*, from the three rounds) regarding understanding from each group ("FH" and "CC") as our sample. The p-value was 0.6011, we cannot reject the *null hypothesis*.

[10] Data available at: `http://rise.com.br/riselabs/vicc/data/vicc3/H03-data.pdf`

$H3_0$: Accepted. The evidence yielded identical populations for both groups and it is insufficient to support a conclusion.

5 Discussion

In this section, we discuss the study results from the participants' programming experience perspective. Then, we present how the replications' results correlate to the participants' motivation and the tasks difficulty while comparing whether or not our results corroborates those from Siegmund et al.'s pilot [29]. Finally, we discuss the answers to our research questions in the light of the raw data presented in the previous sections, the participants' motivation and the difficulty of the tasks.

5.1 On the Participants' Programming Experience

This section presents a discussion of how the results correlate to the participants programming experience. In addition, we replicated he Siegmund *et al.* [30] effort to model programming experience.

5.1.1 Correlation Analysis

Siegmund *et al.* [30] strived to model programming experience of the participants with a reduced number of variables. However, they did not manage to reduce the dimensionality of the problem with no doubts remaining. In this sense, we decided to go for a conservative path and analyzed the correlation of each of the characterization variables with the number of correct answers in our experiment replications. Table 8 shows the variables actually used to measure their experience.

Table 9 shows an overview of the correlation between each of the analyzed dependent variables (`correctness`, `understanding`, and `response time`) and the data collected for each independent variable in the characterization questionnaire (Table 8) – only gray cells denote significant correlations ($p < .05$). More specifically, as we had five values for `correctness` and `understanding` (T1, T2, T3, T4, and T5), we added up the participants scores in each task (0, 1, or 2), which gave us a number between zero (0) – in case of no correct answers – and ten (10) – in case of 5 complete correct answers.

Since we correlated ordinal data, we used the *Spearman* rank correlation [1]. We can assume the following correlation categories regarding the correlation coefficient (r): no correlation ($0 \leq |r| < 0.1$); weak correlation ($0.1 \leq |r| < 0.5$); moderate correlation ($0.5 \leq |r| < 0.8$); and strong correlation ($0.8 \leq |r| \leq 1$). Most of the characterization variables taken individually have weak or no correlation with our addressed dependent variables (`correctness`, `understanding`, and `response time`). The only exception is the moderate correlation between the number of years the participant have been programming and the response time of the IFDEF group.

In contrast, Siegmund *et al.* [30] participants sample was homogeneus. Still, most of their participants fail to have, at least, half of their tasks answered correctly. Similarly, Siegmund *et al.* also found low levels of correlation among the same characterization variables and their specific dependent variables. This may indicate that none of these

Table 8 Description of the variables used for measuring the participants' programming experience. Extracted from Siegmund's *et al.* work [30].

ID	Variable	Description	Scaling
Q1	`degree`	The higher academic degree of the participant.	1:Bachelor; 2:Specialist; 3:Master.
Q3	`prog-years`	Number of many years programming.	$x \in \mathbb{Z}$
Q4	`courses-taken`	Number of courses taken so far in which the participants had to program.	$x \in \mathbb{Z}$
Q5	`java`	The participant experience in `java`.	LikertA
Q6	`c`	The participant experience in `C`.	LikertA
Q7	`haskell`	The participant experience in `Haskell`	LikertA
Q8	`prolog`	The participant experience in `Prolog`.	LikertA
Q9	`other-l`	Number of other programming languages the participant know at least to a mediocre level.	$x \in \mathbb{Z}$
Q10	`logical`	Participant experience in the Logical programming paradigm.	LikertA
Q11	`functional`	Participant experience in the Functional programming paradigm.	LikertA
Q12	`imperative`	Participant experience in the Imperative programming paradigm.	LikertA
Q13	`oo`	Participant experience in the Objected-oriented programming paradigm.	LikertA
Q14	`large-proj`	Whether the paricipant worked with a large software project or not.	0: No; 1: Yes.
Q15	`work-years`	Years working with large projects.	$x \in \mathbb{Z}$
Q17	`proj-size`	The actual size of the project in lines of code.	0: None; 1: Small; 2: Medium; 3: Large.
Q18	`students`	Programming experience self-assessment against group-mates.	LikertB
Q19	`professionals`	Programming experience self-assessment against programmer with 20 years of experience.	LikertB

ID: refers to the question identifier (Appendix A). **LikertA**: $x \in$ {0: Very inexperienced; 1: Inexperienced; 2: Mediocre; 3: Experienced; 4: Very inexperienced.} **LikertB**: $x \in$ {0: Clearly worse; 1: Worse; 2: As good as; 3: Better; 4: Clearly better.}

characterization variables alone can be a good indicator of success in code comprehension, neither the diversity of our participants be seen as the main factor of our failure to reject the null hypotheses. Therefore, we provide in the next session a factor analysis as a side contribution of this study.

5.1.2 Factor Analysis

Given the results of the correlation analysis, and in line with Siegmund *et al.* [30] goals of selecting questions to conveniently and reliably measure programming experience in different experimental settings. To cope with such a long run goal, we also used *factor analysis* [2] to extract a model of programming experience from the data. The goal is to reduce a number of observed variables to a small number of underlying *latent* variables or *factors* (*i.e.*, variables that cannot be observed directly). The factors group the variables that better describe the data under analysis relying in their inter-correlations.

Table 9 Correlations between each characterization independent variable and each dependent variable of this study.

Variable	correctness		understanding		response time	
	FH	CC	FH	CC	FH	CC
degree	-0.185	0.293	-0.143	0.347	0.424	0.185
courses-taken	0.271	-0.218	0.454	-0.282	0.262	-0.101
imperative	0.167	0.208	0.298	0.177	-0.061	0.237
c	0.303	0.376	0.351	0.392	0.029	0.254
functional	0.121	-0.037	-0.106	-0.012	0.079	-0.247
haskell	0.340	0.014	0.436	-0.132	0.188	-0.289
oo	0.189	0.261	0.407	0.305	-0.053	0.123
java	-0.011	0.242	0.198	0.306	0.078	0.165
logical	-0.084	-0.113	0.056	-0.045	0.037	-0.300
prolog	0.118	0.080	0.335	-0.143	0.032	-0.050
other-l	-0.164	0.309	0.291	0.276	0.020	0.299
prog-years	-0.016	0.360	0.179	0.329	0.160	0.504
work-years	-0.017	0.433	0.276	0.379	0.107	0.372
large-proj	0.434	0.376	0.425	0.335	0.067	0.379
proj-size	0.098	0.425	0.322	0.336	0.149	0.319
students	-0.012	0.456	0.205	0.374	-0.231	0.263
professionals	-0.123	0.416	0.237	0.271	0.027	0.317

Gray cells denote significant correlations ($p < .05$).

Table 10 shows the results of our exploratory factor analysis. The numbers in the table denote correlations or factor loadings of the variables in our questionnaire with identified factors. By convention, factor loadings that have an absolute value of smaller than .32 are omitted, because they are too small to be relevant [7].

The first factor of our analysis summarizes the variables `oo`, `java`, `other-l`, `large-proj`, `proj-size`, and `students`. It means that these variables have a high correlation amongst each other and can be described by this factor. This seems to make sense since `java` and its corresponding paradigm are similar and often taught at undergraduate courses. Besides, we conjecture that `large-proj` and `proj-size` also loads on this factor, because they explain the projects the graduate students eventually had to work with.

Additionally, since all participants are graduate students, it is normal they have to work with a number of different languages (`other-l`) other than those they learned at the university. In fact, those participants who have any professional experience with programming estimate their experience higher compared to their class mates(`students`). Except from the variables `c` and `imperative` – which were grouped by the third factor, more on this later –, this factor looks like a merge of the two most representative factors identified by Siegmund *et al.* [30].

The second factor summarizes the variables `work-years` and `prog-years`, which seems also reasonable since the number of years professional experience are intrinsically related to the amount of programming experience years. Actually, the variable `prog-years` was introduced in the questionnaire to provide means to measure the programming experience of those participants with no professional experience. This factor seems to corroborate with the fourth factor identified by Siegmund *et al.* [30].

The third factor summarizes the `imperative` and `c`. In fact, the value of the loadings of these variables are pretty similar to their loadings in the first factor. Perhaps, it would be reasonable to disregard this factor in favor of the first one with no or low representativeness loss.

The fourth factor summarizes the `courses-taken`, which represents the amount of courses taken in the university in which the participant had to program. In the Siegmund *et al.* [30] model, this variable appeared together with the programming years – our second factor. Indeed these variables are related, the fact that were grouped separately here might be explained by the differences in the size of the sample.

Table 10 Factor loadings generated by the factor analysis of each characterization independent variable of the characterization questionnaire.

Variable	Factor 1	Factor 2	Factor 3	Factor 4
degree				
courses-taken				0.662
imperative	0.598		0.682	
c	0.601		0.620	
functional				
haskell				0.378
oo	0.733		0.427	
java	0.676		0.379	
logical				0.370
prolog			0.414	0.356
other-l	0.664	0.545		0.380
prog-years	0.540	0.750		
work-years	0.560	0.781		
large-proj	0.990			
proj-size	0.885			
students	0.623		0.546	
professionals	0.440	0.566		

Only 4 factor are shown due p= .032 for hypothesis test of sufficiency. Gray cells denote main factor loadings.

Table 11 shows correlations between the inner product of the factor loadings and the independent variable observations of that factor and each dependent variable of the study. We used the Spearman correlation; the significat values (*i.e.*, $p < .05$) are shadowed. None of the factors yielded strong correlations, which means either the factor analysis was unable to generate good predictors for our dependent variables or the number of observations was insufficient to produce strong correlations, which is often an issue with factor analysis.

5.2 On the Participants' Motivation, Task' Difficulty, and Results

We asked the participants to rate their motivation while performing each task and their perception of the difficulty of each task after they finished the experiment activities. We decided to present both motivation and difficulty, because we believe both

Table 11 Correlations between each factor and each dependent variable of this study.

Variable	correctness		understanding		response time	
	FH	CC	FH	CC	FH	CC
Factor 1*	0.072	0.381	0.370	0.355	0.279	0.279
Factor 2*	0.046	0.496	0.254	0.442	0.484	0.484
Factor 3*	0.227	0.294	0.346	0.279	0.265	0.265
Factor 4*	0.271	-0.218	0.454	-0.282	-0.101	-0.101

Gray cells denote significant correlations ($p < .05$). * This variable is actualy the inner product of the factor loadings and the independent variable observations of that factor.

are related, since harder tasks might affect the participants' motivation. We used a five-point Likert-scale [21] to code their answers. The scale range comprised the following values: *(0)* Very difficult/unmotivated; *(1)* Difficult/unmotivated; *(2)* Normal difficulty/motivation; *(3)* Easy/motivated; and *(4)* Very easy/motivated. Figure 11 shows the participants' feedback regarding difficulty, and Figure 12 shows the results for motivation. In addition, we use quotes from the original study [29] to guide our discussion.

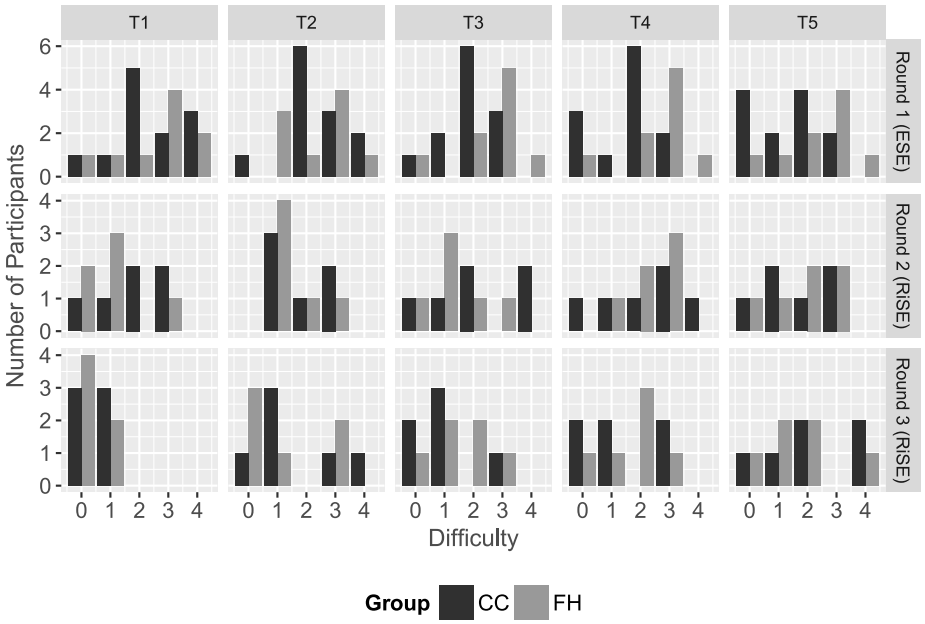


Fig. 11 Overall perception of difficulty of the participants in each round. Scale: *(0)* Very difficult; *(1)* Difficult; *(2)* Normal difficulty; *(3)* Easy; and *(4)* Very easy.

> "Regarding the opinion of participants, we found a tendency that the CC group found the tasks easier to solve, except for Task 2". [29]

In Round 1, both groups of participants' responses of the ESE replication had a similar tendency. Both groups responses rated the tasks difficulty as "normal" or "easy".

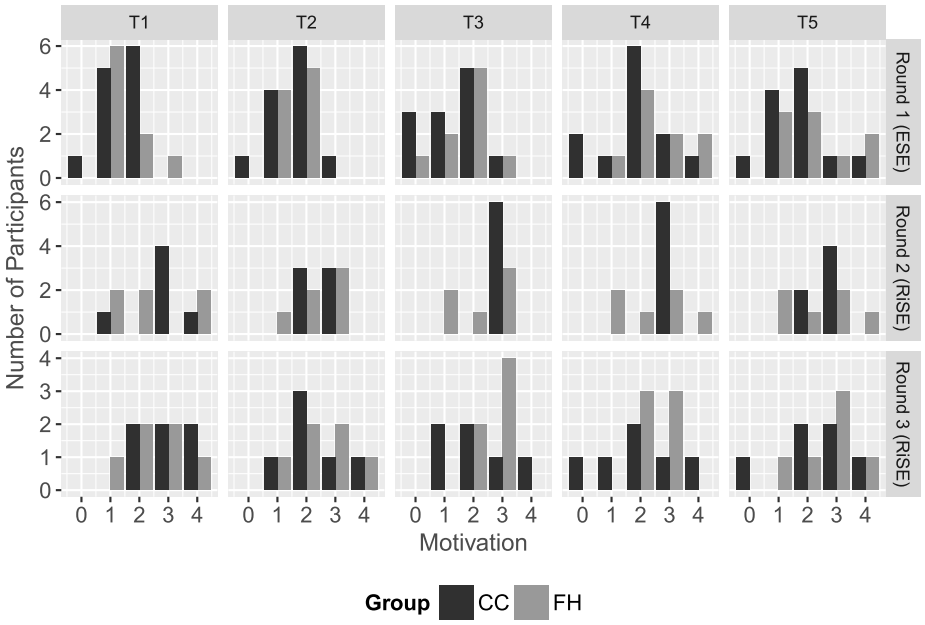


Fig. 12 Overall feeling of motivation of the participants in each round. Scale: *(0)* Very unmotivated; *(1)* Unmotivated; *(2)* Normal motivation; *(3)* Motivated; and *(4)* Very motivated.

Thus, the FH group answers did not follow the observations of Siegmund et al. [29]. In Round 2, where the RiSE replication participants performed the same tasks of Round 1, their responses were closer to the results of Siegmund et al. [29]. For the FH group, the three first tasks were mostly (very) difficult, whereas only Task 2 was rated as difficult by the CC group, which shows that the FH group felt that the tasks were harder than the CC group. In Round 3, in which the participants targeted a different system, their responses seemed to be homogeneous, regardless of the variability representation, rating the tasks mostly (very) difficult. This indicates that the RiSE group might have an equivalent background knowledge to those participating in the original study, whereas the ESE group show signs of having a deficient background.

> "For motivation, there is an observed trend that participants of the CC group are more motivated to solve a task. Such an observation might be caused by the fact that two participants of the FH group were unhappy to be in that group (as they told us). Therefore, the FH version appears more difficult to participants and they did not like it. This can affect their performance, such that they work slower". [29]

With regard to motivation, both groups in all three rounds had equivalent motivation, with the exception of Tasks 3 and 4 in Round 3. In Round 1, most ESE group participants were feeling unmotivated or normal motivation, whereas in the Rounds 2 and 3, the RiSE participants were feeling mostly normal to motivated, with the exception of the Task 3 in Round 3 – in which they felt unmotivated. Some of the participants from both groups mentioned they felt their motivation increase when they started to get familiar with the code they were working on, and the opposite effect as they stepped through the tasks with difficulty to answer them. Although the ESE group participants (Round 1) feeling of easy tasks during the experiment, they felt unmotivated, which

make us reinforce the claim of deficient background knowledge. The RiSE participants (Rounds 2 and 3) judged the tasks mostly hard to solve, but remained motivated during their tasks. We believe this happened because of their familiarity with the context of the experiment activities.

Next, we present Spearman correlations among the participants' *motivation*, tasks' *difficulty* perception and the results of our replications in terms of three dependent variables (correctness, understanding, and response time). Table 12 shows in the first row the correlations between participants' *motivation* (CP09) and the experiment dependent variables, whereas the correlations between the *difficulty* (CP01, CP06) perception and each variables are shown in the second row. The correlation coefficients indicate significant ($p < .05$) moderate correlations between the participants' motivation and correctness or understanding. This makes sense, since the motivation of the participants may keep them focused on their activity and helps them extract most of their abilities to transform in the effort needed to successfully complete the tasks.

Table 12 Correlations between the participant motivation and their feeling of difficulty and each dependent variable of this study.

Variable	`correctness`		`understanding`		`response time`	
	FH	CC	FH	CC	FH	CC
`motivation` (CP09)	0.589	0.705	0.758	0.676	0.136	0.359
`difficulty` (CP01, CP06)	0.135	0.322	-0.109	0.320	-0.254	0.177

Gray cells denote significant correlations ($p < .05$).

5.3 Analysis of findings

The evidence gathered in our empirical evaluation is preliminary due to the exploratory nature of the study. However, it may offer an opportunity to rethink the way researchers could carry out future experiments. They highlighted the peculiarities of the experimentation on the influence of differences of variability representations on program comprehension, in particular those related to bug-finding tasks. For instance, it is hard to assure equivalent knowledge of both variability representations under evaluation by training the participants prior to the experiment session. We are aware of the importance of assure that results are comparable, in particular, because FeatureHouse is an emerging technology that most developers were about to have their first contact during the experiment activities. Next, we discuss the main findings and their implications.

– The variability mechanism does not seem to have any effect on the overall comprehension of the source code and the difficulty of the maintenance tasks arises from other factors than how the variability is performed. In addition, the fact that both partial correctness and understanding had few occurrences in any of the rounds also reinforces the claim that without a complete understanding, the correct answer can be compromised. In fact, high quality and proper debugging tools seem to play an important role for program comprehension of unfamiliar code, especially when it comes to bug-finding tasks. The Prophet tool is rather limited to global and local

searches and does not allow the execution or compilation of the code, which can be one of the reasons of participants failure.
- Results of correctness and understanding are not clear enough, so we could not draw any conclusions on whether the strong foundations on variability of the RiSE group played any the difference in achieving better results than the participants of the ESE group. The raw data about the participants' perception of the tasks' difficulty might explain why they failed to completely understand the code. While participants with superficial experience with software variability (ESE Group) mostly found tasks easy, the participants with strong foundations of variability (RiSE Group) were not that excited. Our claim is that, although most participants fail to completely understand, those with the specific background required to the maintenance task are the ones who had the proper commitment to the experiment. This fact may point to the importance of background knowledge for future replications. We conjecture that a deep understanding of each of the particular mechanism is key to find the origin of a problem in unfamiliar code while addressing change requests.
- Despite a decrease of the response time required to finish follow-up tasks, we observed neither increasing nor decreasing trends in understanding and correctness levels. We did observe the increase of the motivation after the initial tasks for the RiSE participants, which can explain the decrease of the range of time spent to answer the final tasks. Surprisingly, we found a weak correlation between motivation and response time. We conjecture that the low motivation of the ESE participants influenced this result. The differences in the time range between the beginning and ending tasks may arise from to the time needed for the participants to familiarize with the source code and it could also be associated to the lack of motivation in the first tasks.

6 Threats to Validity

In this section, we discuss possible threats to the validity of this empirical study. Presenting detailed information is a way to contribute to further research and replications of this study [33], which may be built upon the results presented herein. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion validity*.

6.1 External validity

We identified some threats that may limit the ability to generalize our results. For example, the study was carried out in an *in-vitro* setting, which means a sample selected by convenience. The issue here is that conclusions may be impossible to generalize the findings to professionals, although there is evidence showing students and professionals perform similarly when they apply a development approach in which they are inexperienced [22]. In this sense, we recruited students with different experience levels, as shown by the questionnaire applied prior to the experiments. Although some may argue that the different background of the two groups (ESE and RiSE) used in the replications would threat the joint analysis of their data, we could have a significant and heterogeneous sample.

Moreover, the size of the subject systems we used to perform the tasks of the experiments could be questioned. We are aware of such a threat, however, regardless

of the systems' size, these kinds of tasks (*i.e.*, software maintenance tasks) will always be formulated in a reduced scope. This enables performing controlled experiments in a short amount of time with minimized fatigue effect. Moreover, our subject systems are not trivial, but not too complex. Therefore, we believe this fact put our systems in a good position, at least to a certain extent, towards generalizing the findings.

6.2 Internal validity

There are possible threats that may happen without the researcher's knowledge affecting individuals from different perspectives, such as *(i)* maturation and learning effects and *(ii)* the experiment instrumentation. These threats to internal validity were mitigated by choosing different features for each task, as well as by controlling communication among the participants in all the rounds.

Again, as discussed regarding external validity, the choice of subject systems and participants might also threaten internal validity, mainly due to: *(i)* the low number of students involved in the study; *(ii)* the high variability of the subjects (Master, Ph.D., and different working experience), and *(iii)* that fact that the experimental objects differ in size. However, it is difficult to get both the ideal groups of professionals for participating in the experiment and the the proper size subject systems. Besides, the emerging nature of the feature-oriented programming still causes an imbalance regarding the background knowledge of the participants using either such technology and the conditional compilation. With regard to the size of the subject systems, the artifacts needed for well-designed experiment tasks (*e.g.*, feature model, documentation, and with equivalent versions written with each technology) are usually unavailable. Thus, to the best of our knowledge, we used suitable participants and systems samples for our case.

Confounding constructs may affect our findings. For instance, the motivation and the difficulty of each task might have affected the participants' perception of the influence of the variability representations. The issue here is that we cannot assure that the tasks were not too hard to be solved by the participants. We believe this threat was mitigated in our study by the attempt to fine tune the difficulty with pilot studies and using the same artifacts from previous experiments.

6.3 Construct validity

Construct validity refers to the fact that the construct to measure was not operationalized correctly. Perhaps the way we coded the understanding and correctness regarding the answers of the participants may be a potential threat, because of the used pseudo-ordinal measure scale. There is a fact that only a few partial understandings/correctness might be a sign of such threat. As a participant's understanding cannot be measured easily by others, we attempted to mitigate the threat by measuring the behavior (problem identification and solution). In addition, the approach of mixing Ph.D and Master students can constitute a threat, as the former are theoretically more experienced than the latter.

6.4 Conclusion validity

Our discussions were based on a small sample, which limits the power of statistical tests to reveal a true pattern in the data. For instance, our Finding 1 could be strengthened with a bigger sample. We mitigated this threat by employing well defined measures, *i.e.*, problem identification and solution, and response time to conduct our analysis. Nevertheless, our results could be affected by a Type-II error due to low statistical power. Another observed threat is the fact that the results of the recruited groups (ESE and RiSE) might not be comparable, since they have different backgrounds. We are aware of this and tried to alleviate it by comparing the results both inter and inner groups. Thus, we argue that we have sufficiently controlled this threat.

7 Related Work

There is little published work [9, 10, 23, 24, 29] addressing the differences yielded by different variability representations – such as annotations and feature-oriented programming – on program comprehension.

Back in 2009, Feigenspan et al. [9] presented a preliminary guide on how to compare program comprehension in FOSD empirically. They highlighted the importance of maintaining the confounding variables constant and also the importance of reducing the number of independent variables for a realistic comparison. Later, Siegmund et al. [29] firstly investigated the influence of CC and FOP on program comprehension by conducting a pilot study comparing program comprehension of virtually and physically separated concerns, which we refer to perform our experiments. This study brought preliminary evidence to the field and encouraged researchers to replicate the pilot, as well as to extend their findings. In addition, the survey of confounding variables in controlled experiments from Siegmund et al. [31] and the decision drivers exploratory study from Santos and Mendonça [25] also inspired our work.

Recently, Santos et al. [23, 24] extended the corpus of evidence with two different studies. First, they addressed the influence of the existing differences between two `JavaScript` approaches to implement variability in feature location tasks [23]. They used two open-source systems with equivalent versions implemented with a "standard" (basically control-flow statements) and our RiPLE-HC (feature-oriented) approaches. The empirical evaluation yielded evidence of reduced effort in feature location, and benefits when introducing systematic reuse aspects in `JavaScript` code. Furthermore, Santos et al. [24] conducted a focus group aiming at identifying a set of aspects influencing the feature-oriented software comprehension. They identified and enumerated several benefits (*e.g.*, traceability among feature implementation and flexibility provided by annotations) and drawbacks (e.g., too much duplicated classes due feature refinements and reinforced the long term known #ifdef-hell [32]) of both approaches (CC and FOP).

Another set of related studies has focused on program comprehension from the perspective of a given particular variability implementation mechanism. They mostly studied the comprehension of feature-oriented software based on CC [10, 16, 18, 19, 26]. Siegmund et al. [10] presented a family of controlled experiments regarding the effectiveness of the use of background colors to help with the maintenance of software systems using CC. The main question addressed was whether the use of colors instead

of and in addition to `#ifdef` directives could improve program comprehension in configurable systems. Background colors showed potential to improve program comprehension independent of size and programming language of the project. Schulze et al. [26] and Malaquias et al. [16] addressed the importance of the discipline of annotations, but reaching contrasting results with the most recent one claiming the undisciplined annotations should not be neglected.

Melo et al. [18] carried out a controlled experiment to quantify the influence of the degree of variability in bug-finding tasks. Their results show the speed of bug finding decreases linearly with the degree of variability, while effectiveness of finding bugs is relatively independent of the degree of variability. Moreover, they discovered that the task of participants to identify the exact set of affected configurations appears to be harder than finding the bug in the first place. We addressed this kind of task in our investigation, but using two different ways of realizing variability. While they used only CC, we also considered FOP. Moreover, Melo et al. [19] showed that the presence of variability correlates with an increase in the number of gaze transitions between definitions and usages for fields and methods.

Finally, there are also studies not related to variability mechanisms. For instance, Maalej et al. [15] conducted an exploratory study on the identification of strategies which developers use to comprehend code, tools supporting their work, important knowledge during bug-finding tasks, channels they share such knowledge, and the problems faced in real-world experiences. They found a gap between theory and practice. They raised questions about the usefulness of comprehension tools suggested by research, because none of the developers mentioned the use of visualization, metrics, or concept-location tools in practice. They also identified differences in the understanding of program comprehension among developers and researchers. While researchers have the "comprehension" aspect in the core of maintenance activity and try to systematize the whole process, developers avoid them whenever it is possible, and attempt to focus only on the expected output.

8 Concluding Remarks and Future Work

We addressed the lack of empirical evidence on the difference of the influence of traditional (CC) and emerging (FOP) variability representations. To the best of our knowledge, there was only one pilot study carried out so far addressing this particular setting. The results of our study allow us to draw the following conclusions:

- Indicators of no statistical difference regarding the *(i)* required effort to understand and to *(ii)* find flawed source code, and *(iii)* regarding the time required to finish bug-finding maintenance tasks while using FOP and CC paradigms;
- There is a moderate correlation between the motivation of programmers and their efficiency to perform bug-finding tasks correctly.
- A supplementary Web site with the experimentation artifacts of the carried experiments.[11]

We are aware that these findings are far from completely describing the relationship among variability representations chosen and the different confounding parameters associated to comprehension tasks in software engineering. As our study was limited

[11] http://rise.com.br/riselabs/vicc/

to bug-fixing, it is also important to address other maintenance activities, like addition and/or update of software components. Additionally, it would be interesting to observe professional programmers working with CC and FOP code to observe how they work with different variability mechanisms.

Acknowledgment

This work is partially supported by INES (grant CNPq/465614/2014-0), FAPESB (grants BOL0820/2014 and JCB0060/2016), CAPES (grant PDSE 99999.007061/2015-03), the German Research Foundation (AP 206/6, SI 2045/2/1), and the Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B).

References

1. T. W. Anderson and J. Finn. *The new statistical analysis of data.* Springer, 1996.
2. T. W. Anderson and H. Rubin. Statistical inference in factor analysis. In *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability, Volume 5: Contributions to Econometrics, Industrial Research, and Psychometry*, pages 111–150, Berkeley, Calif., 1956. University of California Press.
3. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, Germany, 2013.
4. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
5. S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *Transactions on Software Engineering*, 39(1):63–79, 2013.
6. D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Lines Conference*, pages 7–20, Berlin, Germany, 2005. Springer.
7. A. B. Costello and J. W. Osborne. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research & Evaluation*, 10:173–178, 2005.
8. P. A. da Mota Silveira Neto, T. L. de Santana, E. S. de Almeida, and Y. C. Cavalcanti. Rise events: A testbed for software product lines experimentation. In *Proceedings of the International Workshop on Variability and Complexity in Software Design*, pages 12–13. ACM, 2016.
9. J. Feigenspan, C. Kästner, S. Apel, and T. Leich. How to compare program comprehension in fosd empirically: An experience report. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 55–62. ACM, 2009.
10. J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.
11. J. Feigenspan, N. Siegmund, A. Hasselberg, and M. Köppen. PROPHET: Tool infrastructure to support program comprehension experiments. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement.* IEEE, 2011.
12. G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. A. Maia. On the use of feature-oriented programming for evolving software product lines— A comparative study. *Science of Computer Programming*, 93, Part A(1):65 – 85, 2014.
13. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, pages 261–270. ACM, 2008.

14. F. N. Gaia, G. C. S. Ferreira, E. Figueiredo, and M. A. Maia. A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Science of Computer Programming*, 96, Part 2(1):230 – 253, 2014. Selected and extended papers of the Brazilian Symposium on Programming Languages 2012.
15. W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the comprehension of program comprehension. *Transactions on Software Engineering and Methodology*, 23(4):31:1–31:37, 2014.
16. R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. The discipline of preprocessor-based annotations - does #ifdef TAG n't #endif matter. In *Proceedings of the International Conference on Program Comprehension*, pages 297–307, 2017.
17. F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 495–518. Dagstuhl Publishing, 2015.
18. J. Melo, C. Brabrand, and A. Wąsowski. How does the degree of variability affect bug finding? In *Proceedings of the International Conference on Software Engineering*, pages 679–690. ACM, 2016.
19. J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski. Variability through the eyes of the programmer. In *Proceedings of the International Conference on Program Comprehension*, pages 34–44. IEEE, 2017.
20. B. Meyer. *Object-oriented software construction*, volume 2. Prentice Hall, New York, 1988.
21. L. Rensis. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):5–55, 1932.
22. I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *37th IEEE International Conference on Software Engineering (ICSE)*, pages 666–676. IEEE, 2015.
23. A. R. Santos, I. C. Machado, and E. S. Almeida. RiPLE-HC: Javascript systems meets SPL composition. In *Proceedings of the International Systems and Software Product Line Conference*, pages 154–163. ACM, 2016.
24. A. R. Santos, I. C. Machado, and E. S. Almeida. Aspects influencing feature-oriented software comprehension: Observations from a focus group. In *Proceedings of the Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 1–10. ACM, 2017.
25. J. A. M. Santos and M. G. de Mendonça. Exploring decision drivers on god class detection in three controlled experiments. In *Proceedings of the Annual ACM Symposium on Applied Computing*, SAC '15, pages 1472–1479. ACM, 2015.
26. S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pages 65–74. ACM, 2013.
27. F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. The role of replications in empirical software engineering. *Empirical Software Engineering*, 13(2):211–218, 2008.
28. J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering*, pages 378–389. ACM, 2014.
29. J. Siegmund, C. Kästner, J. Liebig, and S. Apel. Comparing program comprehension of physically and virtually separated concerns. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 17–24. ACM, 2012.
30. J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.
31. J. Siegmund and J. Schumann. Confounding parameters on program comprehension: A literature survey. *Empirical Software Engineering*, 20(4):1159–1192, 2014.
32. H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 185–197. USENIX, 1992.
33. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Germany, 2012.

A Questionnaire

Table 13 shows the questionnaire used for the characterization of the programming experience.

Table 13 Questionnaire for measuring programming experience of the participants. Extracted from Siegmund's et al. work [30]

ID	Question
Q1	Higher Academic Degree
Q2	Course of Study
Q3	For how many years are you programming?
Q4	How many courses were you enrolled in which you had to program?
Q5	How experienced are you with Java?
Q6	How experienced are you with C?
Q7	How experienced are you with Haskell?
Q8	How experienced are you with Prolog?
Q9	In how many more programming languages are you experienced at least to a mediocre level?
Q10	How experienced are you with the Logical programming paradigm?
Q11	How experienced are you with the Functional programming paradigms?
Q12	How experienced are you with the Imperative programming paradigms?
Q13	How experienced are you with the Objected-oriented programming paradigms?
Q14	Have ever worked on one or more large programming projects in a company or at the university or are you currently working on a large programming project?
Q15	Since when are you working in a company/at the university on larger projects?
Q16	In which domain were/are those projects?
Q17	How many lines of code did these projects usually have?
Q18	How do you estimate your programming experience with other students of this course?
Q19	How do you estimate your programming experience with programmers that have 20 years of experience?