Universidade Federal da Bahia
Instituto de Matemática

Programa de Pós-Graduação em Ciência da Computação

INVESTIGATING FEATURE-ORIENTED SOFTWARE COMPREHENSION

Alcemir Rodrigues Santos

TESE DE DOUTORADO

Salvador, Bahia – Brasil
Agosto, 2017

ALCEMIR RODRIGUES SANTOS

INVESTIGATING FEATURE-ORIENTED SOFTWARE COMPREHENSION

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Eduardo Santana de Almeida
Co-orientador: Prof. Dr. Ivan do Carmo Machado

Salvador, Bahia – Brasil
Agosto, 2017

Modelo de ficha catalográfica fornecido pelo Sistema Universitário de Bibliotecas da UFBA para ser confeccionada pelo autor

```
   Santos, Alcemir Rodrigues
   Investigating Feature-Oriented Software Comprehension /
Alcemir Rodrigues Santos. -- Salvador, 2017.
   195 f. : il

   Orientador: Eduardo Santana de Almeida.
   Coorientador: Ivan do Carmo Machado.
   Tese (Doutorado - Programa de Pós-Graduação em Ciência da
Computação) -- Universidade Federal da Bahia, Instituto de
Matemática, 2017.


   1. Variability. 2. Software Maintenance. 3. Program
Comprehension. 4. Feature-Oriented Software Development. I.
Almeida, Eduardo Santana de. II. Machado, Ivan do Carmo. III.
Título.
```

ALCEMIR RODRIGUES SANTOS

“INVESTIGATING FEATURE-ORIENTED SOFTWARE COMPREHENSION”

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.

Salvador, 11 de agosto de 2017.

ACKNOWLEDGEMENTS

It is of no surprise to anyone how hard it is to get to such an accomplishement of a Ph.D. degree in life. Most do not get even a try. That is why we need so much people around us. This English piece is for all those international ones I got to know back in Germany and in the conferences a I took part during this period, whose in the name of Professor *Sven Apel* I thank you all! Viellen Dank! Gracias! Merci! You all have somehow contributed to such a step forward.

Repito, agora em Português, para os que estão aqui mais perto. Não é surpresa para ningúem o quão difícil é alcançar o grau de Doutor durante a vida. A maioria não tem nem ao menos a oportunidade de tentar. E é por isto que a gente precisa de tanta gente ao nosso redor. Agora agradeço à aqueles que, mesmo sem muita ciência do que eu buscava durante todo esse tempo, me apoiaram e trouxeram palavras de conforto durante todo o período: meu pai, *Almir José dos Santos* e minha mãe, *Maria Eunice Rodrigues dos Santos*. A quem, eu agradeço em nome de ambas suas famílias. Este título também é de vocês.

Agradeço ainda aos companheiros de laboratório, que muitas vezes me ajudaram durante todo o processo e tempos de UFBA e RiSELabs: Raphael, "Belle", Tassio, Mário, Jaziel, Magno, Jonatas, Leandro, Alberto, Amancio, Paulo, Crescencio, Renato, Bruno, Simone, Glaucya, Iuri, Renata, Larissa, Matheus, Rose, Karla, Anna, Daniel, Taijara e demais. Aos orientadores, *Eduardo Santana de Almeida* e *Ivan do Carmo Machado*, meu muito obrigado por todos os conselhos, sugestões, tempo e atenção dispensados! Vocês todos também contribuíram para esse trabalho.

Por fim, mas não menos importantes, todos amigos e amigas que caminharam comigo durante esta jornada, meu muito obrigado! Sem os momentos de diversão, festas, treinos e descontração com vocês, nada disso teria sido possível. Vocês todos também contribuíram para esse trabalho. Muito obrigado!

RESUMO

Atualmente, uma gama de técnicas e ferramentas para a implementação de variabilidade estão disponíveis e vem gradualmente sendo utilizadas para o desenvolvimento de sistemas de software grandes e complexos. Algumas delas alcançaram um alto nível de popularidade na indústria, como a compilação condicional, outras ainda residem o ambiente acadêmico, como Programação Orientada à Características (FOP). Pesquisadores têm investigado as limitações de cada uma delas em busca de facilitar a adoção e de seu uso.

No entanto, ainda não existe apoio à implementação de variabilidade em alguns domínios –*e.g.*, sistemas baseados em `JavaScript`– soma-se a isto a falta de evidências sobre o impacto das differenças e similaridades de tais técnicas na compreensão dos programas escritos e consequentemente no esforço que estas demandam dos desenvolvedores para a conclusão de suas tarefas de manutenção.

Esta tese contribui em ambas direções. Primeiro, apresentamos uma estratégia para engenharia de linhas de produtos baseada em composição híbrida (RiPLE-HC). Híbrida pois mescla abordagens *composicionais* e *anotativas* para implementar variabilidade. Segundo, construímos um corpo de evidências sobre compreensão de programas com variabilidade, incluindo fatores que facilitam e dificultam a compreensão de sistemas equanto utilizando-se de técnicas representativas de ambos os grupos, o popular e o emergente.

Na primeira direção, conduziu-se estudos preliminares da viabilidade e scalabilidade da abordagem RiPLE-HC, tanto no ambiente industrial quanto acadêmico. Na segunda direção, conduziu-se uma família de experimentos – chamada de Compreensão da Compreensão da Implementação de Variabilidade (VICC). Considerou-se tanto estudos quantitativos quanto qualitativos na família VICC, à saber três *quasi*-experimentos (VICC1-3) e um grupo focal (VICC4). Os estudos VICC consideram duas linguages de programação (`Java` e `JavaScript`) e uma representação de variabilidade representativa dentre as baseadas em anotação e composição para o desenvolvimento de software orientado à caracteristicas (FOSD). VICC1 utilizou-se de tarefas de localização de interesses, enquanto VICC2 e VICC3 utilizaram-se de tarefas de correção de problemas, e VICC4 buscou identificar fatores de influência na comprehensão de programas.

Embora os participantes do grupo focal tenham destacado os benefícios da FOP para manutenção, os *quasi*-experimentos não produziram evidencias estatísticas significativas destas vantagens para quaisquer das representações de variabilidade equanto os participantes executavam tarefas de manutenção. Adicionalmente, encontrou-se que que engenheiros de software podem perceber o efeito de parâmetros de confusão de forma diferente dependendo da representação de variabilidade utilizada.

Palavras-chave: FOSD; Variabilidade; Compreensão de Programas; Manutenção de Software; `Java`; FeatureHouse; `JavaScript`; RiPLE-HC.

ABSTRACT

A number of techniques and tools to handle variability are available and they have been increasingly applied in the development of large and complex software systems. Some of them have reached high levels of popularity in industry, such as conditional compilation, whereas some are mostly known in academia, such as Feature-Oriented Programming (FOP). Researchers have addressed the existing drawbacks of both in order to improve adoption and ease their use.

However, there is still a lack of support to variability implementation in some domains – *e.g.*, `JavaScript` -based systems – and also a lack of understanding of the impact of the different ways to implement variability on program comprehension and consequently on the effort they demand from developers, so they could successfully accomplish the assigned maintenance tasks.

This thesis contributes in both facets. First, we present the RiSE Product Lines Engineering approach based on Hybrid Composition (RiPLE-HC) to implement variability in `JavaScript`-based systems. By hybrid composition, we mean the blending of *compositional* and *annotative* approaches to implement variability. Second, we built an evidence corpus on program comprehension in the presence of variability, including factors easing and hindering program comprehension in software systems using representative approaches from both groups of techniques, the popular and the emerging ones.

In the first facet, we carried out a preliminary evaluation of the viability and scalability of the RiPLE-HC approach both, in industry and academic settings. In the second facet, we carried out a family of experiments – named Variability Implementation Comprehension Comprehension (VICC). We considered quantitative and qualitative studies in the VICC family, namely three *quasi*-experiments (VICC1-3) and a focus group (VICC4). VICC studies considered two programming languages (`Java` and `JavaScript`) and a representative variability representation representing either the annotative or the compositional approaches for Feature-Oriented Software Development (FOSD). VICC1 addressed the concept location tasks, while VICC2 and VICC3 addressed bug-fixing tasks, and VICC4 addressed the influence factors on program comprehension.

Although the participants of the focus group highlighted the benefits of the FOP for maintenance, the *quasi*-experiments yielded no significant statistical difference regardless of the variability representation while addressing maintenance tasks. Additionally, we found that software engineers may perceive confounding parameters differently depending on the used variability representation.

Keywords: FOSD; Variability; Software Maintenance; Program Comprehension; `Java`; FeatureHouse; `JavaScript`; RiPLE-HC.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DBLP	Computer Science Bibliography	22
FOP	Feature-Oriented Programming	16
FOPLE	Feature-Oriented Product Line Engineering	14
FOSD	Feature-Oriented Software Development	3
IDE	Integrated Development Environment	65
KLOC	Thousand Lines of Code	49
LOC	Lines of Code	49
RiPLE-HC	RiSE Product Lines Engineering approach based on Hybrid Composition	4
SDLC	Software Development Life Cycle	18
SoC	Separation of Concerns	16
SPL	Software Product Line	3
UI	User Interface	19
VICC	Variability Implementation Comprehension Comprehension	61

PART I

OVERVIEW

Chapter
1

INTRODUCTION

Since the early stages to establish the Software Product Line (SPL) engineering field, several work proposed means to improve source code modularity (*e.g.,* "feature-oriented" and "aspect-oriented" software development) [1, 2, 3]. The research community has tried to achieve modularity as a prominent strategy to mitigate known conditional compilation (also known as preprocessor-based implementation) issues, such as the increased occurrence of crosscutting concerns and code obfuscation [4, 5]. These problems originated considerably by the use of `C` preprocessor-like (`#ifdef`) annotations to enable the modification of software behavior depending on the features selected. Nevertheless, conditional compilation has been the widely accepted strategy to implement variability in software systems, despite its proven drawbacks [6]. However, both the ease of use and flexibility provided by the `#ifdef` annotations, together with usually available robust tool support, it is possible to develop variable systems sufficiently sheltered from inconsistences, even in large software systems such as the Linux kernel [7].

Feature-Oriented Software Development (FOSD) is the term coined to refer to the paradigm used for the construction, customization, and synthesis of large-scale software systems relying on features [8]. A *feature* satisfies a requirement while performing design decisions, which in other words means an increment in the program functionality [9]. There is a number of languages and tools [1, 2, 3] to serve as variability representation in SPL engineering. The variability representations usually implement either pure composition (*e.g.* AHEAD [3]) or annotations (*e.g.* `C` preprocessor-like). The novelty introduced by the pure composition approaches has imposed different adoption barriers, such as the novel concept of *class refinement* [3] and feature-based code organization. Such barriers may have produced the low adoption of feature-oriented approaches [10]. Thus, to accommodate the benefits of both compositional and annotative approaches to implement variability, the so-called hybrid approaches have emerged [1, 2]. Hybrid approaches, such as FeatureHouse [1] and FeatureC++ [2] allow to explore the benefits of both (composition and annotation). They usually avoid the introduction of new elements – usually unknown – in the existing programming languages, which may ease its adoption.

Such hybridism may also be seen as a transition path towards systems modularization using purely composition. This blending (composition with annotation) allows software engineers to explore benefits from both, as well as minimizing their existing drawbacks.

Even with the advances of FOSD research, at least two facets are still under development. First, given the big number of programming languages available, most of them are not supported by any FOSD approach. One of them is `JavaScript`, a widely used and the *de facto* programming language for the Web. Software engineers working with this language still need to resort of external constructs (apart from language-native ones) to achieve reasonable modularization, such as package managers (*e.g.*, `npm`, `jam`, `bower`) – that can be used to install software packages written by others and made available in the Web to be used locally in backend development – and file/module loaders (*e.g.*, `requireJS`) – that load single files or modules in the browser, asynchronously, on demand.

Second, after years of research in the the enhancement of FOSD approaches, little is known about the differences of the influence of variability representations on program comprehension. To the best of our knowledge, Siegmund [11] was the first one to address such an important topic and systematize empirical studies in FOSD context. Her framework helped the program comprehension experimentation by discussing inappropriate measures to program comprehension, confounding parameters, and a preliminary questionnaire to measure programming experience.

In this thesis, we investigate both facets and propose a contribution *(i)* to `JavaScript`-based product line engineering and *(ii)* to improve the understanding about the influence of FOSD variability representations on program comprehension. This Chapter contextualizes the focus of this work and starts by presenting its motivation and a clearer definition of the research problem, in Section 1.1. Next, Section 1.2 provides details of the thesis statement, highlighting the research goals. We present the steps taken to conduct this work in Section 1.3 and enumerate topics out of scope of this thesis in Section 1.4. The main contributions are listed in Section 1.5, and finally Section 1.6 outlines the thesis structure.

1.1 MOTIVATION

The ever increasing use of `JavaScript` for implementation of large software systems imply to deal with a higher complexity [12]. Such fact raises many kinds of challenges, *e.g.*, regarding modularization and the systematic reuse of artifacts. Such scenario fulfills the requirements for introducing SPL engineering, since it addresses these kind of issues. However, despite the researchers and practitioners effort spent towards establishing affordable and effective FOSD approaches [10], the `JavaScript`-based systems domain remains uncovered. Therefore, we proposed the RiSE Product Lines Engineering approach based on Hybrid Composition (RiPLE-HC) – to extend the FOSD umbrella and address such a problem. In general terms, RiPLE-HC is aimed to *(i)* allow the modeling of such systems in terms of features improving its modularity and *(ii)* fostering the improvement of the practices of artifacts reuse while also controlling the variability at low level with preprocessing annotations.

Additionally, although an international conference[1] discusses program comprehension for quite some years already, there is few evidence about how developers understand such relatively new programming paradigm, FOSD, in maintenance activities. In this scenario, Siegmund [11] paved the path to further investigations in this direction and encouraged the research community to pursue answers to different sets of unanswered questions (*e.g.*, what is the difference regarding comprehension effort while addressing maintenance tasks in feature-oriented software using either CONDITIONAL COMPILATION or FEATUREHOUSE?).

In this effect, considering existing support for SPL engineering, the `JavaScript` systems development demands, as exemplified above, and the importance of program comprehension to software maintenance, the central problems addressed in this thesis are the *lack of adequate support for the low-level variability in `JavaScript` product lines engineering* and the *lack of understanding of the influence of different variability representations on program comprehension.*

1.2 OBJECTIVES

The objective of this thesis is twofold: *(i)* to extend the umbrella of feature-oriented software development to cover the new demands of the `JavaScript`-based software systems development and *(ii)* to contribute to a better understanding of the influence of FOSD on program comprehension. In this sense, this thesis proposes and evaluates a hybrid composition approach for `JavaScript`-based systems and presents a family of experimental studies on the comprehension of annotation-based and composition-based systems. The research is guided by the following goals:

Research Goal 1: Propose a hybrid feature-oriented approach to support `JavaScript`-based product lines development, evaluate its feasibility – as the ability to handle variability in real world `JavaScript` systems – and its scalability – as the ability to scale to large `JavaScript` systems.

Research Goal 2: Contribute to the understanding of the influence of annotative-based and compositional-based approaches on program comprehension.

The first goal refers to a technical rather than a theoretical research contribution and we state no general research question, since the contribution only builds upon previous research to fill a specific gap. On the other hand, the second goal requires considerable amount of research, so we establish the following research question that drives the investigation:

What are the implications of relying on the feature-oriented software development paradigm on program comprehension during maintenance tasks?

[1] International Conference on Program Comprehension (<http://www.program-comprehension.org>)

We hypothesize that FOSD approaches hinder data-flow comprehension although it might improve on the system modularization. FOSD approaches that uses code hierarchies to hold source-code that belongs to each feature – *e.g.*, FeatureHouse [1] and FeatureC++ [2] – improve traceability feature-to-code and vice-versa. On the other hand, the *class refinements* [3] may hinder the understanding of the product line data-flow as a whole when in comparison to annotation-based code.

1.3 RESEARCH METHOD

This section describes the research design employed as the basis of this work. We split this investigation in three main parts: *Background*; *JavaScript FOSD*; and *Variability Implementation Comprehension*. Each of this parts used different research methods, which are described next.

Background. The initial part comprises an overview of the basic concepts on the foundations of this thesis, such as feature-oriented product lines and program comprehension, together with a literature review encompassing *state-of-the-art* approaches.

First, we manually reviewed the different ways of implement variability as feature-oriented product lines (*e.g.*, hybrid composition), which are in the core of the thesis. We are aware of systematic methods of literature reviews [13, 14], however, we found recent reviews [15, 16] addresing our topic of interest and decided to perform an *ad-hoc* complimentary review with exhaustive snowballing [17]. The evolution of the foundations on program comprehension research are also target of this review, including the different models of program comprehension proposed back in time. Such concepts provided the ground for us to devise our research questions and then narrow down the possibilities to be included in this investigation. Then, we performed a literature review to serve as an in-depth analysis of the current existing knowledge on the comprehension of feature-oriented programs.

JavaScript FOSD. The second part comprises the proposal of the feature-oriented approach to `JavaScript` systems together with preliminary empirical studies.

As a means of providing an overview of the novel approach (RiPLE-HC), we present benefits and drawbacks the hybrid composition can have according to the characteristics coming from annotative and pure compositional approaches. We also present the developed tool support available to software engineers. Furthermore, we performed proof-of-concept studies to assess the feasibility – in a single case study into the industry context – and scalability of the RiPLE-HC– by refactoring open-source projects into product lines.

Variability Implementation Comprehension. The third part comprises the definition of the family of experimental studies to gather evidence on the influence of the variability representations on program comprehension.

We believed this thesis would benefit from having both qualitative and quantitative methods in its studies. This mix of methods provides different sources of information, which are usually complementary. Thus, we decided to combine two types of experimental methods to carry our studies, namely the controlled *(quasi-)*experiments [18] and a focus group [19]. Each experiment had slightly different designs and focused on different aspects of the program comprehension. Furthermore, the focus group complemented the findings of the controlled experiments. It provides additional information impossible to capture with the experiments and can bring light to aspects uncovered in this thesis.

1.4 OUT OF SCOPE

It is out of scope of this thesis, the following topics:

- `JavaScript` **language grammar**: although we proposed a way to preprocessing `JavaScript` code to extract annotated blocks from the final product codebase, to extend the language grammar to cope with such annotation keywords or even to adapt the dynamic aspects of the language is out of scope;

- **Synergy between RiPLE-HC and** `JavaScript` **frameworks**: since there is several frameworks available to manage dependences as mentioned before, the evaluation of the synergy among RiPLE-HC and those pieces of `JavaScript` code is also out of scope.

- **Tool support for feature-oriented software comprehension**: although our studies raised evidence of the importance of the existence of them, the implementation or the improvement of those tools already available are tasks also out of scope.

- **Theory on feature-oriented software comprehension**: we believe this to be a long term goal. Several additional emprirical studies would be necessary to have enough evidence to build a grounded theory.

- **Comprehension Measuring**: we are aware of the difficulties to measure program comprehension. We used measures such as `response time` and `correctness` in our investigation, however the construction of a framework for comprehension measurement would require significant evidence, which is not available yet, even after our efforts in this thesis. Thus, this topic is also out of scope.

1.5 CONTRIBUTIONS

In accordance with our goals, the main expected contributions of this work are related to **feature-oriented software development** and they are listed in the following:

- `JavaScript` **development.** An approach and tool support for the development of `JavaScript`-based feature-oriented software [20, 21, 22].

- **Body of knowledge on program comprehension.** A literature review of feature-oriented program comprehension, a family of experimental studies (including extended empirical study designs) on the influence of different variability representations on program comprehension [21, 23, 24].

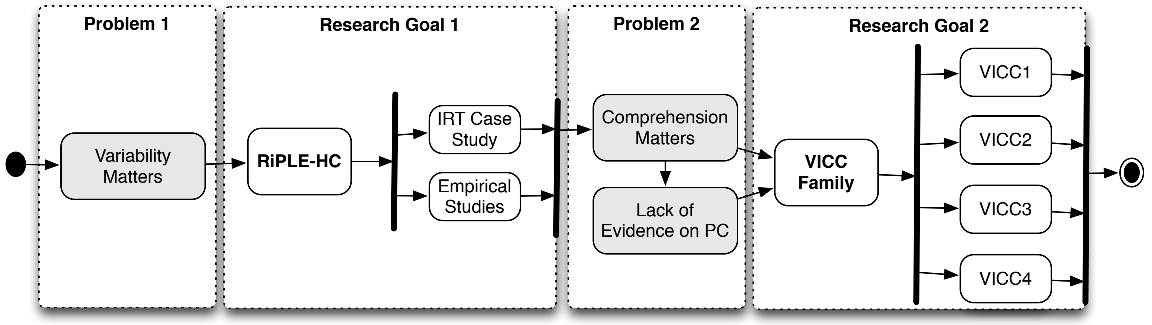


Figure 1.1 Schematic overview of the thesis development.

Figure 1.1 shows the history of this thesis development. From the literature and the background knowledge that our research group gathered over the years, we knew beforehand the importance of variability management (Problem 1). At some point, the opportunity of a collaboration with an industry partner brought us to the challenge of handling variability in `JavaScript`-based system (Research Goal 1). After the end of the industry partnership, we questioned ourserlves on the influence of such constructs of feature-oriented sofware development on the software maintenance activities. As program comprehension plays an important role in such activities and we found from a literature review the lack of evidence on the topic (Problem 2), we decided to plan and perform a family of experimental studies (Research Goal 2).

Table 1.1 shows a list of publications related to the thesis topic in order to get an overview of our contributions. The column "Participation" concerns the overall contribution of this author to the published work.

1.6 THESIS OUTLINE

The remaining parts of this thesis is structured in four parts and four appendices, as described next. Figure 1.2 shows a schematic overview of the thesis structure.

Part II - Background. This part provides background concepts on the topics involved in this investigation, namely *feature-oriented product line engineering* and *program comprehension*. In addition to the basic concepts, it also presents a literature review on approaches discussing program comprehension in the field of FOSD.

Chapter 2 (Concepts). Underlying concepts regarding the topic of this thesis.

Chapter 3 (Literature Review). We present a program comprehension literature review.

Part III - JavaScript FOSD. This part motivates and provides readers with detailed information about the novel approach to handle variability in `JavaScript` SPL engineering. We discuss how it was conceived and present the carried out evaluation on the viability and scalability of the approach. This part of the thesis covers the **Research Goal 1**.

Table 1.1 Publications during the Ph.D. research.

Paper Title	**Venue**	**Participation**
thesis related publications		
Low-level Variability Support for Web-based Software Product Lines [20]	**VaMoS'14**	*Significant*
RiPLE-HC: JavaScript Systems Meets SPL Composition [21]	**SPLC'16**	*Major*
RiPLE-HC: Views on the Features Scattering and Interactions [22]	**SPLC'16**	*Major*
Aspects Influencing Feature-Oriented Software Comprehension: Observations from a Focus Group [23]	**SBCARS'17**	*Major*
Exploring the Influence of Variability Representations on Program Comprehension *(*submitted)* [24]	**ESE**	*Major*
Related topics publications		
Strategies for Consistency Checking on Software Product Lines: A Mapping Study [25]	**EASE'15**	*Major*
Do #ifdef-based Variation Points Realize Feature Model Constraints? [7]	**SEN'15**	*Major*

Chapter 4 (RiPLE-HC). Definition and discussion of the approach.

Chapter 5 (RiPLE-HC Evaluations). Evaluation of the viability and scalability of the approach.

Part IV - Variability Implementation Comprehension. This parts presents our family of experimental studies planned and carried out to gather evidence regarding the influence of different variability representations on program comprehension. This part of the thesis covers the **Research Goal 2**.

Chapter 6 (Controlled Experiments Family). Overview of the carried out studies, as well as the variations in the experimental setup among them, including the description of the target systems, tasks, overall participants characterization, and topic addressed in the tasks.

Chapter 7 (Concept Location Experiment). Controlled experiment concept location with RiPLE-HC and *Standard* `JavaScript` development.

Chapter 8 (Bug-finding Experiments). Two replications of a bug-finding pilot experiment found in the literature with systems written using Conditional Compilation and FeatureHouse.

Chapter 9 (Focus Group). A qualitative study looking for new aspects easing, hindering, as well as the confirmation of the aspects already found in the previous studies.

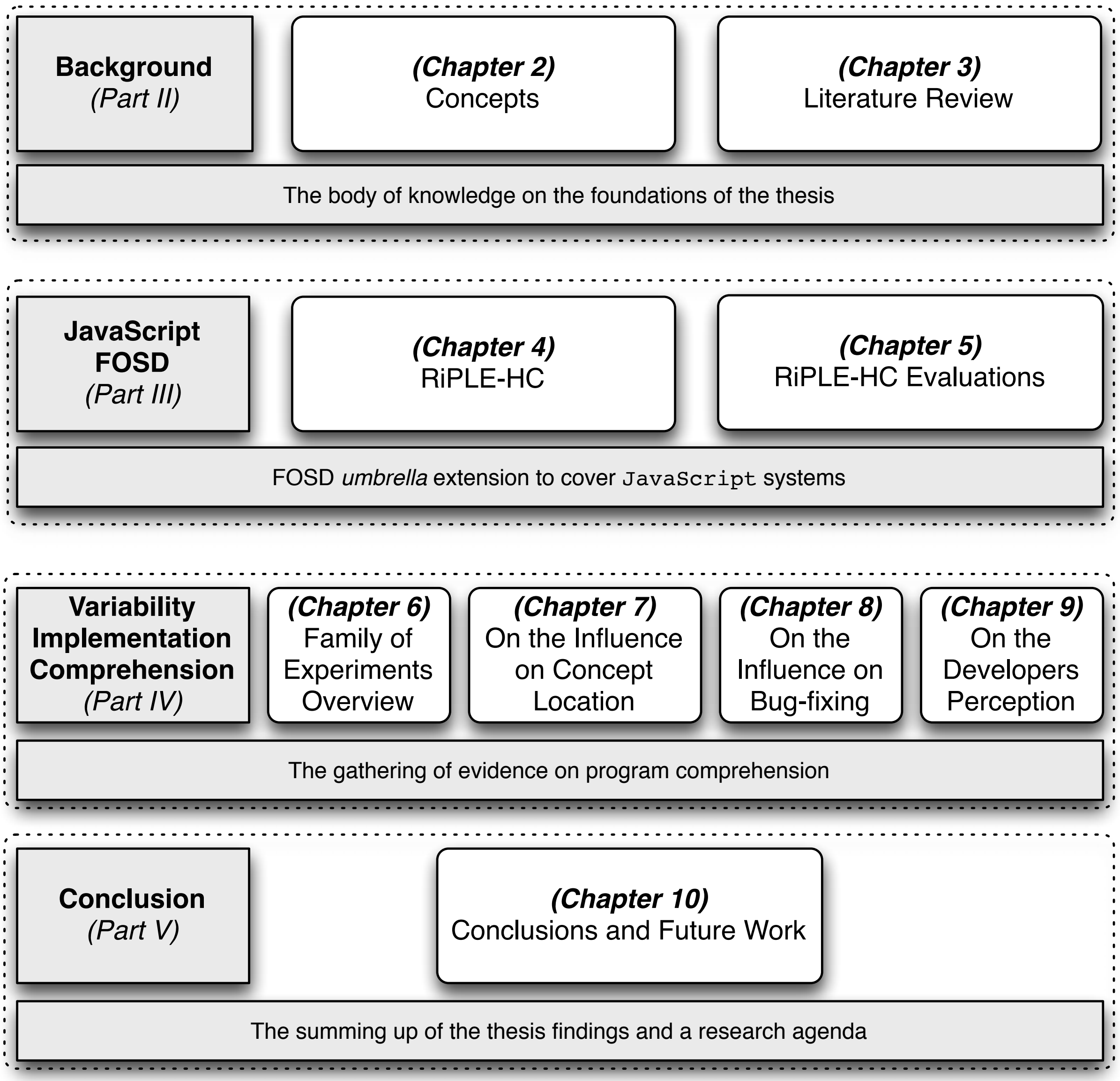

Background
(Part II)
(Chapter 2)
Concepts
(Chapter 3)
Literature Review
The body of knowledge on the foundations of the thesis
JavaScript FOSD
(Part III)
(Chapter 4)
RiPLE-HC
(Chapter 5)
RiPLE-HC Evaluations
FOSD umbrella extension to cover JavaScript systems
Variability Implementation Comprehension
(Part IV)
(Chapter 6)
Family of Experiments Overview
(Chapter 7)
On the Influence on Concept Location
(Chapter 8)
On the Influence on Bug-fixing
(Chapter 9)
On the Developers Perception
The gathering of evidence on program comprehension
Conclusion
(Part V)
(Chapter 10)
Conclusions and Future Work
The summing up of the thesis findings and a research agenda

Figure 1.2 Schematic overview of the thesis structure.

Part V - Conclusions. Finally, this part concludes the thesis, with a summary of findings of the studies and a research agenda.

Chapter 10 (Concluding Remarks and Future Work). The summing up of the thesis findings and directions for future work.

We suggest the following sequence of reading to better unders:

PART II

BACKGROUND

Chapter
2

FUNDAMENTAL CONCEPTS

The goal of this chapter is twofold: *(i)* to present the underlying concepts needed to understand this thesis; and *(ii)* to provide an overview of the software systems used in the empirical assessment. The chapter consists of two main sections. Section 2.1 presents the basics of *FOSD*, as a mainstream strategy to deliver SPL, deeply addressed in this thesis. Section 2.2 discusses program comprehension models and describes the *state-of-the-art* on *Program Comprehension* experimentation.

2.1 FEATURE-ORIENTED SOFTWARE DEVELOPMENT (FOSD)

FOSD is a paradigm used for the construction, customization, and synthesis of large-scale software systems relying on features [8]. A *feature* satisfies a requirement while performing design decisions, which in other words means an increment in the program functionality [9]. In fact, FOSD aims essentially at three properties: *structure*, *reuse*, and *variation* [8]. The *features* are the *reuse* unit in a software system *structure* designed to cope with *variation.*

FOSD emerged from ideas coming from different software engineering areas, such as programming languages, software architecture, and software modeling. Therefore, it shares several goals with other paradigms, such as *(i)* to encapsulate the individual development steps that implement distinct decisions – from the *stepwise and incremental software development* field –, *(ii)* to modularize crosscutting concerns – from the *aspect-oriented software development* field –, *(iii)* to construct software systems on demand using off-the-shelf components – from the *component-based software engineering* field [8].

The SPL engineering aims at constructing families of software systems as an alternative strategy to traditional Software Engineering, in which each system is developed individually from scratch. In SPL engineering, the emphasis is given to the similarities among the systems instead of their differences [8]. While software engineers can resort of all those paradigms to implement variability[1], they can favor the FOSD paradigm to

[1]This work consider Apel *et al.* [10] definition for "variability", which recalls to the ability to derive different products from a common set of artifacts.

organize and structure the whole SPL process, as well as all the artifacts involved in terms of features [10]. In such cases, we have Feature-Oriented Product Line Engineering (FOPLE), which in terms of variability implementation can be achieved through a multitude of techniques.

Apel *et al.* [10] defined three dimensions for classifying the techniques used to achieve variability in FOPLE: **Binding time**, **Technology**, and **Representation**. The first dimension concerns the time when the features are bound, also known as *product derivation* – one of the tasks of SPL engineering –, with regard to a given product configuration. Still, the binding can occur at *compile time* – before the product deployment – or *runtime* – while the software is executing. The second dimension concerns the mechanisms provided to operate the artifacts in order to derive products, which according to them can be at *language-based*, when a programming language is the provider, or *tool-based*, when a set of tools is the provider. Lastly, the third dimension concerns how to represent the variability in the software artifacts. Such representation can be performed through *annotations*, which are predefined keywords that allow the selection of a given feature code inside of an artifact, or *composition*, when the code belonging to each feature are placed in different artifacts.

In this work, we focus on the **representation** dimension of Apel's classification [10]. The **binding time** dimension requires an extensive and expensive context for experimentation in comparison to the representation dimension. In addition, the **technology** dimension concerns to aspects too specific of the software development, which in our point of view, the eventual contributions of would not last as much as those produced regarding the representation dimension, since technology can change fast. Moreover, we extend by including the so-called *hybrid* representation. The hybrid representation blends annotations and composition representations bringing to software engineers characteristics from both other representations together. Next, we present these three different representations. Moreover, as the three dimensions share common approaches, while introducing the representations, we may mention the technology dimension at some point for the sake of understanding.

2.1.1 Annotation-based Approaches

Definition 2.1. *Annotation-based approaches* annotate a common code base, such that code belonging to a certain feature is marked accordingly. During product derivation, all code that belongs to deselected features or invalid feature combinations is removed (at compile time) or ignored (at runtime) to compose the final product [10].

In this thesis, we call the annotation-based approaches *annotative approaches* for short. In these approaches, the code base is instrumented with annotation marks delimiting the code that belongs to the different features and removed on demand, which is called *negative variability*. CONDITIONAL COMPILATION is a preprocessing technique well known from the `C/C++` languages and later implemented in other languages either natively or as third-party tools. Such technique allows software developers to resort of annotation directives (*e.g.*, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`) to control the inclusion of the code that belongs to the selected features or the exclusion of the code from the deselected ones.

```java
//#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) {...}
    }
...
}
```

Figure 2.1 CONDITIONAL COMPILATION code example extracted from `MobileMedia` [26].

Figure 2.1 shows one code snippet of refinements of a given method by different features in the `MobileMedia` [26] with CONDITIONAL COMPILATION. All the code is in a single file, which is going to be preprocessed to remove the unwanted code before the actual compilation depending on the product configuration.

The drawbacks of the annotative approaches, such as the lack of modularity and poor readability have been roughly criticized in the research community. However, the ease of use and its flexibility made it the most used variability implementation mechanism [6]. In fact, developers seemed to be aware of the problems since they mostly use disciplined annotations[2] [27], although it may be necessary to rely on code replication in favor of better readability.

2.1.2 Composition-based Approaches

[2]In `C`, annotations on one or a sequence of entire functions and type definitions (*e.g.*, `struct`) are disciplined. Furthermore, annotations on one or a sequence of entire statements and annotations on elements inside type definitions are disciplined. All other annotations are undisciplined [27].

Definition 2.2. *Composition-based approaches* implement feature in the form of composable units, ideally one unit per feature. During product derivation, all units of all selected features and valid feature combinations are composed in a final product [10].

In this thesis, we call the composition-based approaches *compositional approaches* for short. Among these approaches, we can include the classic frameworks and components, as well as the advanced language abstractions and composition mechanisms to implement SPL, including feature-oriented and aspect-oriented programming. These approaches support the so called *positive variability* – when composition units are added on demand – and aim at keeping an *one-to-one* mapping between features and composition units.

Feature-Oriented Programming (FOP) [29] is a widely accepted language-based approach to achieve variability while resorting on composition. In other words, FOP is a compositional approach for building SPL that decompose a system's design and code into features it provides [29, 30]. In this effect, feature modules are commonly represented by file-system directories – called *containment hierarchies* – in several contemporary FOP languages and tools. For instance, in the `Jak` language together with the AHEAD tool suite [3, 30] and the FEATUREHOUSE experience [1]. In these cases, classes and their refinements are stored in files inside the corresponding containment hierarchies [10].

FEATUREHOUSE [1] is itself an (independent) language-based technique to implement variability, which provides mechanisms to operate artifacts to derive products in a composition-based approach. Figure 2.2 shows three code snippets showing the refinement of a given method by different features in the `MobileMedia` [26] with FEATUREHOUSE [28]. Each 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. These different snippets are supposed to be composed depending on the product configuration and the order in which they were listed for binding. The method call "`original()`" serves the purpose to insert the piece of code from the feature `Base`, which is mandatory, the the other refinements already binded to the final code in exact place of the call.

2.1.3 Hybrid Approaches

Figure 2.3 shows the big picture of the main differences between (a) annotative and (b) compositional approaches to product-line implementation. The latter emphasizes the modularization. A *hybrid approach* should explore the best characteristics from both worlds, such as the ability of feature interactions handling from annotative and the strengthened modularization of compositional approaches.

Kästner *et al.* [4] discussed the possibilities of the implementation of hybrid approaches. They showed an eventual path to combine advantages, increase flexibility for the developer, and ease its adoption. Later on, Kästner *et al.* [31] laid the foundation for such integration by providing a model that supports both physical and virtual Separation of Concerns (SoC) and by describing refactorings in both directions. In general words, virtual SoC differs from physical SoC by the use of colors to represent variability instead of the actual code annotations. Besides, Apel *et al.* [10] provided the following example:

(a) Music_OR_Video

```
public class MusicMediaUtil extends MediaUtil {
   private boolean isSupportedMediaType(MediaData ii){
        return false;
   }

   public byte[] getBytesFromMediaInfo(MediaData ii)
              throws InvalidImageDataException{
        try {
             byte[] mediadata = super.getBytesFromMediaInfo(ii);
             if (ii.getTypeMedia() != null) {
                if (isSupportedMediaType(ii)){
                  ...}
             }
             return mediadata;
        } catch (Exception e) {...}
   }
...
}
```

(b) Music

```
public class MusicMediaUtil extends MediaUtil {
     private boolean isSupportedMediaType(MediaData ii){
         return original(ii) ||
               ii.getTypeMedia().equals(MediaData.MUSIC);
    }
}
```

(c) Video

```
public class MusicMediaUtil extends MediaUtil {
    private boolean isSupportedMediaType(MediaData ii){
        return original(ii) ||
               ii.getTypeMedia().equals(MediaData.VIDEO);
    }
}
```

Figure 2.2 FEATUREHOUSE code example extracted from MobileMedia [28].

> “ We could decompose a system into composable units, where certain components are themselves variable in the sense that their implementations are annotated. During product derivation, a generator would select a subset of composition units and remove annotated code from them that belongs to deselected features.
>
> *Apel et al.[10]* ”

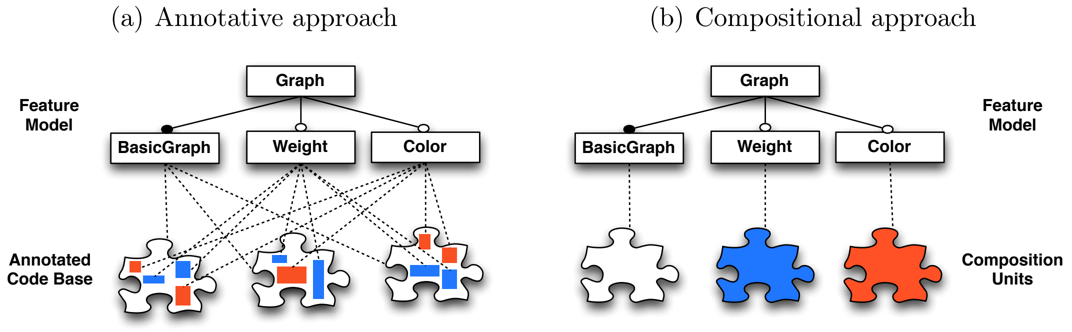


Figure 2.3 Annotative and compositional approaches to product-line implementation. Adapted from Apel *et al.* [10].

Recently, Apel *et al.* [1] introduced a language-independent approach based on superimposition, called FEATUREHOUSE, which unintentionally allowed hybrid composition of `C/C++` systems, while implementing software composition for such languages without cut the preprocessor annotations out. In fact, their work inspired us while to conceive our hybrid approach for `JavaScript`-based systems, the RIPLE-HC (which we present in detail in Chapter 4). The RIPLE-HC was developed after discarding such initiative as a viable solution for an industrial partner. The last quote describes the behaviour of our proposed approach. Similarly with FEATUREHOUSE, RIPLE-HC compose units (without the `original()` method feature) and remove annotated code blocks belonging to deselected features.

2.2 PROGRAM COMPREHENSION

Definition 2.3. *Program comprehension* is an internal complex cognitive problem solving process of understanding unfamiliar program code [32].

The program comprehension process takes place in both, development and maintenance phases of the Software Development Life Cycle (SDLC). Besides, it can happen through different ways depending on the background knowledge – which consists of the (in) formal education, as well as techniques, languages, and paradigms known by the developer – and the developer understanding of the program domain – the context in which the program was built for. Moreover, the amount of knowledge a developer in charge of a maintenance task holds may determine how he/she addresses the comprehension tasks. There are three kinds of models which describe the comprehension tasks, as follows: top-down models [33, 34, 35], bottom-up models [36, 37], and integrated models [38].

In a nutshell, while developers understand a program in a top-down fashion if they are familiar with the program's domain, they use a bottom-up approach otherwise. In other words, in the former situation, they state hypotheses on the program purpose without take a closer look in the source code by comparing the current program with familiar ones. In an integrated model, they cannot rely on any comparison, thus they are supposed to take a closer look into the program statements. The integrated models consider developers hold an adequate knowledge about the program domain, but it is usually not enough to clearly understand the code, in the sense they need to rely on a different number of strategies, such as interacting with User Interface (UI) to test expected program behavior or debug application to elicit runtime information [39].

Assessing program comprehension is not a trivial task due to the cognitive nature of the process, especially when it involves measuring the comprehension of different individuals, with different backgrounds. Siegmund and Schumann [15] presented a literature survey on the confounding parameters of the program comprehension tasks used in a wide range of experiments. They also presented an overview of the measurements and the control techniques for these parameters. In fact, Siegmund proposed a framework for measuring program comprehension in her Ph.D. thesis [11]. She relied on those different comprehension models to build a guideline framework to aid the design of program comprehension experiments. More specifically, Siegmund's framework [11] presented four main contributions:

1. It is not recommended to use feature-oriented software measures as indicators of program comprehension, since the results of their experiment did not indicate a relationship between such measures and program comprehension. [40];
2. An extensive list of confounding parameters (Table 2.1) for program comprehension presented together with their controlling techniques [15];
3. An evidence-based questionnaire to reliably and conveniently measure program experience, which included questions regarding knowledge of programming languages, programming paradigms, number of courses taken, and experience with large software projects [41];
4. A tool to fulfill common requirements of program-comprehension experiments [42, 43].

Table 2.1 Confounding parameters on program comprehension experimentations [15].

Type	**Confounding parameters**
Individual knowledge	Ability, Domain knowledge, Education, Familiarity with study object, Familiarity with tools, Programming experience, and Reading time.
Individual circumstances	Fatigue, Motivation, and Treatment preference.
Individual background	Color blindness, gender, culture, and intelligence.

Siegmund's framework is robust, in the sense that it covers the different facets of program comprehension experimentation, *i.e.*, planning, controlling, measuring, and support. Therefore, we are going to use it as the proper guideline to plan and execute our experimentation studies.

2.3 CHAPTER SUMMARY

In this chapter, we presented an overview of the topic approached in this thesis. We started by introducing the big picture of the FOPLE while discussing the main goal of each variability representation techniques. Regarding the variability representations, we detailed annotative approaches, compositional approaches, as well as the hybrid approaches consisting of a blending of composition and annotations. Next, we briefly introduced the existing program comprehension models and described the foundation concepts regarding experimentation in such a context. We also discussed Siegmund's framework for program comprehension experimentation, including the use of software measures, confounding parameters, the measuring of programming experience, and the available tool support.

Next chapter presents a literature review on the program comprehension experimentation.

Chapter 3

LITERATURE REVIEW

The goal of this chapter is to present a literature review of the experimentation on influence of the differences among variability representations on the program comprehension, the main topic addressed in this thesis. We performed the literature review in two steps: *(i)* conducted a manual search using both, the main venues related to program comprehension in feature-oriented software and the Siegmund and Schumann survey confounding parameters on software engineering experimentation [15]; and *(ii)* conducted an *ad-hoc* search using Google Scholar for both aforementioned topics.

The chapter encompasses four main sections. Section 3.1 presents the method used to carry out the literature review. Section 3.2 presents an overview of the quantitative data of the selected papers. Section 3.3 discusses the selected papers in the literature review process. Finally, Section 3.4 presents related work that may contribute to the kind of investigation carried in this thesis.

3.1 METHOD

In this section, we present the method used to review the existing literature regarding to empirical assessment of the impact of FOSD on program comprehension evidence corpus. As earlier introduced, we performed the review in two phases, namely manual and automated *ad-hoc* searches instead of use a systematic methods of literature review [13, 14]. However, we we did follow a method using previous literature reviews we found [15, 16] as the basis of ours. We describe the process next.

3.1.1 Selection Process

In the manual phase, we used the selected papers set (842 in total) from the catalog of confounding parameters found in controlled experiments on program comprehension [15], which covered from 2001 until 2010. Additionally, we selected 2458 papers from a broad and representative set of journals (3), conferences (5), symposia (2), and workshops (2) published since 2011, except from GPCE, which was not covered in the Siegmund and

Schumann survey [15] found at the Computer Science Bibliography (DBLP) for the matter of completeness. Both venues' list can be found in Appendix A. In the *ad-hoc* phase, we did not took into account the amount of the papers reviewed since we adapted the search string recurrently until no new relevant paper could be retrieved from the first five result pages of Google Scholar. The author of this thesis performed the review himself.

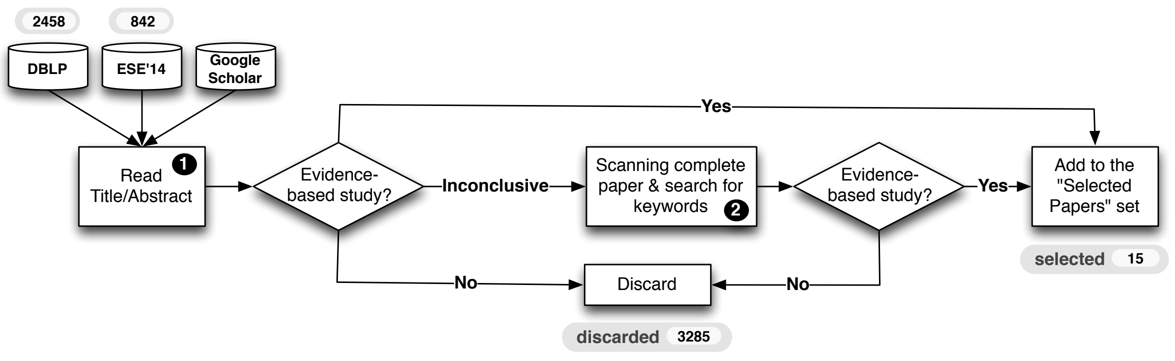


Figure 3.1 Papers selection process.

Figure 3.1 shows the paper selection process used in both, manual and *ad-hoc* reviews. We started the selection of the papers in the *Step* ❶ with the reading of the papers' titles – and in inconclusive cases also the abstract. Whenever the reading of abstracts could not clearly indicate either the presence or absence of evidence-based research, we proceeded with the *Step* ❷ in which the main task was to scan the whole paper to clarify the nature of the work. All the paper selection was made only by the author of this thesis without any review from other researchers.

3.1.2 Inclusion/Exclusion Criteria

In order to select the papers as relevant in our literature review, we searched for a set of keywords in the title/abstract/body as described before. We used these keywords to characterize the publications as research based on evidence concerning the program comprehension of feature-oriented software. The keywords used in the searches of the literature review were the following: "program comprehension", "software understanding", "variability implementation", "virtual separation of concerns", "physical separation of concerns", "controlled experiments", "empirical studies". We also considered possible synonyms of them.

3.2 DATA COLLECTION

In this section, we present the quantitative data about the selection of papers according to the described selection process. Tables 3.1 and 3.2 show the number of papers selected per year from each venue in the manual and *ad-hoc* searches, respectively. For each venue, the tables show the amount of papers target of review ("Available" rows) and how many out of them were selected ("Selected" rows). The last two rows of the tables subsume such values from all the venues.

We knew beforehand few papers addressing the influence of variability representations on program comprehension. So far, the main work addressing such identified problem has been published by Siegmund *et al.* [28]. In the paper, the authors discussed the importance of such studies and argued in favor of further experimental studies in such context for a better understanding of the differences in the point of view of the (potential) developers.

Our paper selection process confirmed such assumption of lack of studies in the field. For the manual search, Tables 3.1 and 3.2 show only 11 papers – all selected from the DBLP set of papers and 4 out of those selected in the Siegmund and Schumann survey [15] – contributed, at least to a certain extent on this matter. For the *ad-doc* search, we could find any additional paper concerning the comprehension of different paradigms in the variability implementation. Therefore, we only selected papers concerning to literature reviews, opinion/discussion papers on the role of program comprehension in software engineering tasks.

3.3 RESULTS

In this section, we discuss the selected studies in the manual and the *ad-hoc* searches of the review process. The final set contains contributions of different types, therefore, we present them grouped by their macro topics in the following subsections. Namely, we discuss *(i)* literature reviews, *(ii)* understanding of program comprehension, and *(iii)* experiment reports. These are discussed next.

3.3.1 Literature Reviews

An important subset of the selected papers are literature reviews. We found three of them [44, 45, 16]. They provide a comprehensible landscape of the current *state-of-the-art* of program comprehension research. Storey [45] presented a review of some of the existing key cognitive theories that emerged prior to 2006, whereas Siegmund [16] recently published her thoughts exploring the research field discussing the past, the current state, and outlining what might be the next steps in the field.

Regarding their research questions: Mayhauser and Vans [44] addressed common elements of six cognition models and compare them based on their scope and experimental support available prior to 1995; Storey [45] addressed the diverse theories, research methods and tools published prior to 2006 that were raised and produced from the multitude of differences in program characteristics, programmer ability, and software tasks; Finally, Siegmund [16] highlight past successful theory-driven research on program comprehension and efforts to support the programmer, discuss the current state and its problems, and outline where future research might be directed to. Siegmund looked back from the year 2016. All these litereature reviews are complimentary. Next, we detail each of them.

Mayhauser and Vans [44] reported on program comprehension during software maintenance and evolution. They related the former existing models of program comprehension with the different tasks of software maintenance. They surveyed the evaluation studies of those models, which they classified on observational, correlational, and hypotheses-testing experiments. They observed a lack of approaches providing the research community with

Table 3.1 Number of selected paper in each forum per year.

Venue		**2011**	**2012**	**2013**	**2014**	**2015**	**2016***	`Total`
ESE	Available	25	23	32	55	50	48	233
	Selected	-	-	1	-	-	-	1
TSE	Available	48	82	95	63	61	31	380
	Selected	-	-	-	-	-	-	0
TOSEM	Available	17	18	35	43	22	8	143
	Selected	-	-	-	-	-	-	0
SPLC	Available	31	34	28	36	37	-	166
	Selected	1	-	-	1	-	-	2
ICSR	Available	16	-	23	-	24	24	87
	Selected	-	-	-	-	-	-	0
GPCE	Available	18	15	20	13	20	-	86 + 128**
	Selected	-	-	-	-	-	-	0
ICPC	Available	18	23	19	20	23	-	103
	Selected	1	-	-	-	1	-	2
ICSE	Available	62	105	100	99	83	101	550
	Selected	-	-	-	1	-	-	1
FSE	Available	34	34	49	61	88	-	266
	Selected	-	-	-	-	1	-	1
ESEM	Available	40	21	34	38	21	-	154
	Selected	1	-	-	-	-	-	1
VAMOS	Available	21	22	19	21	16	14	113
	Selected	-	-	-	-	1	-	1
PLEASE	Available	12	16	14	-	7	-	49
	Selected	-	-	-	-	-	-	0
FOSD	Available	8	10	6	6	-	-	30 + 28**
	Selected	-	1	-	-	-	-	2
	`Total Available`	353	400	474	455	452	226	**2516**
	`Total Selected`	2	1	1	1	2	-	**10+1****

* until July of 2016; ** number of papers analyzed from the editions before 2011.

a clear picture of the program comprehension processes. Fortunately, we might observe an increased interest by the research community at investigating the program comprehension field, as pinpointed by Storey [45] in the second review.

Storey [45] discussed different facets of the program comprehension research, covering concepts and terminology, comprehension models (top-down, bottom-up, knowledge-based, which mix the top-down and bottom-up models), opportunistic and systematic strategies, and finally the integrated metamodel built upon influences from the previous models. The author then discussed the impact of program characteristics and the influence of individuality of developers in the program comprehension tasks. Besides the

Table 3.2 Number of selected paper in each forum per year on the paper selected from the Siegmund and Schumann survey [15].

Venue		**2001**	**2002**	**2003**	**2004**	**2005**	**2006**	**2007**	**2008**	**2009**	**2010**	`Total`
TOSEM	Available [15]	-	-	-	-	1	-	3	-	-	3	7
	Selected	-	-	-	-	-	-	-	-	-	-	-
TSE	Available [15]	5	6	6	5	5	3	2	5	3	5	45
	Selected	-	-	1	1	-	-	-	-	-	-	2
ESE	Available [15]	2	9	4	9	10	7	7	4	7	3	62
	Selected	1	-	-	-	-	-	-	-	-	-	1
JSEP	Available [15]	4	4	3	5	1	2	2	3	1	5	30
	Selected	-	-	-	-	-	-	-	-	-	-	-
CHI	Available [15]	16	16	16	19	22	24	22	35	52	47	269
	Selected	-	-	-	-	-	-	-	-	-	-	-
ICPC	Available [15]	-	-	-	-	-	2	4	6	4	3	19
	Selected	-	-	-	-	-	-	-	-	-	-	-
ICSE	Available [15]	8	10	8	1	11	11	10	8	4	12	83
	Selected	-	-	-	-	-	-	-	-	-	-	-
ICSM	Available [15]	7	10	4	5	8	11	13	7	5	5	75
	Selected	-	-	-	-	-	-	-	-	-	-	-
VLHCC	Available [15]	-	-	-	11	9	10	8	7	9	11	65
	Selected	-	-	-	-	-	-	-	-	-	-	-
WCRE	Available [15]	4	6	6	7	3	6	9	4	5	11	61
	Selected	-	-	-	-	-	-	-	-	-	-	-
FSE	Available [15]	-	1	-	-	1	2	2	3	1	1	11
	Selected	-	-	-	-	-	-	-	-	-	-	-
ESEM	Available [15]	-	-	-	-	-	-	12	3	11	8	34
	Selected	-	-	-	-	-	-	-	-	-	-	-
CHASE	Available [15]	-	-	-	-	-	-	-	11	9	8	28
	Selected	-	-	-	-	-	-	-	-	-	-	-
HCC*	Available [15]	11	11	6	-	-	-	-	-	-	-	28
	Selected	-	-	-	-	-	-	-	-	-	-	-
IWPC**	Available [15]	2	3	4	8	8	-	-	-	-	-	25
	Selected	-	-	-	-	1	-	-	-	-	-	1
	`Total Selected`	1	-	1	1	1	-	-	-	-	-	**4**

***** HCC became VLHCC in 2004. ****** IWPC became ICPC in 2006.

theories, the author also discussed the tools supporting them and enumerated a number of requirements identified to provide such support, such as "providing of data and control flow information to maintainers", "software visualization", as well as "search and history" features.

Siegmund [16] argues that research on the understanding of how and why the advances on programming environments, software visualizations, programming languages help developers are rather limited. The author points out the mid-90's ceasing of the progress in the related research for over a decade as one of the main reasons. Siegmund work [16] discussed different facets of the research field, such as the *(i)* measuring of program comprehension, *(ii)* modeling program comprehension, *(iii)* programming languages, and *(iv)* programming tools. The author highlighted the following points from the developers's life that should be considered in further empirical studies:

- Getting an overview of a large program or software architecture;
- Understanding type structures and call hierarchies;
- Understanding the relationship between components; and
- Identifying the developers who are responsible for a component.

Indeed, these are broad topics and will take several research effort to cover all of them. However, in this thesis, we intend to go a step further by contributing with the understanding of the influence factors on developers comprehension of the variability implemented in both, annotative and compositional approaches.

3.3.2 Understanding Program Comprehension

While deciding to conduct research on program comprehension, it is essential to understand its role in Software Engineering as a whole. Rajlich and Wild [46] presented a preliminary discussion on the role of concepts[1] in program comprehension. Their work mainly described concept location scenarios and case studies pinpointing their importance in the construction of domain knowledge. They highlighted the role of search techniques in the process of understanding unfamiliar code.

Burkhardt *et al.* [47] investigated the effect of three different factors on program comprehension of the object-oriented (OO) paradigm. Namely, they addressed programmer expertise, programming task and the development phase. Although it only addresses OO systems, their work raises interesting aspects that can eventually be evaluated in the context of other paradigms. Moreover, they showed that the models used to measure comprehension were too simplistic and/or language-dependent. Lately, in the International Conference on Program Comprehension (2007), Penta *et al.* [48] carried out a working session with the conference's attendees. The main topic was the designing of program comprehension empirical studies, which reinforces the importance of such type

[1]Such concepts were lately described as concerns in aspect oriented research [26] – which are associated to the feature concept discussed in the Chapter 2.

of research. In fact, Siegmund [11] also contributed to measure of different aspects involved in the comprehension process, such as programming experience of the developers and the confounding parameters observed during experimentation.

Later, Maalej *et al.* [39] conducted an exploratory study on the identification of strategies which developers use to comprehend, tools supporting their work, important knowledge during comprehension tasks, channels they use to share knowledge, and the problems faced in real experiences. They found a gap between the *state-of-the -art* and *-practice* that questions the usefulness of comprehension tools suggested by research (*e.g.*, 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 comprehension in the core of maintenance activity and try to systematize the whole process, developers avoid it whenever possible focusing on the expected output.

Recently, Siegmund *et al.* [49] started a sophisticated series of studies on program comprehension. They analyzed which brain areas were activated while developers performed such cognitive process by using functional magnetic resonance imaging. They found clear activation of specific brain regions associated with working memory, attention, and language processing. Moreover, they claim the programming education might be more efficient when focused in the language skills instead of working memory or problem solving tasks.

3.3.3 Reports on Experimental Studies

We found few reports involving experimental studies on program comprehension among different paradigms [50, 28, 51]. In fact, only Siegmund *et al.* [28, 51] addressed feature-oriented software, while Kosar *et al.* [50] assessed different languages using feature diagrams as one of the evaluation scenarios.

Kosar *et al.* [50] presented a family of experiments on the program comprehension of domain-specific and general-purpose languages. The experiments covered three different domains and different applications for each task analyzed. The results showed developers are more accurate and more efficient in program comprehension when using a domain-specific language than when using a general-purpose language. Such kind of evidence is unavailable regarding #`ifdef`-based and feature-orientated software, which we plan to contribute by identifying influence factors on program comprehension while developers address code using different variability representations.

Siegmund *et al.* [28] conducted a pilot study on the comparison of program comprehension while addressing annotative and compositional source code, which is one of the main topic of this thesis. To the best of our knowledge, this is the only work addressing program comprehension on such different paradigms for feature-oriented software. Yet, this study brought only preliminary evidence to the field and encouraged researchers to replicate the pilot, as well as to extend their findings. We replicated their study with students enrolled in a Software Engineering graduate program in order to strengthen or contrast their findings [52]. Such a replicated study is further addressed in Chapter 8.

Siegmund *et al.* [51] presented a family of controlled experiments regarding the physical and virtual SoC effectiveness of the use of background colors to help on the maintenance of software systems instead of an annotative approach. The main question addressed was whether the use of colors instead of `#ifdef` directives reduces the code obfuscation introduced by the latter while implementing variability in feature-oriented software. Background colors showed potential to improve program comprehension independent of size and programming language of the project.

3.4 RELATED WORK

In this section, we present some related work that did not addressed the influence of variability representations on program comprehension directly, but might have contributed somehow to our research.

Robillard *et al.* [53] investigated the behavior of developers while inspecting code in order to implement a change request. They concluded that "in the context of a program investigation task, a methodical investigation of the code of a system is more effective than an opportunistic approach." Although they addressed only OO code, they contributed with a set of detailed observations about the characteristics of effective program investigation behavior and a detailed methodology for performing empirical studies of programmers where it is important that the programmer behavior be studied in detail.

Fritz *et al.* [54] investigated a novel approach to classify the difficulty of code comprehension tasks while developers are programming. They recorded the study using psycho-physiological sensors, *e.g.*, eye-tracking. They also relied on the record of think-aloud narrative and screen capture. Finally, they used machine learning to define classifiers to predict the task difficulty. They achieved the best overall performance outcomes when predicting a new task with 84.38% precision and 69.79% recall.

Melo *et al.* [55] carried out a controlled experiment to quantify the impact of the degree of variability in the bug finding tasks. Their results showed 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 for the subjects identify the exact set of affected configurations appears to be harder than finding the bug in the first place. In our investigation, instead of the degree of variability, we resort on tasks using two different variability representations for participants to identify feature precedence in warming up tasks for a focus group (details in Chapter 9).

3.5 CHAPTER SUMMARY

In this chapter, we presented a literature review carried out to provide an overview of the existing research related to the topic of this thesis. We first detailed the used method to conduct the review, including the sources of literature, the revisited previous literature review, the considered venues, and the include/exclude criteria adopted in the process. Next, we enumerated the quantitative data regarding the selected papers. Finally, we concluded the chapter discussing the selected papers and showing how they relate to this work.

Next part of this thesis consists of two chapters. Chapter 4 presents our feature-oriented approach to handle variability in `JavaScript`-based systems, whereas Chapter 5 brings the preliminary evaluation of our approach.

PART III

JAVASCRIPT FEATURE-ORIENTED SOFTWARE DEVELOPMENT

Chapter 4

JAVASCRIPT HYBRID COMPOSITION (RIPLE-HC)

The RiSE Product Lines Engineering approach based on Hybrid Composition (RiPLE-HC) implements a strategy to handle variability at both feature modeling and code level for `JavaScript` -based systems. It encourages the use of the feature-based code organization and allows the use of preprocessing annotations to handle fine-grained variability. This chapter is dedicated to introduce the RiPLE-HC concept and the methods underlying the strategy.

The chapter consists of five main sections. Section 4.1 presents the motivation for such a proposal. Section 4.2 discusses the lack of work addressing the systematic reuse in `JavaScript`-based systems. Section 4.3 describes its inception and its overall characteristics. Section 4.4 details the implementation of the approach and the available tool support. Section 4.5 presents a rationale on the inherited characteristics by RiPLE-HC from pure composition and annotative approaches.

4.1 MOTIVATION

`JavaScript`-based systems can be found in different platforms and such programming language is not only used to implement Web-based systems [56], *e.g.*, Brackets is a powerful general purpose text editor implemented in `JavaScript`.[1] Moreover, developers are turning their efforts to build modular code, so as to foster software reuse. For instance, Silva *et al.* [12] investigated the use of classes-like and inheritance constructs in `JavaScript` development. They concluded that, although the language is class-free, prototype-based, and it will probably always keep such status, 74% of the analyzed systems (out of 50 systems) make use of "classes" – from which 8% have the vast majority of their data structures implemented with such construct. Only 26% of them do not rely on such construct in order to organize the code. They suggested to consider the adaptation of the `JavaScript` ecosystem to provide tools, concepts, and techniques that cope with characteristics of systems that use class-based languages.

[1]Available at: <http://brackets.io/>

At the same time, the complexity of `JavaScript`-based software systems is increasing and a significant amount of complexity comes from handling the dynamic behavior of their features, which sometimes depend on either the presence or absence of another feature. This ever increasing complex scenario satisfies SPL engineering key characteristics, as it may provide `JavaScript`-based systems with the opportunity to move from a custom software development approach to build a set of products and assembling reusable modules, in a systematic and coordinated fashion. Unless the business goals establish a limited audience for the developed systems, SPL engineering can be considered as a suitable strategy to cope with the large amount of system variations and complexity [57].

Research effort concerning the introduction of SPL engineering in the Web systems domain can be found elsewhere [58, 59, 60]. However, they are mostly concerned with modeling domain variability in a high-level abstraction, as a means to represent the common and variable features. For instance, in the feature-oriented software development paradigm, the use of composition as the only mechanism to support variability may limit the handling of feature interactions, as well as the support of fine-grained variability management [61]. While it can facilitate the understanding of how products can be composed in terms of features, it is rather important to manage variability in both, coarse and fine-grained implementation levels, given that source code holds important role in establishing variable behavior.

In fact, Medeiros *et al.* [6] showed that developers keep using conditional compilation in such scenarios in spite of the recommendations against its use due to its likely harmfulness. In fact, the *easy-to-use* approach and the flexibility `#ifdef` annotations provide, may allow the development of highly configurable systems sheltered from inconsistencies, even in large systems, such as the Linux kernel [7]. In this scenario, some initiatives – such as FEATUREC++ [2] and FEATUREHOUSE [1] – made viable to explore the benefits of both compositional and annotative approaches to provide a middle-term solution for the problem, hybrid approaches.

To the best of our knowledge, despite of the recommendations on the modularization of large `JavaScript` applications[2] of the `JavaScript` developers community, there is no proper tool support for a systematic reuse and organization of the code, apart from the external constructs, such as package managers (*e.g.*, `npm`, `jam`, `bower`) and dependencies managers (*e.g.*, `requireJS`). We discuss this lack of approaches in details in the next section.

4.2 REUSE IN JAVASCRIPT-BASED SYSTEMS

Apel *et al.* published a book on feature-oriented product lines in 2013 [10], which we assume as a broad and representative list of relevant related approaches. Therefore, the following studies were selected from an *ad-doc* search regarding the existing approaches addressing the systematic reuse in `JavaScript`-based systems and Web-based product lines.

There is a number of tools available to foster modularity in `JavaScript`-based sys-

[2]Recommendations on the code organization of large `JavaScript` applications (<http://goo.gl/iKcJPD>). Accessed in July 12th, 2017.

tems, namely, package managers (*e.g.*, `npm`, `jam`, `bower`, etc.), dependencies managers (*e.g.*, `requireJS`), among others. In addition, the `JavaScript` developers community package their scripts usually compressed by removing unecessary blank spaces to reduce the size of the documents users would have to download. This fact by itself already illustrates the need of removing also the unecessary code, which the use of "classes" or the modularization recomendations mentioned befored do not solve. However, these approaches do not allow the project features management based on a feature model or product composition. In fact, our approach does not exclude or intend to substitute such tools, but to improve the reuse in such systems instead. We could not found any literature addressing such an issue. Next, we discuss some investigations, which do not deal with such problem but we considered as related to ours.

Clone-and-own requires no major upfront investments and it is intuitive. Fischer *et al.* [62] proposed a new approach to enhance its use and address the lack of systematic reuse methodologies in industry scenario. They support the development and maintenance of software product variants by providing proper guidance during the artifacts adaptation tasks.

Apel and Kästner [8] pointed out as a key challenge for feature-oriented programming approaches the handling of feature interactions. Recently, empirical studies concerning to feature-oriented programming provided evidence that reinforced their claim [61, 63]. As far as we know, Prehofer [29] is one of the first researchers to highlight the relevance of such a problem, while proposing dedicated modules to implement feature interactions, called *lifters*. Liu *et al.* [64] extended the *lifters* notation to *derivatives* and presented a theory for feature interaction.

We also found some studies dealing with the composition of Web systems to a certain extent, *e.g.*, by using strategies such as XML-based [58] or feature-oriented programming [59]. Nevertheless, they also focus rather on modeling aspects. For instance, Trujillo *et al.*[60] presented a mix of FOP and Model-Driven Development (MDD), the Feature-Oriented Model Driven Development (FOMDD), which shows how products in an SPL can be synthesized in an MDD way by composing features to create models, and then transforming these models into executables. By contrast, in this present investigation, we considered a lower level of abstraction, while proposing a strategy to cope with variability at the implementation level. None of these studies deal with feature-based composition nor present any empirical evidence of such. Therefore, we might observe a lack of empirical evidence on the impact of hybrid composition software development and on the maintenance tasks in `JavaScript` -based systems.

Capilla and Dueñas [59] presented a light-weight SPL architecture to control the evolution and maintenance of new Web products and facilitate the maintenance operations on Web sites. The authors claim their approach reduces the development costs, and the benefits of the SPL engineering can be noticed earlier. The research was based on data from an initial analysis of two Web sites.

Pettersson and Jarzabek [58] used an XML-based Variant Configuration Language to turn a Web portal into a more flexible architecture to reap the benefits of new business opportunities that required rapid development and further maintenance. However, developers had to manage multiple languages (XVCL, ASP, HTML) without specific

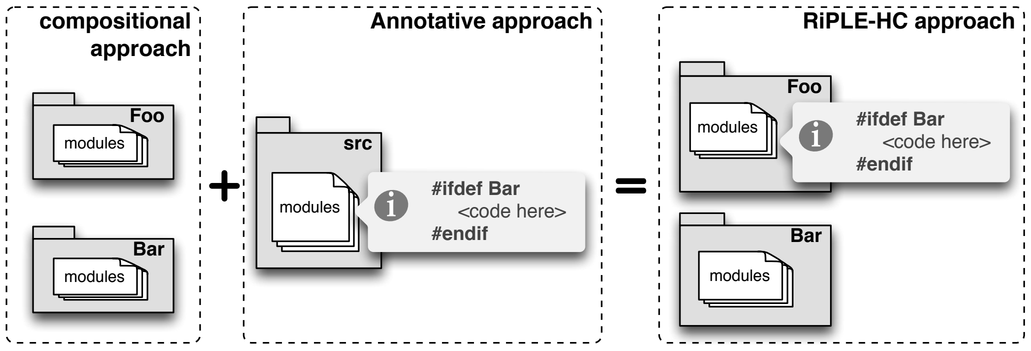


Figure 4.1 RiPLE-HC code organization: blending feature-based code organization and preprocessing annotations.

tool support, which may impact on both productivity and maintainability. Additionally, while runtime debugging the webportals this approach forces developers to remember the several mappings used, increasing complexity.

Trujillo *et.al.* [60] blended FOSD and model-driven development (FOMDD) to optimize the synthesis of portlets in Web portals. A benefit of FOMDD is that it is mathematically based, and this makes connections with category theory [65] easier to recognize. Conversely, FOMDD requires additional effort to manage model and perform all transformations needed, which without suitable tools it is likely that it might lead to higher development costs and reduced quality of the models.

All of these studies propose strategies to handle Web-based SPLs to a certain extent. Nevertheless, they also focus rather on modeling aspects. Besides, the FeatureHouse experience [1] was discarded while trying to extend the approach to cope with the `JavaScript` language structure. Therefore, we considered a lower level of abstraction and propose a strategy to handle `JavaScript` -based systems variability at implementation level. Our approach aims to promote the modular and systematic reuse of artifacts in a feature-oriented way. The approach is presented next.

4.3 CONCEPT

As the name suggests, RiPLE-HC is a hybrid approach that blends *compositional* and *annotative* approaches of SPL engineering [10]. RiPLE-HC explores the modularization of the compositional approaches and the flexibility that annotative approaches enable to handle feature interactions. Such a blending allows to manage variability at different levels of the development phase. Besides, while the composition handles the inclusion or exclusion of an entire functionality in a product variant (coarse-grained variability), the annotations enable inner-function statements to behave differently (fine-grained variability), depending on the selection of a given feature.

Figure 4.1 shows how the RiPLE-HC employs the concept of feature-oriented software product lines [10] to organize the source code. Containment hierarchies organize the features [30], in which each directory holds elements of a given feature, including the source code. The containment hierarchy is a way to modularize the code and ease the composition implementation. However, in practice, feature interaction problems – the behavior of a given feature `Foo` being changed due the presence or absence of feature `Bar` (as Fig. 4.1 illustrates) – make it too hard to have no code scattering, which directly impacts the code organization. The hybrid composition of RiPLE-HC makes it possible to handle feature interaction limitations of pure composition [61] by allowing the use of preprocessing annotations. Thus, there may be eventual preprocessing annotations concerning a given feature (*e.g.*, `Bar`) scattered through different folders (*e.g.*, `Foo`). It is worth to notice that our approach does not make any assumptions regarding how the `JavaScript` modules are structured.

Thus, the composition-based approach handles most of the work while composing a new product and the annotative approach adds a preprocessing step preceding the real composition. Although preprocessing annotations can be used anywhere within a module, so that variability management can count solely on annotations, the feature-oriented code organization fosters the inclusion of code mostly belonging to a given feature (*e.g.*, `Foo` in its particular folder. Conversely, annotations should preferably handle fine-grained variability (*e.g.*, feature interactions handling) to avoid problems with code obfuscation [4].

4.4 IMPLEMENTATION

RiPLE-HC[3] was implemented as a plugin for FeatureIDE [66], a variability management tool designed to provide automated support to SPL development. Thus, we expanded the FeatureIDE capabilities to integrate an annotative with the native compositional approach, as a more general approach to enable variability management at implementation level. While the former enables inner-function statements to behave differently, depending on the selection of a given feature, the latter handles the inclusion or exclusion of an entire function in a product variant. In this approach, we cope with *functional interactions*, subsuming interactions that could potentially violate functional specifications [10]. It is worth to notice that there is no native support for annotations in `JavaScript` and by using this approach, the language remains as the original. For the purpose of supporting them, we made use of parsing comment lines for the defined keywords (Details about the keywords in Section 4.4.2). In addition, none of our implementations were tested against variability awareness. We next describe the implementation *architecture*, how the approach handles *coarse* and *fine*-grained variability, and the support to deal with *scattering* and *tangling*.

4.4.1 Architecture

This section discusses the architecture and components of the RiPLE-HC implementation. Figure 4.2 shows its deployment view, highlighting how the plugin relates to external entities. More specifically, such external entities comes from three different projects

[3]Available at: <https://goo.gl/Ar2cJC>

(FeatureIDE, VJET[4], and GEF4 Zest – all of them in a shadowed dashed rectangle). Next, we detail the role of each bundle.

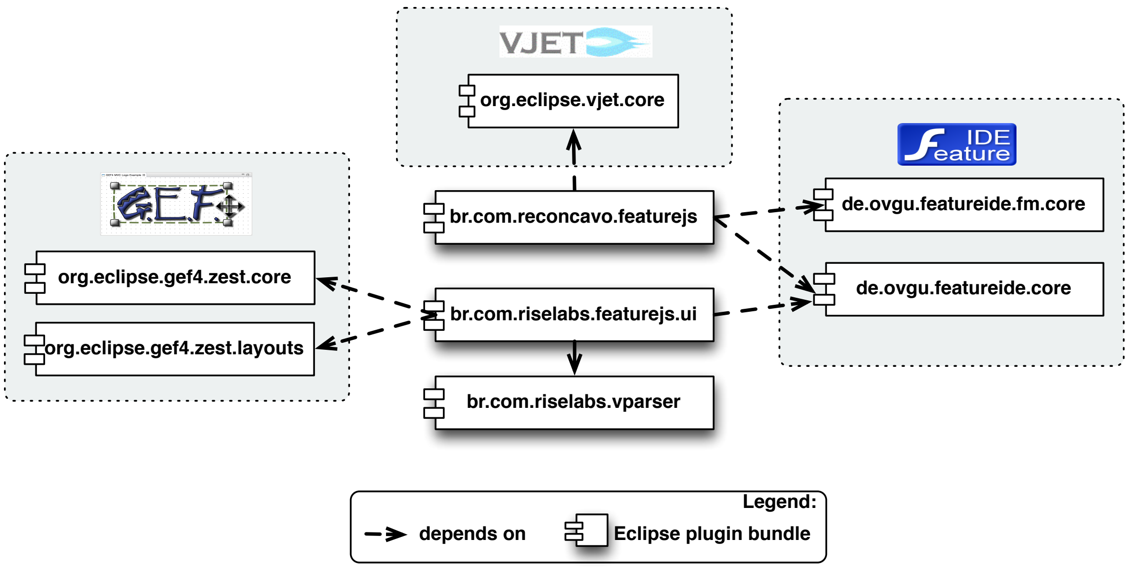

VJET
org.eclipse.vjet.core
IDE Feature
de.ovgu.featureide.fm.core
de.ovgu.featureide.core
GEF
org.eclipse.gef4.zest.core
org.eclipse.gef4.zest.layouts
br.com.reconcavo.featurejs
br.com.riselabs.featurejs.ui
br.com.riselabs.vparser
Legend:
depends on
Eclipse plugin bundle

Figure 4.2 RiPLE-HC deployment view.

`br.com.reconcavo.featurejs`: It is the core RiPLE-HC bundle. This bundle manages the hybrid SPL composition by first performing the preprocessing of the existing annotations and then the real composition of features. For instance, once a .js file for a given feature is written and the annotations referencing other features included throughout the code, this bundle is responsible to resolve which annotated blocks will remain after the build of the variant.

`br.com.riselabs.featurejs.ui`: The user interface RiPLE-HC bundle. This bundle implements views on the features interaction and annotations scattering throughout the source code. This is a helper bundle designed to support code maintenance by showing where the annotations referencing each feature are and the relationship among them as well.

`br.com.riselabs.vparser`: The variability annotations parser RiPLE-HC bundle. This bundle parses the source code to locate the annotations and collect information needed to support the user interface bundle on the construction of its visualization tools.

The external bundles are the following:

`de.ovgu.featureide.core`: The core FeatureIDE bundle. This bundle allows external bundles implementing custom composers to extend the FeatureIDE composition capabilities. RiPLE-HC relies on it to perform the composition.

[4]<http://eclipse.org/vjet/>

`de.ovgu.featureide.fm.core`: The feature model core FEATUREIDE bundle. This bundle provides default implementations for feature models, features, and configurations. RIPLE-HC relies on it to manage the product configuration.

`org.eclipse.gef4.zest.core`: The core GEF4 ZEST bundle. This bundle contains implementations of default graph-based models. RIPLE-HC relies on it to build the feature interactions graph.

`org.eclipse.gef4.zest.layouts`: The layouts GEF4 ZEST bundle. This bundle contains a set of graph layout implementations for different presentations of them. RIPLE-HC relies on it to arrange nodes and edges of the feature interactions graph.

`org.eclipse.vjet.core`: The core VJET bundle. This bundle provides IDE capabilities to support `JavaScript` faster development, such as code completion, code templates, wizards, debug support, and native type and syntax checking to identify errors through semantic validation.

4.4.2 FeatureJS: The Core Bundle

This section details the core bundle implementation (`br.com.reconcavo.featurejs`). Figure 4.3 shows how the packages and classes relate to each other, and how they relate to FEATUREIDE core entities. RIPLE-HC uses `ComposerExtensionClass`, from package `de.ovgu.featureide.core.composers`, as this is the default composer implementation provided by FEATUREIDE. The RIPLE-HC encompasses the following classes:

FeatureJsCorePlugin. This class links FEATUREIDE to RIPLE-HC plugin by creating an activator class, allowing this latter to be managed by the FEATUREIDE framework.

FeatureJSComposer. This is a **business rule class** that integrates FeatureIDE Wizard's information. It is responsible for retaining variability information for further treatments. This works by screening each file, within the allowed extensions, to treat the preprocessor directives. It also creates the containment hierarchies, and handles the software configuration build performance.

FeatureJSModelBuilder. This class is responsible for traversing each feature and the associated files, displaying them in FEATUREIDE's `FSTModel`, which represents the project structure.

FileManager. This class manages files that have been copied to remove the non-selected features, keeping only the selected ones. It is a final class which implements the singleton pattern, hence it cannot be neither inherited nor instantiated. We found it to be a safe approach to avoid likely concurrency problems. As a **business rule class**, it defines the precedence logic for each preprocessor directive, and the replacement policy. Once a directive's block code contains an error, this class handles the error, maintaining the code portion with deviation in a product variant. Meanwhile, an exception is thrown to report the occurrence.

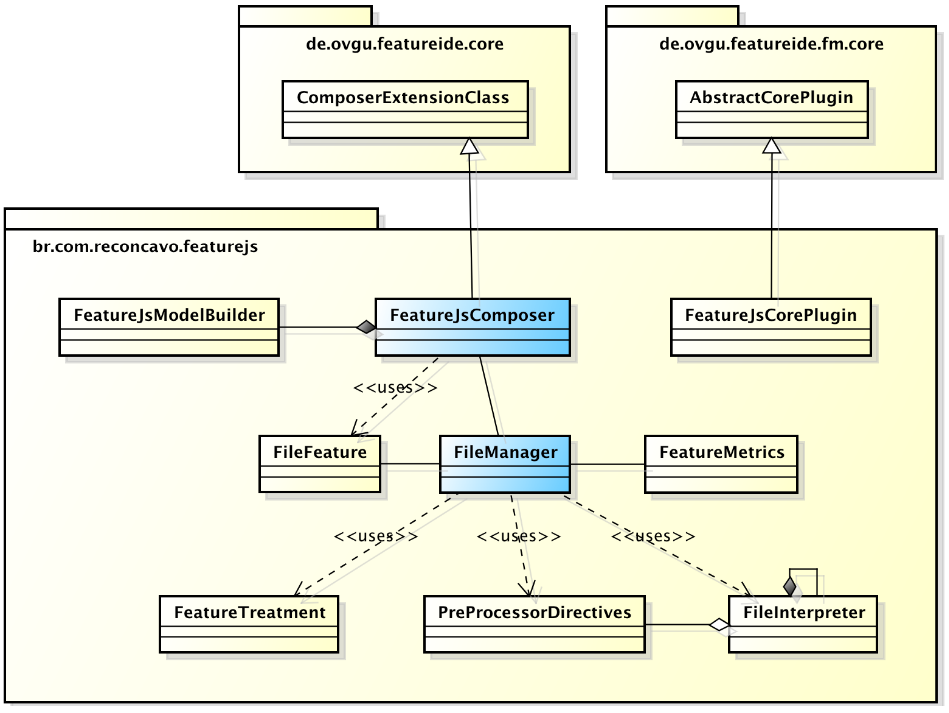


Figure 4.3 FeatureJS package and class diagram.

FileFeature. This class is responsible for identifying the feature interactions at implementation level, namely the files and features affected by other features. It retrieves all feature interaction occurrences.

FileInterpreter. This class is an abstraction of a code fragment framed by a directive. It implements a logical Doubly Linked List structured, as follows:

- The previous `FileInterpreter` is considered as a parent and known by the attribute `parentFileInterpreter`.
- The next `FileInterpreter` is considered as a child and known by the attribute `childFileInterpreter`.
- It has a pattern for the code fragment framed by a directives by the attribute `originalCode`, and its replacement; if it is needed, it is defined by the attribute `innerCode`.

FeatureTreatment. This is a final and non-instantiable class that encapsulates the treatment of each feature, by generating a specific regular expression pattern to recognize a directive. This is an utilitarian class, only accessible by its static methods.

PreProcessorDirectives. This `Enum` represents all available preprocessor directives that are considered during the feature treatment phase in a file. Five directives are available to control conditional compilation, as follows: `#ifdef, #ifndef, #else, #elif,` and `#endif`.

FeatureMetrics. This class gather metrics for measuring the complexity of the SPL. This class enables the identification of source files containing preprocessor directives. Besides, it also provides a view to list which files are affected by each feature. It is useful for maintenance purposes, as it is possible to track files that need modification when a feature is included, excluded, or modified.

4.4.3 Coarse-grained Variability

RiPLE-HC relies on the FeatureIDE capabilities to automatically create the containment hierarchy (Figure 4.1), in which there is a directory to store all the code belonging to each concrete feature. This is a FeatureIDE inner concept. While *abstract* features are dedicated to group *concrete* features and usually are named with more general terms, the concrete features are those which actually provide the functionalities' code. When a new product is to be configured, the automated product generator picks all files from the directories associated to all corresponding features and deploys the product variant in a safe and effective manner. The `JavaScript` composition is made in the file level and no assumption is made regarding the existence of methods with the same signature.

In the FeatureIDE, the variability is partially controlled at implementation level, *i.e.*, if a given file associated to a feature behaves differently depending on the selection of an external feature, it replaces the entire file associated to that feature. In programming languages such as `Java`, *refinement declarations* [30] serve as a strategy to handle changes a feature makes to a program, without changing the core code, *e.g.*, fields and methods can be added to a class, and those will be reached in a program variant only if the feature containing those refinements is selected. However, for programming languages which do not enable those declarations, such as `JavaScript`, applying such a technique to control inner-function variability would lead to a large amount of duplicated code.

In an ideal SPL, where there is a direct, one-to-one mapping between a problem domain variation and a variation point in the solution domain, this strategy would work seamlessly. However, we should assume that feature interactions can also occur at implementation level, and a single feature can be mapped to multiple code fragments.

4.4.4 Fine-grained Variability

FeatureIDE allows the representation of constraints between features, controlled by the configuration view. In such a view, a configuration either enables or disables the selection of a given feature according to the constraints associated to it. RiPLE-HC relies on such control for the composition and adds its own support to handle such dependencies with low-level annotations.

For example, considering an SPL project called `algorithms.js` (Fig. 4.4) – which has a root feature `Algorithms` representing the domain under analysis – a set of mandatory features including a *concrete feature* called `Knapsack` and an *abstract feature* called `Queue`. `PriorityQueue` and `SingleQueue` are alternative children of `Queue`. Therefore, one and only one of them can be included, if their parent feature is included in a configuration.

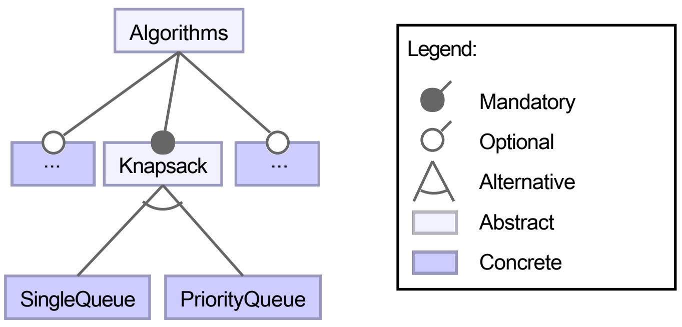


Figure 4.4 `algorithms.js` SPL sample feature model.

The feature `Knapsack` have a containment hierarchy which holds a number of items, which should behave differently depending on the `Queue`'s sub-feature selection. Listing 4.1 illustrates how the RiPLE-HC deals with the use of preprocessor directives (annotative approach) to manage variability at implementation level.

The directives in the source code delimit blocks of program that are compiled only if a specified condition is `true`. They may be employed to generate different product variants by assembling the code fragments in cases where more than one product configuration includes the same `JavaScript` file, but a given *function* behaves differently depending on the feature selection. The main reason is that composition rules for augmenting functions with new properties in `JavaScript` is not always safe [67]. In addition, this strategy may reduce the maintenance effort, as the business rules from a single function will be self-contained in a single file.

In the `algorithms.js` SPL example, after binding the variants, the variable `queue` declaration statement will be set differently, depending on the selection of either feature `PriorityQueue` or `SingleQueue`. This shows how RiPLE-HC might anticipate program-level customization of core assets for a custom product to an earlier phase in the development cycle. Besides, it controls and manages variability at both model and implementation levels to handle product enhancements.

```
function knapsack(items) {
                                  ⋮
  var queue;
//#ifdef PriorityQueue
  queue = new PriorityQueue();
//#elif SingleQueue
  queue = new SingleQueue();
//#endif
                                  ⋮
  });
```

Listing 4.1 Excerpt code from *Knapsack.js* (feature `Knapsack`).

In summary, the syntax of the use of annotations was inspired by the ANTENNA[5] syntax. The keywords are supposed to be inserted in the single line comments ("//") in any place of the .js file. The available keywords are `#ifdef, #ifndef, #else, #elif,` and `#endif`. Each annotation block must be closed properly with the `#endif` keyword after being opened with `#ifdef` and `#ifndef`. The keywords `#else` and `#elif` can be used to provide an alternate behavior to program depending on the result of the evaluation of the annotation block opening. These keywords keep the meaning of ANTENNA's syntax. However, the opening keywords take only one feature instead of a logical expression. Such a restriction limits the flexibility/expresiveness of the annotations, yet in the same time corroborates to the use of discplined annotations [27].

4.4.5 Scattering Support

RIPLE-HC implementation provides support for increasing the awareness of the developers regarding the SPL feature interactions and the scattering of annotations. Such support consists of two views, the *Feature Interactions View* and the *Scattering Tree View*. We decided to implement these views after the feedback of developers who used the approach during *(i)* the case study we present in the Chapter 5 and *(ii)* in the experiment we present in Chapter 7. Next, we detail each of these views.

The *Feature Interactions View* (Figure 4.5) consists in a tabbed panel with directed graphs in each tab. The *left-most* tab exhibits the overall interaction of the selected project. The remaining tabs exhibit a product-based graph containing only the selected features for each product configuration available in the project. These graphs represent the presence of `#ifdef` macros throughout the SPL source code, which we called *feature interactions* (*i.e.*, we do not deal with unexpected feacture interactions that usually cause problems in software systems [10]). Nodes represent the product line features, whereas the directed edges indicate the presence of conditional compilation annotations in the code implementing the feature of the source node concerning the ending node feature. Numbers over the edges account how many macros were found in the feature code relating both features. We faced no problems while generating the graphs for the systems addresed in the experimental study we present in the Chapter 5. We did not generate the graph with a huge number of features.

The *Scattering Tree View* (Figure 4.6(a)) shows all the concrete features of the product line as roots of a tree. Concrete is a FEATUREIDE inner concept. While *abstract* features are dedicated to group *concrete* features and usually are named with more general terms, the concrete features are those which actually provide the business rule implementation. All the `JavaScript` files in the respective feature folder containing annotations are shown as child of feature root node. Besides, each line containing an annotation is a child itself from the file nodes. This view is connected to the code Editor (4.6(b)). A double-click in the `#ifdef` macro node in the tree opens the editor in the corresponding line. Thus, the *Scattering Tree View* allows the developer to keep track of the exact location of each conditional compilation annotations in the current project.

For the sake of understanding, we recall the `algorithms.js` SPL (Figure 4.4), but this

[5]Examples available at <http://antenna.sourceforge.net/wtkpreprocess.php>.

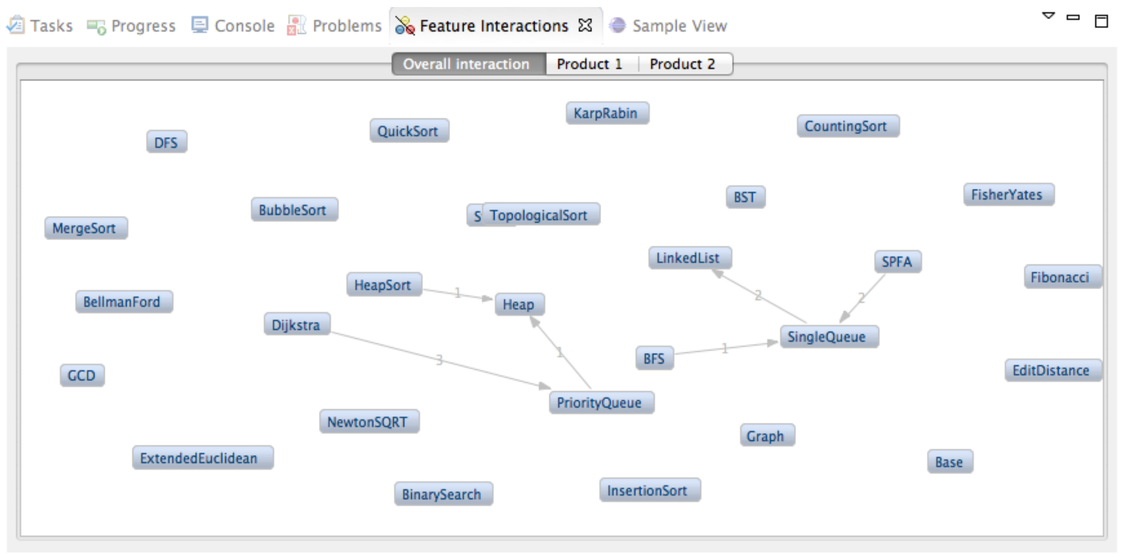
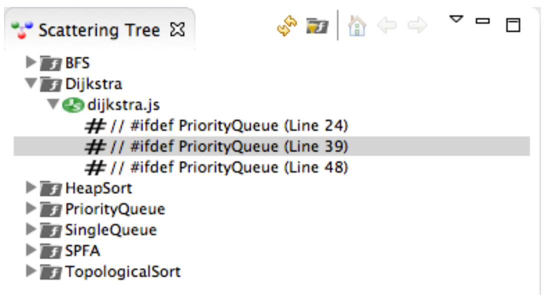


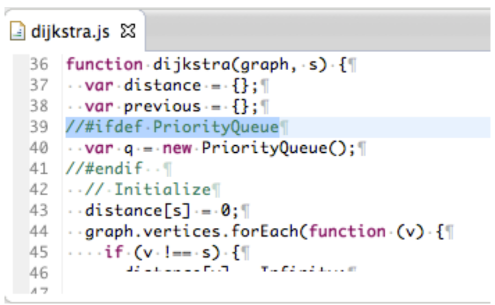
Figure 4.5 Feature Interactions View of the RiPLE-HC toolkit.

(a) Scattering Tree View



```
▶ BFS
▼ Dijkstra
    ▼ dijkstra.js
        # // #ifdef PriorityQueue (Line 24)
        # // #ifdef PriorityQueue (Line 39)
        # // #ifdef PriorityQueue (Line 48)
▶ HeapSort
▶ PriorityQueue
▶ SingleQueue
▶ SPFA
▶ TopologicalSort
```

(b) JavaScript Editor



```
36 function dijkstra(graph, s) {
37   var distance = {};
38   var previous = {};
39 //#ifdef PriorityQueue
40   var q = new PriorityQueue();
41 //#endif
42   // Initialize
43   distance[s] = 0;
44   graph.vertices.forEach(function (v) {
45     if (v !== s) {
46
```

Figure 4.6 RiPLE-HC visualization toolkit related environments.

time refer to the concrete optional features `Dijkstra` and `PriorityQueue`. The directives in the source code delimit blocks of program that are compiled only if a specified condition is `true`. It enables modifying the behavior of a function or a statement, depending on the selected feature. Therefore, assembling the code fragments that corresponds to a configuration generates a product variant. For some reason, a variable `q` should be declared as an instance of `PriorityQueue` only if this feature is selected for the product variant. A developer added `#ifdef` annotation in the *line 39* of the file *dijkstra.js* that implements the functionalities of the feature `Dijkstra`.

Figure 4.6 illustrates how the *Scattering Tree View* shows the annotation (a) and the `JavaScript Editor` highlights the line (b) after a double-click in the respective annotation in the tree. In this example, after binding the variants, the variable `q` declaration statement (line 40) will be available depending on the selection of the feature `PriorityQueue`. At the same time, Figure 4.5 shows an edge between the features `Dijkstra` and `Priority Queue` labeled with 3, which is the number of children in the `Dijkstra` tree (Figure 4.6(a)).

Table 4.1 RiPLE-HC inherited characteristics from compositional and annotative approaches.

Compositional Approaches	**Annotative Approaches**
✘ *Drawbacks*	
(+) Coarse granularity	(+/–) Code obfuscation
(−) Poor feature interactions handling	(+/–) Separation of concerns
(+/–) Difficult adoption	
✔ *Benefits*	
(+/–) Modularization	(+) Simple programming model
(+/–) Traceability	(+) Fine granularity
(+/–) Disciplined variability support	(+) Ease to use
	(+) Strong feature interactions handling

4.5 INHERITED CHARACTERISTICS

We built the RiPLE-HC upon influences of previous hybrid approaches, namely FeatureC++ [2] and FeatureHouse [1], which unintentionally allow the use of annotations together with composition. In fact, compositional and annotative approaches pushed the state-of-the-art and practice, respectively, to another level. While the former has grown significantly and as a consequence has gathered much attention by researchers [10], the latter is one of the most used approaches in the implementation of SPL in industry.

Table 4.1 enumerates benefits and drawbacks of RiPLE-HC. Each table item has a mark indicating whether the RiPLE-HC inherited the characteristics of those approaches completely (+), partially (+/–), or ignored them (−). It is not proven that hybrid approaches inherit all valuable characteristics from compositional and annotative approaches. Kästner and Apel [4] advocated that although it does not automatically fix all disadvantages of either approach, some benefits from both still holds after blending. Additionally, we did not carry out an evaluation of any benefits or drawbacks originated specifically from the `JavaScript` language – it would require software engineers with deep knowledge of both the language and at least some of these characteristics, which we were not able to recruit a reliable sample. In fact, these assertive are rather general ones regarding all languages.

Additionally, from Table 4.1, it can be seen that the RiPLE-HC resorts from better modularization (SoC) to provide better handling of *feature interactions* (Listing 4.1). Some sort of scattering should not be seen as a design flaw when kept under a defined threshold [68]. Thus, although the scattering code traceability and the maintenance of the variability may be affected by the scattering introduced by conditional compilation, the provided tool support minimizes such effect.

Benefits and limitations of compositional approaches: The compositional approaches implement features in distinct modules (*i.e.*, it aims to eliminate code tangling).

The benefits of using them include: *(i) modularization* – they compose selected modules to bind a product instance; *(ii) traceability* – it is straightforward the location of the code implementing each feature of the feature model; and *(iii) language support for variability* – the languages are designed in a disciplined and well-defined way being aware of variability. As the drawbacks, they entail *(i) feature interactions handling* – although there are significant gains in terms of modularization, handling feature interactions is still a challenge in compositional approaches; *(ii) coarse granularity* – which is too restrictive for implementing variability, especially in the occurrence of *feature interactions*; and *(iii) difficult adoption*, which is usually for the introduction of new language concepts and raised complexity of the SPL implementation [4].

Benefits and limitations of annotative approaches: The benefits of using an annotative approach include: *(i)* the *simple programming model* – code is annotated and removed; *(ii)* the *fine granularity* – arbitrary code fragments can be marked; *(iii)* the *variability despite the feature interactions* – they are able to handle the interaction between dependent features. Conversely, it also has its drawbacks as follows: *(i)* the *separation of concerns* – the modularity and traceability are likely the biggest problems with preprocessors; and *(ii)* the *code obfuscation* – the use of preprocessors at a fine granularity with nesting (*e.g.*, indisciplined annotations [27]) can make difficult to read and follow the control flow of the code [4].

4.6 CHAPTER SUMMARY

This chapter presented the lack of technical support to SPL engineering in `JavaScript` software systems, which motivated us to develop the novel hybrid composition approach to fill such gap called RiPLE-HC. Thus, we introduced the concept of the approach that blends compositional and annotative approaches to handle fine and coarse granularity. Later, we detailed both the architecture and the implementation of the plugin as and extension of the FeatureIDE capabilities, including the visualization support for the maintenance while dealing with the scattered annotations dependent features. Finally, we carried out a brief discussion on the inherited characteristics from compositional and annotative approaches.

Next chapter presents a set of empirical studies aimed at to assess the viability and the scalability of the approach.

Chapter 5

RIPLE-HC EVALUATIONS

This chapter discusses the empirical studies conducted to assess the feasibility – as the ability to handle variability in real world `JavaScript` systems (Section 5.2) – and the robustness – as the ability to scale to large `JavaScript` systems (5.3) – of the RiPLE-HC approach. They were carried out in industry and academic settings. In addition to the previous chapter, they covered the **Research Goal 1** (pg. 5).

The chapter consists of five main sections. Section 5.1 discusses the planning of our preliminary studies. Section 5.2 discusses the industrial case study – a supervised migration of a set of K-12 learning objects products into SPL [20]. Section 5.3 discusses the manual refactoring of open-source systems into SPL versions from six selected *open-source* projects [21]. Section 5.4 highlights strengths and weaknesses identified regarding the proposed approach. Finally, Section 5.5 discusses the identified threats to validity during the preliminary evaluation.

5.1 GOAL-QUESTIONS-METRICS (GQM)

In this section, we present the planning of two empirical studies that evaluated the proposed approach. Next, we present our Goal-Question-Metric (GQM) statement [69] applied to formalize the planning, execution, and report of the results of the studies:

> Analyze RiPLE-HC for the purpose of *characterization* with respect to its *feasibility* and *scalability* from the point of view of the *software engineers* in the context of *industrial projects.*

We carried out two preliminary empirical studies with the goal to observe RiPLE-HC *feasibility* and *scalability*. Both studies have the point of view of *software engineers*. The first study is a case study in industry, which aims at showing the ability of RiPLE-HC to handle variability in a real world context. Thus, it addresses the *feasibility* and we aimed to answer the following research question:

RQ1: Does RiPLE-HC handle variability in `JavaScript` software projects in industrial context?

In this first study, we discuss the results regarding the characterization of an SPL built and the effort needed to complete the project. In other words, we underlay our discussion based on measures, such as the number of features and the time spent in the development of the variants to answer the research question.

The second one is an empirical study with open-source systems from different sizes and domains, which aims at showing whether the proposed approach could be suitable to projects from different application domains and sizes. Therefore, it addresses the *scalability* and we aimed to answer the following research question:

RQ2: Does RiPLE-HC scale to systems from different domains and sizes?

In this second study, we discuss the results regarding the characterization of SPL built and the effort needed to complete their build. In other words, we ground our discussion based on measures, such as the number of features, modules, annotation directives, scattering of the annotations and the time to build a variant selecting all the available features to answer the research question. Therefore, the metric used to measure scalability is then the *build-time*.

5.2 INDUSTRIAL CASE STUDY

We first conducted a case study in the industrial setting. More specifically, we supervised the development of an SPL project in a software development company based in Salvador, Brazil, named *Recôncavo Institute of Technology*.[1] Next, we describe the experience and the feedback we gained from such a partner.

5.2.1 Domain

The advent of Web-based digital interactive technologies has led teaching and learning methodologies to a next instructional setting level. The goal is to use such technologies to stimulate the learners' knowledge formation and retention. This instructional technology concept is known as "learning object". Learning objects are generally understood to be digital entities deliverable over the Internet, making them accessible and usable by multiple users in parallel [70]. They have a great potential for reusability, generativity, adaptability, and scalability. This principle is based upon the idea that a course or lesson can be built from reusable instructional components which can be built separately but modified to user's needs [71].

In this scenario, our industry partner has developed a series of learning objects, intended for K-12 education, as a subcontractor for one of the leading provider of online educational content to K-12 schools in Brazil.

The previous stage to adopt an SPL approach was that applications were usually developed one at a time. Applications were implemented in the `ActionScript` language,

[1]<http://www.reconcavotecnologia.org.br>

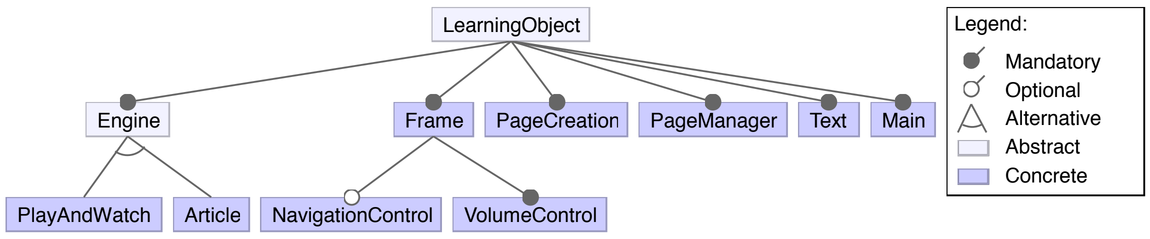


Figure 5.1 Excerpt from the *MDC Learning Objects* feature model.

until they decided to turn their applications platform-independent. As of this point, new applications would be implemented in `HTML5` - mainly `JavaScript` and `HTML` -, so that their applications could reach a greater number of customers, due to the cross-platform capabilities of these technologies.

As reuse was merely opportunistic in their development cycle, the problem with cost and scale was imminent. Since opportunistic reuse can be more expensive than a systematic reuse of software artifacts, our partnership enabled their software development process to transition to an SPL approach. Hence, the SPL selected for this study was part of this project, in the Web-based learning systems domain.

5.2.2 Data Collected

The project, called *MDC Learning Objects*, comprises a set of `42` features. Figure 5.1 shows an excerpt of the feature model of the *MDC Learning Objects*. The core features has, together, around `3.7` Thousand Lines of Code (KLOC). The `MDC` project has `23` boolean configuration variables and can, in theory, be deployed in over `3800` different configurations. However, it is worth saying that such number is not realistic, due to a set of very specialized requirements for each individual learning object, which demands concrete features to be implemented and selected prior to deliver a product configuration. That is, we can generate thousands of different configurations, but for a single product variant to run properly, a series of product-specific features should be implemented, so as to match specific requirements. Those features mainly include the management of metadata, such as the media scripts, particular to every single learning object, and as such must be shared with other objects at all.

Thus, for this particular case study we consider three different products, fully functional, generated from the core asset base. Due to the mutual confidentiality and non-disclosure agreement, we cannot describe applications' name, and some features' name are also omitted. Thus, we will herein call the product variants as APP1, APP2, and APP3. Tables 5.1 and 5.2 show data about our target SPL. The former shows code metrics extracted from each product[2], such as Lines of Code (LOC), number of files, functions, number of declarative (which name a variable, constant, or procedure, and can also specify a data type) and executable statements (which initiate actions). The latter shows the product configurations, highlighting the variable features selected for each product.

[2]Metrics gathered with the *Understand* tool, available at <http://www.scitools.com/download>

Table 5.1 Products metrics generated from the SPL.

	LOC	**Files**	**Functions**	**DS**	**ES**
Core	3,778	47	421	796	2,003
APP1	5,568	62	510	972	3,243
APP2	5,188	61	518	964	3,039
APP3	6,520	63	514	978	4,027

DS: Declarative Statements, ES: Executable Statements.

Table 5.2 Variant configuration matrix.

#	**MDC Features**	**APP1**	**APP2**	**APP3**
1	PageCreation	✔	✔	✔
2	WatchPage	✔		✔
3	PlayPage	✔		✔
4	ArticlePage		✔	
5	MatchColsTask		✔	
6	FillInTask		✔	
7	SubtitleManager	✔	✔	✔
8	VideoManager			
9	AnimationManager	✔	✔	✔
10	AudioManager	✔	✔	✔
11	NavigationControl	✔	✔	✔
12	Article		✔	
13	BP		✔	
14	PlayAndWatch	✔		✔
15	ACJC	✔		
16	AGR			✔
17	Animations	✔	✔	✔
18	Background	✔	✔	✔
19	Buttons	✔	✔	✔
20	Environment	✔		✔
21	Locutions	✔	✔	✔
22	Music	✔		✔
23	Effects	✔	✔	✔

✔: Selected feature.

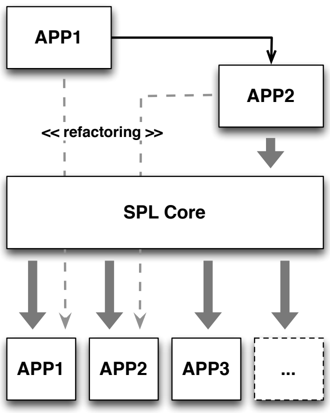


Figure 5.2 Reactive SPL process adopted.

Table 5.3 Development time.

Application	Development Time
APP1	720 engineer-hours
APP2 + SPL Core	448 engineer-hours
APP1 Refactoring	160 engineer-hours
APP3	122 engineer-hours

The MDC project employed a reactive SPL approach [72], in which a single product is subsumed into an SPL. In this approach, not all possible variations are implemented beforehand, but instead only those variations needed in current products are implemented, in an incremental fashion. Figure 5.2 shows the process, which is explained next.

The *APP1* was the first application to be implemented. Based on this first application, *APP2* was implemented. Next, by analyzing existing assets, and defining the SPL commonalities and variabilities, it was possible to define the *SPL Core*. It was necessary to refactor part of *APP1* into the core architecture. Then, the following application could be developed, by systematically reusing the core, and integrating the product-specific parts. Unfortunately, the absence of a managed software process in the culture of our industry partner left us with no data available to answer questions, such as "What kind of refactorings were used in the SPL extraction process?", "How the correctness of the refactorings was evaluated?", "Does any kind of tool support was used in the process?" or "What was the used refactoring procedure?".

At the end of this evaluation, the project MDC deployed three different products, sharing parts of the implementation. Table 5.3 shows the time spent in the MDC SPL implementation, comprising the implementation of both core and products-specific parts. All three applications used the hybrid composition and the choice of the three products was made by the comercial demmand of contractors.

The initial development of the *APP1* took 45 working days of two software engineers, working around *8* hours a day each on this project, for a total of around `720` engineer-hours. The work in this first application comprised tasks such as domain analysis, design, and implementation of the application, identifying opportunities for reuse, and customer validation of the implemented features.

Next, developers took `13` working days, for a total of `208` engineer-hours, to build the *APP2*. In order to identify reuse opportunities in both applications, and systematize what could be leveraged as both common and variable parts, it took an another `15` working days. A total of `240` engineer-hours was employed to build the *SPL Core*, turning the project into a reusable platform.

After establishing the *SPL Core*, it took an additional *ten* working days to refactor the *APP1* for performance increases, and accommodate changes in code so as to make the applications run smoothly on the Android platform, a requirement that was identified during *APP2* implementation. This task took a total of `160` working hours. *APP3* was developed within a `7`-day period, and took developers about `122` engineer-hours.

Although we planed to investigate the gains in the development, such as the impact in time, costs, and maintainability, we were not able to do so. This case study was interrupted prior to the execution of scheduled activities due to the ending of the contract between our partner and their contractor for commercial reasons. Additionally, employees we had contact with left the company and we could collect information whether they kept using RiPLE-HC or not.

Next, we discuss the first research question of this study.

5.2.3 RQ1: Does RiPLE-HC handles variability in `JavaScript` software project in industrial context?

This case study has suggested that the RiPLE-HC *can handle variability in industrial* `JavaScript` *software projects.* In addition, it may positively impact both the development cost and time, and the maintainability of the overall SPL. Our industry partner reported gains in development time, what might result in order of magnitude cost reductions in next products' releases. By looking at Table 5.3, we may observe a reduction in the development time employed in the third product, although it is larger in size than preceding ones, as listed in Table 5.1. This fact corroborates with literature that states that SPL projects reach a break-even point around three systems [73, 74]. In fact, as the core platform was well-established, the time demanded was mainly dedicated to build the product-specific parts.

The capability of the strategy to manage variability at implementation level, by the blend of annotation and feature composition, is way more significant than simply handling variability at modeling level. Especially in the development with `JavaScript` and `HTML`, in which composition rules are not robust enough, counting solely with *mixin* operations to augment functions properties is not either safe nor cost-effective. The use of preprocessor directives also enables inner-function statements to behave differently depending on the feature selected for a given product configuration. Thus, the feature-oriented approach can provide the adequate support for changing an entire function to a final product.

5.3 RIPLE-HC WITH OPEN SOURCE SYSTEMS

We manually migrated six open-source systems into SPL by using the RiPLE-HC approach. After the industrial case study, we would like to know whether RiPLE-HC could handle systems from different domains and sizes. Therefore, the goal of the study was twofold: *(i)* to assess whether our approach could be used in other contexts than the one it was formerly conceived; *(ii)* to assess the scalability of the approach to `JavaScript`-based systems with size representative of real world applications.

The process of transformation of each system consisted basically of three phases: *(i)* feature selection, *(ii)* refactoring, and *(iii)* build. The feature selection relied on the available documentation found in the project website and (or solely in the information found in the) GitHub. The refactoring consisted in moving the `JavaScript` modules that concern to each feature to their respective code hierarchy and the annotation of scattered pieces of code. The build was to make sure that the composition had been successfully accomplished, as well as the preprocessing of the annotation blocks. In this case, we randomly chose annotated blocks to check whether the blocks were removed or not. We took around one month to perform all refactorings.

Table 5.4 Characterization metrics of the target systems, extracted from the *qualitas.js* corpus.

System(v)	**LOC**	**# Modules**	**# Features(CT)**	**Domain**
`algorithms.js` (0.20)	1,594	29	28 (6)	*miscelaneous*
`jasmine` (2.0.0)	2,956	48	4 (-)	*testing*
`floraJS` (1.0.0)	3,325	26	18 (-)	*simulation*
`video.js` (4.6.1)	7,939	38	13 (-)	*video player*
`TimelineJS` (2.25.0)	18,237	89	15 (-)	*web library*
`brackets` (0.41)	122,971	403	13 (1)	*text editor*

v: version; **CT:** Number of cross-tree constraints;

The systems were selected from *qualitas.js* corpus of real world `JavaScript`-based systems dataset [12]. Silva *et al.* [12] composed the dataset with the most popular systems from GitHub. We selected systems to refactor into SPL ranging from small to large systems. Table 5.4 shows descriptive metrics reproduced from the *qualitas.js* – such as *(i)* lines of code (LOC) and *(ii)* number of modules of each system – and the metrics extracted from the SPL versions of the systems, such as, *(iii)* the number of features and cross-tree constraints.

Table 5.4 shows the characterization of the refactored versions of each system as follows: *(i) algorithms.js-SPL* has 28 concrete features and 6 cross-tree constraints – when a feature from one branch of the feature model imposes one restriction to the selection of (or is imposed by) other feature; *(ii) video.js-SPL* has 13 features and no cross-tree constraints; *(iii) floraJS-SPL* has 18 concrete features and no cross-tree constraints; *(iv) jasmine-SPL* has 4 concrete features and no cross-tree constraints; and the *(v) TimelineJS-SPL* has 15 concrete features and no cross-tree constraints. The rationale for some important aspects regarding the use of RiPLE-HC is discussed next.

RiPLE-HC slightly modified how the `JavaScript`-based systems should be structured. In comparison with the current *state-of-the-practice*, instead of using an *ad-hoc* organization (*i.e.*, there is no standard followed by all the projects), RiPLE-HC requires a more systematic way to organize the source code, regarding the features. Regardless of the notable differences in the code organization, there was no additional effort for feature code location in maintenance tasks.

5.3.1 Granularity

RiPLE-HC enables developers to adjust the granularity of the variability by annotating the corresponding scattered variability. Thus, at least two main levels of granularity could be experienced: modules dedicated to a given feature processed by composition, and the scattered feature code by pre-processing the conditional compilation annotations.

While extracting the SPL from the target systems, we did not experience any issues but those regarding the build of the products from systems with annotated nested blocks. In addition, we realized that some systems (*e.g.*, `algorithms.js` and `jasmine.js`) lack a systematic source code organization, which means that most modules are placed in the same folder. In fact, as soon as the systems increase in size and/or complexity some folder organization is used, according to modules' functionalities, which recalls to the feature-oriented way to organize the code. Although such a characteristic was not statistically checked, the way the code is organized may be a factor of influence on the demanded effort to migrate a set of single systems to an SPL with RiPLE-HC.

5.3.2 Scalability

Table 5.5 shows metrics collected in order to evaluate the scalability of the approach. More specifically, the metrics are *(i)* the number of annotated blocks processed; *(ii)* the number of files with annotated blocks; *(iii)* the average time of build (measured in seconds). All these metrics were calculated by the RiPLE-HC plugin itself. We executed a full product build with all the features selected 10 times to compute the average of time needed. Figure 5.3 shows the time to build in each iteration. All iterations were executed using a 2.3GHz Intel Core i5 processor, with 16GB 1333 MHz DDR3 memory module. We used the build time as a measure of scalability because it shows whether the time needed grows indefinitely or it stays in reasonable range, in the sense that it would not make the project unfeasible.

To investigate scalability issues, we included in the analysis both small and large-sized `JavaScript` projects. The *qualitas.js* dataset was built with the most popular repositories from *GitHub*. We believe there is no `JavaScript`-based project of size and variability equivalent to the Linux kernel, therefore we also believe that such systems are representative of the scalability demmanded by the real world systems. Moreover, we faced no difficulties apart from to extract SPL from the `brackets` project (*i.e.*, the second biggest project in the corpus) with nested annotated blocks. This problem can happen when more than one feature impose behavior changes in a piece of code. The implications of do not

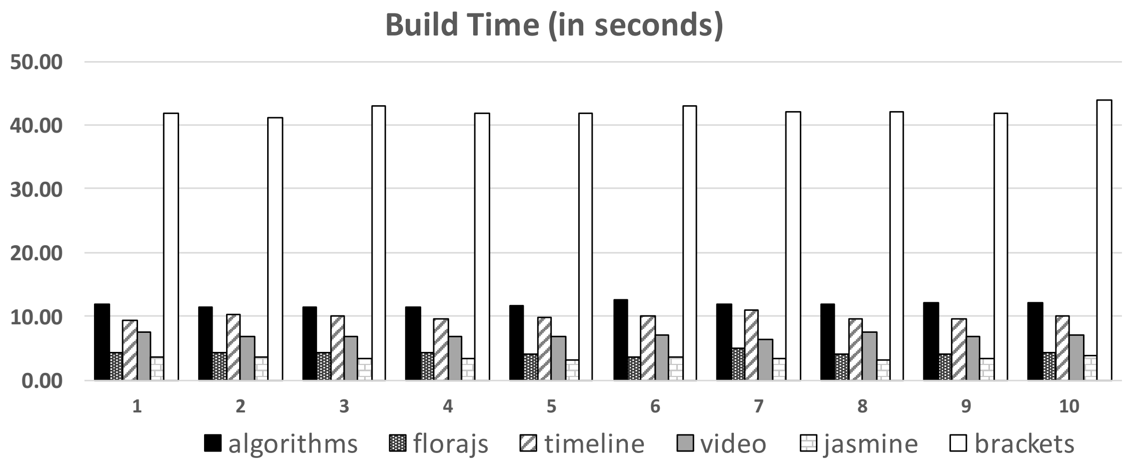


Figure 5.3 Case studies build time in each iteration.

have support for nested annotated blocks is that developers are forced to use discplined annotations [27]. When the nested blocks were left aside, the build occurred in around 40 seconds. We could observe that the measures *number of features* and *number of existing annotated blocks* may impact on the time to build as both yield more `I/O` operations. For instance, `algorithms.js` took more time than the remaining systems smaller than `brackets`. Additionally, the build time gets larger as the number of annotated blocks increases. The case studies showed that RiPLE-HC can provide support to handle most of the `JavaScript` projects, since the case studies successfully accomplished are representative of the corpus, given that about 75% of them are smaller than 4,85 KLOC in size, and 50% of them are smaller than 1,3 KLOC. Next, we discuss the second research question of this study.

Table 5.5 Target systems characterization metrics.

System(v)	**# Directives**	**# Files**	**Build(s)**	**Domain**
`algorithms.js` (0.20)	6	4	11.89	*programming*
`jasmine` (2.0.0)	14	4	3.52	*testing*
`floraJS` (1.0.0)	16	2	4.27	*simulation*
`video.js` (4.6.1)	29	10	7.03	*video player*
`TimelineJS` (2.25.0)	75	6	9.98	*web library*
`brackets` (0.41)	107	19	42.27	*text editor*

Directives: Number of annotated blocks processed; **Files:** Number of files with annotated blocks; **Build:** Average of time to build; **s:** seconds.

5.3.3 RQ2: Does RiPLE-HC scales to systems of different domains and sizes?

This case study has suggested that RiPLE-HC *can scale to systems of different domains and sizes.* Even though this evaluation has unveiled some limitations of the approach (*e.g.*, nested blocks), the different characterization of the used systems allowed us to observe how the approach can be used to manage variability in a set of real world open-source systems with size ranging from small to large in different domains. Additional work is needed to overcome the discussed limitations, but the results are enough to encourage further improvements in the approach and tool support.

5.4 STRENGTHS AND WEAKNESSES

After these two empirical studies we can enumerate some lessons learned throughout the evaluation process, as well as the strengths and weaknesses of RiPLE-HC.

The use of RiPLE-HC may serve as a useful resource to foster SPL development in `JavaScript`-based systems. The implemented hybrid approach enables the systematic reuse of code between products by the separation of the software functionalities in features. Besides, by mapping features and preprocessor directives with the help of the auxiliary views, RiPLE-HC unveils the set of files where annotation concerning each feature can be found, which in turn might improve maintenance tasks.

On the other hand, the prototype nature of the RiPLE-HC plugin ended up by preventing a more robust experience of the approach. More specifically, the use of comment blocks to introduce the annotations is definitely not the best way to it, but rather the easier to accomplish. A better option would be extend the language to accommodate the annotation keywords, as well as built an preprocessor with support to solve logical expressions in the variability points (*i.e.*the begging of each annotation block) and a proper engine to solve nested annotation blocks.

5.5 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of the evaluation. We believe that presenting such detailed information may contribute to a clear understanding of what is being presented herein. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion* validity.

5.5.1 External Validity

As the main threat to the validity of this study, the limited and constrained evidence we have at our disposal prevent us from drawing general conclusions on the results. However, the studies conducted serve the purpose of gathering initial evidence on the feasibility and the scalability of the method and its tooling support. They also served to gather insights about open rooms for improvement in the strategy, as well as further investigation.

5.5.2 Internal Validity

The selection of features to the refactoring in the open source systems was made by only one person without any revision. Such fact may pose a threat to the validity of these studies since the extracted features may not be representative of the variable characteristics of the real world scenarios. Hence, we included systems of different domains trying to mitigate such lack of industrial standpoint analysis.

5.5.3 Construct Validity

The main gathered data refers to the development time as a function of the product variant size. It is an important area of concern. Given that there was not previous data to serve as baseline values, the reduction in development time (industrial case study) for the n-ary variant releases might be caused by a maturation effect. The issue observed while preprocessing nested annotated blocks, might also lead to flawed results and must be considered in future evaluations.

5.5.4 Conclusion Validity

It is worth mentioning that these studies were not designed to draw quantitative conclusions based on descriptive statistics, for instance, regarding the scalability of the tool support. However, we relied our discussion concerning the benefits and drawbacks of the use of RiPLE-HC solely in the data gathered and the informal feedback of our partner. This fact serves to a clear purpose of reinforce conclusion validity.

5.6 CHAPTER SUMMARY

This chapter presented empirical studies carried out to evaluate the proposed hybrid approach supporting feature-oriented `JavaScript`-based software systems. The case studies were carried both in the industrial and open source systems. While the former addressed the viability in the real development context, the latter addressed mainly scalability with open source projects.

These studies showed that RiPLE-HC can handle real-world systems from small to large-sized projects, as well as systems from different domains. As expected from the literature, the time needed for building a new variant seems to be associated with the number of features defined and the number of existing annotated blocks. Scalability problems were faced with nested blocks, as such, they are not recommended in the current stage of the prototype implementation. Additionally, we observe that even with no systematic way to structure the code, as soon as the systems increase in size, the project structure tends to assume characteristics of feature-oriented organization, which may indicate that larger projects might benefit from this novel approach. Moreover, the study in the industry showed the payoff would occur in the third product, which required only 17% of the engineer-hours of the first product.

Next chapter presents the family of experimental studies we carried out to investigate the impact of FOSD variability representations on program comprehension.

PART IV

VARIABILITY IMPLEMENTATION COMPREHENSION

Chapter

6

A FAMILY OF EXPERIMENTS ON PROGRAM COMPREHENSION

This chapter presents a family of experiments we carried out to address the **Research Goal 2** of this thesis. More specifically, the goal of these experiments is to gather evidence regarding the influence of different variability representations on program comprehension of the feature-oriented software. This chapter consists of three main sections. Section 6.1 presents an overview of the family of experiments by presenting the context of each member of the family. Section 6.2 discusses the planning of the family of experiments, including the characterization of the *Target Systems* used for experimentation in this thesis. Section 6.3 shows variations in the experimental design throughout the carried studies.

In the end of this chapter, we should have a macro overview of the studies planning, the understanding of how they were organized, as well as a discussion about the aspects of program comprehension in feature-oriented software development addressed in the following chapters of this thesis.

6.1 FAMILY OVERVIEW

This section presents the family of empirical studies on the comprehension of software in the presence of variability. Henceforth, we call the family of experiments as *Variability Implementation Comprehension Comprehension (VICC)).* We planned and executed four studies as part of the VICC family, out of which three were controlled (*quasi-*)experiments (quantitative studies) and one a focus group (qualitative study). Each study is named after the family by the VICC*i* acronym, where `i` is an integer indicating the order to which the study was carried out.

Given the lack of studies on the differences regarding the comprehension of software variability implementation with different paradigms (Chapter 3), we took the experience on carrying out experiments on program comprehension reported by Siegmund *et al.*[28] as an inspiration to compose this family of studies.

The experimental studies were carried out with Software Engineering students with different levels of expertise (*e.g.*, undergraduate, master, and Ph.D.). Each experiment is supposed to address different comprehension aspects on the software maintenance phase with different systems, which we present later in this chapter. More specifically, the experiments' tasks consider two of the most used programming languages nowadays, namely `JavaScript` and `Java`, as well as its respective feature-oriented variability representations, respectively, the proposed RiPLE-HC [21] and FeatureHouse [1].

The reason we split several tasks through a family of experiments is explained next. We may observe a series of issues regarding working time and amount of tasks assigned during during experiments [18]. For example, fatigue due long periods of concentration is frequently cited as a confounding parameter in software engineering experiments and the recommendations found are of sessions no longer than two hours [15]. As we need several tasks to address the impact of the variability representation during maintenance, it is infeasible and unsuitable to perform all of them in only one experiment. Otherwise, experiment's participants would take longer than two hours to finish their assignments. In fact, we are aware that there is a number of maintenance tasks that should be consider, however some of them will remain uncovered by the executed experiments. Next, we describe the context of each experiment.

VICC1 – the first experiment of the family addresses the concept location problem [75]. It was carried with systems implemented using the common approach to develop `JavaScript`-based systems, named here as Standard, and using the proposed approach, RiPLE-HC. Given that no previous annotative approach has been proposed for `JavaScript`, the experiment investigated the impact of RiPLE-HC feature-oriented way to organize the code in terms of efficiency – regarding *response time* variable –, and correctness of the answers – regarding *precision*, *recall*, and f_1*-score*. Chapter 7 reports details on the planning and execution of VICC1.

VICC2 – the second experiment of the family was planned as a exact replication [52] of Siegmund's pilot [28]. Their pilot evaluated whether separating source code concerns in a feature-oriented fashion would improve program comprehension in terms of *correctness/response time*. Their study compared participants' behavior while inspecting two versions of `MobileMedia` implemented twice with different *variability representations* (*i.e.*, conditional compilation and FeatureHouse). The experiment consists of five *bug-fixing* tasks and compare how students approach the code implemented using the different paradigms to solve the maintenance tasks, *i.e.* they investigated the differences on how the subjects addressed bug fixing tasks. Chapter 8 reports details on the planning and execution of VICC2.

VICC3 – the third experiment of the family extends Siegmund *et al.* [28] design to use a second study object from a different domain: information systems. We developed five new *bug-fixing* tasks equivalent to the original ones used in VICC2. This time, the study compares the subjects' behavior while inspecting the two versions of `MobileMedia` and another two versions of `RiSEEvent` in two different rounds. Chapter 8 also reports details on the planning and execution of VICC3.

VICC4 – the last experimental study of the family. To cope with the importance of addressing other aspects of program comprehension in our experimental studies, we decided to address data-flow and feature precedence comprehension tasks. The tasks were followed by carrying out a focus group [19] with the participants. Chapter 9 reports details on the planning and execution of VICC4.

The aspects that can be measured in program comprehension controlled experiments range from observational [51] to cognitive facets [76]. However, although the measuring from the former facet might be unprecise due confounding parameters [15], the latter is costly and demands large interdisciplinary experience. Therefore, we plan to rely on quantitative and qualitative analysis of different aspects from the observational facet point of view. Thus, the VICC family experimental studies are supposed to improve the amount of evidence regarding the comprehension of FOSD.

6.2 OVERALL PLANNING

In this section, we describe the factors involved and the actions used to control the environment for the experiment. The investigation should follow the guidelines for an experimental setup described by Wohlin *et al.* [18] and the Siegmund's framework for experimentation [11]. Moreover, this empirical study was backed up with the Goal-Question-Metric paradigm proposed by Basili and Rombach [69]. This is used as a mechanism for formalizing the characterization, planning, construction, analysis, learning, and feedback of our experiment.

6.2.1 Research Questions

One major goal of this thesis is to built an evidence corpus on the influence of FOSD on program comprehension. Therefore, each experimental study hold its specific micro goals to contribute to such corpus. In this sense, we detail their specific research questions in their respective chapters (VICC1 – Chapter 7 –, VICC2 and VICC3 – Chapter 8, and VICC4 – Chapter 9).

6.2.2 Target Systems

In the VICC family, we employed four distinct target systems. For each experimental study, there were two equivalent versions of the system, implemented with a different variability representation. Thus, in this section, we present the target systems we used in our studies. All of them, 4 in total, have two comparable versions implemented either with a feature-oriented paradigm or with conditional compilation, which covers the representations detailed, annotative, compositional and hybrid. We present the systems grouped by the two reference languages (`JavaScript` and `Java`).

6.2.2.1 `JavaScript`-based Systems. We selected two `JavaScript`-based systems (*algorithms.js* and *video.js*) and with their respective refactored SPL versions.[1] They were chosen because they represent a familiar domain to subjects and both are small

[1]Both versions of both systems are available at: <https://github.com/riselabs-ufba/RiPLE-HC-ExperimentalData>.

sized. Next, we describe each of them.

`algorithms.js` is a small system that implements a set of classic algorithms and data structures in `JavaScript`. They range from searching (*e.g.*, depth-first search, binary search), ordering (*e.g.*, bubble sort, merge sort, insertion sort), up to math algorithms (*e.g.*, fibonacci, Newton's square root). Figure 6.1 shows the feature model of the version refactored into an SPL. *algorithms.js-SPL* has 28 concrete features and 6 cross-tree constraints.

`video.js` is a small Web video player system that supports HTML5 and Flash video, as well as YouTube and Vimeo formats, through the use of plugins. The tool supports video playback on desktops and mobile devices. The project was started in mid 2010, and it has been used by over 100K websites. Figure 6.2 shows the feature model of the version refactored into an SPL. *(ii) video.js-SPL* has 13 features and no cross-tree constraints.

6.2.2.2 `Java`-based Systems. In addition, we conducted an experimental study including systems implemented in `Java` and with their respective refactored SPL versions using FeatureHouse [1]. This is convenient because the experiment round should run in academic environment and `Java` is still one of the most used programming languages. Besides, FeatureHouse is language-independent, which strengthen eventual conclusions we should draw from the experiments. Next, we introduce the chosen systems that have both versions available, namely `MobileMedia`[2] and `RiSEEvent` [77].[3]

Table 6.1 shows metrics of the packages (as in the FeatureHouse version there are several packages duplications, we decided to add the number of feature-code containers), classes, and lines of code for each version. Next, we briefly describe each of the two systems.

Table 6.1 Target systems characterization.

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

CC: Conditional Compilation; **FH:** FeatureHouse; *****: Number of feature code containers.

[2]Both versions of `MobileMedia` are available at: <http://fosd.net/experiments>.

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

6.2.3 Tasks and Measures

Tasks. While pursuing the evidence on the maintenance effort required by each variability representation, it makes sense to design tasks to assess the impact of such representation in a software evolution context. In our case, we planed the experiment tasks regarding three categories, namely *concept location* – tasks required when the developers are addressing improvement requests, such as the implementation of a new feature or functionality for a given software system –, *bug fixing* – tasks required from the subjects to find and fix incorrect of code leading to deviations of the software system behavior – and *data-flow comprehension* – tasks required from the subjects to explain/demonstrate how the feature relationships are taking place for a given configuration. VICC1 addresses the first category, VICC2 and VICC3 address the second, and VICC4 the third. The tasks of each experimental study are further addressed in the following chapters.

Measures. Figure 6.5 shows the dependent variables of each central topic of the family of experimental studies. There is a set of such variables in each study, which in turn is associated to the measures used to discuss the experimental study findings. The measures themselves are detailed in the respective experimental study chapter.

6.2.4 Support Material

During the different phases of the experimental studies, we used a set of supporting artifacts. Next, we briefly describe the programming environment we used and how the training sessions took place.

6.2.4.1 Programming Environment. To avoid bias from either the familiarity with the Integrated Development Environment (IDE) or the lack of it, we used the `PROPHET`[4] [41] infrastructure in our experiment. It allows planning the experiment with a clean and HTML fashioned user interface in a way that researchers could control the additional tools provided, such as the search (global or local) and the possibility to go back and forth. For the experiment tasks, the only artifacts provided to the students were the feature model of the systems under analysis and the codebase.

6.2.4.2 Training. It is common to run a training session with the subjects, specially when it involves new approaches or tools evaluation. Although feature-oriented is not our proposal or a new approach, we decided to run a training session because the subjects might not know it since the heterogenous nature of the courses we plan to run the experiment sessions. Therefore, we plan to prepare the session as a class in the scope of the experiment. The training session has two phases, the oral presentation of the concepts and details of how the experiment will be carried and a warming up task to help the subjects in the familiarization with the programming environment, the programming language, and the code organization (just in case). In this task, participants should count the occurrence of a feature (CONDITIONAL COMPILATION version) or how often a class is refined (FEATUREHOUSE version).

[4]`PROPHET` is free and open-source available at: <https://github.com/feigensp/Prophet/>

6.3 VARIATIONS IN THE EXPERIMENTAL SETUP

Figure 6.5 shows the variations in the experimental setup of the VICC family. The four dashed rectangles show a summary of each study design, whereas the three central rectangles enumerate their dependent variables. The gray arrow shows the timeline, *i.e.*, the execution ordering of the experiments. Next, we present these variations.

Study type. Three out of the four studies carried were planned and executed as controlled *(quasi-)*experiments (VICC1, VICC2, and VICC3). Only the fourth study was conducted as a focus group (VICC4). We believe replications play an important role in the empirical software engineering research, yet, at the same time, the use of different research methods to address a topic is also important and may be beneficial as it allows to interpret results from different sources [19, 52, 78].

Study design. Each of the experiments had a different design. VICC1 and VICC3 had a crossover design executed in two rounds with two groups, whereas the VICC2 had two groups and was executed in one round only. This happened, because we decided to perform an exact replication of the Siegmund *et al.* [28] pilot first and than extends their design with a second round and observe whether the results differ from one design to another.

Participants background. All studies used a set of students to perform the designed tasks, VICC1 used undergraduate (computer science) students and the other three only graduate (computer science) students. First, we relied on students in all experiments both by convenience and because they perform similarly professionals when they apply a development approach in which they are inexperienced [79]. Second, we justify the shift from undergraduate to graduate students with the complexity that variability introduces in the code. As variability may be an advanced topic, after VICC1, we decided to avoid such bias and rely on graduate students in follow-up studies.

Study tasks. All four studies involved software maintenance tasks or at least a part of it. The VICC1 participants addressed concept location tasks, the VICC2 and VICC3 participants addressed bug-finding tasks, and those in VICC4 addressed both concept location and data-flow comprehension tasks. We are aware of the importance of the investigation of different kinds of maintenance (adaptative, perfective, corrective, and preventive) [80]. However, the nature of controlled experiments require tasks to be as simple as possible to allow participants to finish them in a short time. Besides, as the focus of the thesis is the program comprehension rather than the maintenance tasks itself, we planed tasks with only part of maintenance activities, such as the target concept location and bug-fixing.

Programming Languages. After the first experiment (VICC1), we found hard to recruite a good sample of `JavaScript` developers willing to participate in further experimentation. Therefore, we switched from `JavaScript` to the `Java` programming language, which is another popular language, with open-source systems with equivalent versions implementing variability available, and we managed to recruite a good sample of participants.

One of the main reasons of such variations in their design is the possibility to analyze different dependent variables in each study, which can bring complementary findings regarding the FOSD comprehension.

6.4 CHAPTER SUMMARY

This chapter presented the overall planning as well as the variations of the setup of our family of experimental studies (VICC). The goal of the VICC family is to gather evidence regarding the program comprehension of the feature-oriented software using both RiPLE-HC and FeatureHouse approaches. We presented the domain and basic information of the four target systems used in the VICC experiments. Finally, we detailed the planning, subjects, tasks, support material, and experimental design.

Next chapter presents the execution of the first experiment of the family (VICC1).

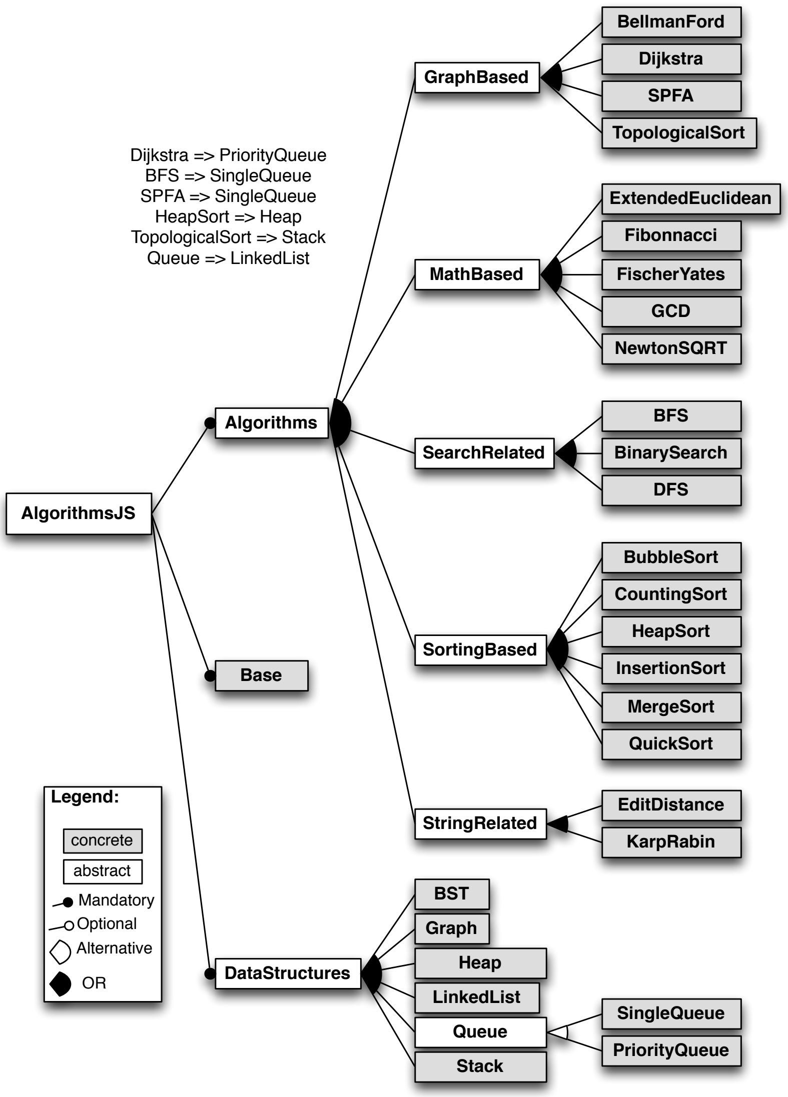


Figure 6.1 `algorithms.js` SPL feature model.

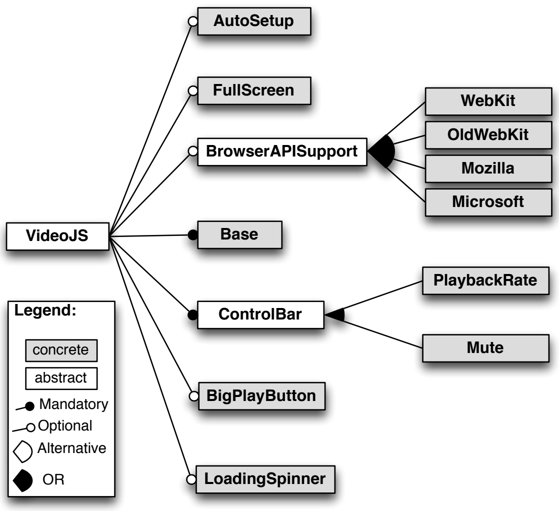


Figure 6.2 `video.js` SPL feature model.

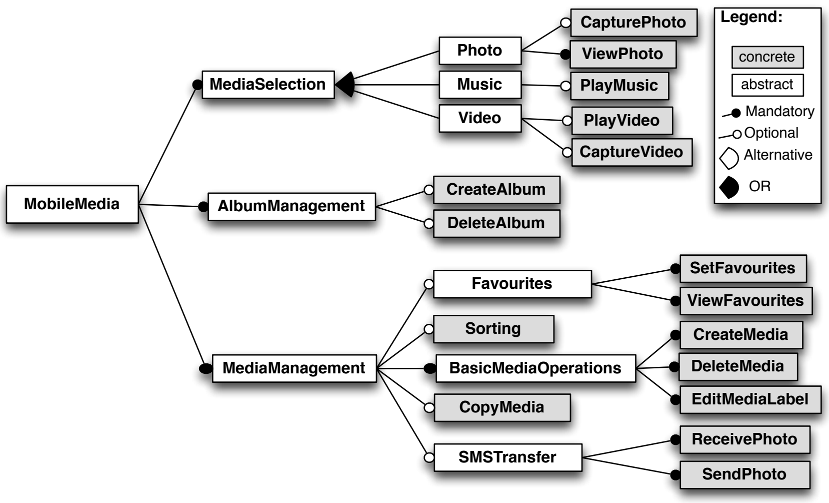


Figure 6.3 `MobileMedia` SPL feature model.

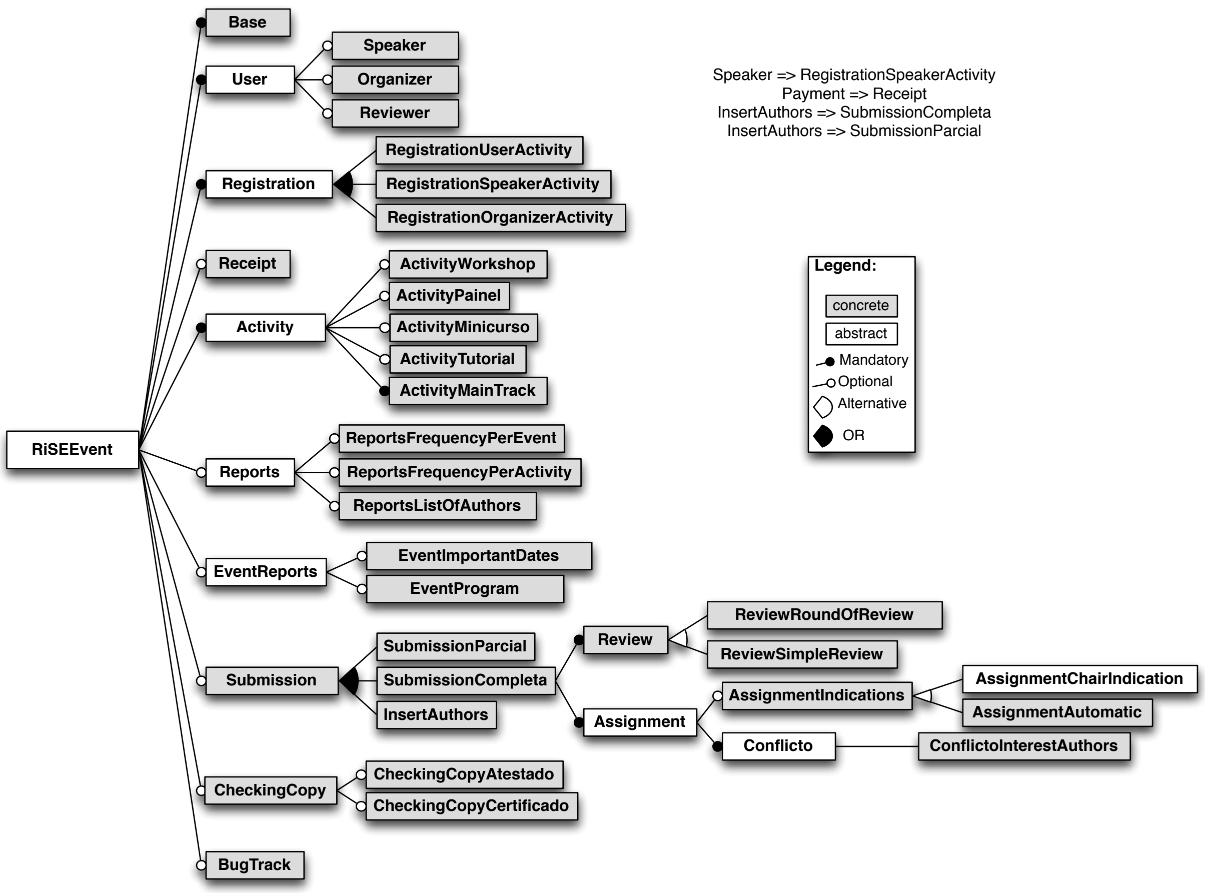


Figure 6.4 `RiSEEvent` SPL feature model.

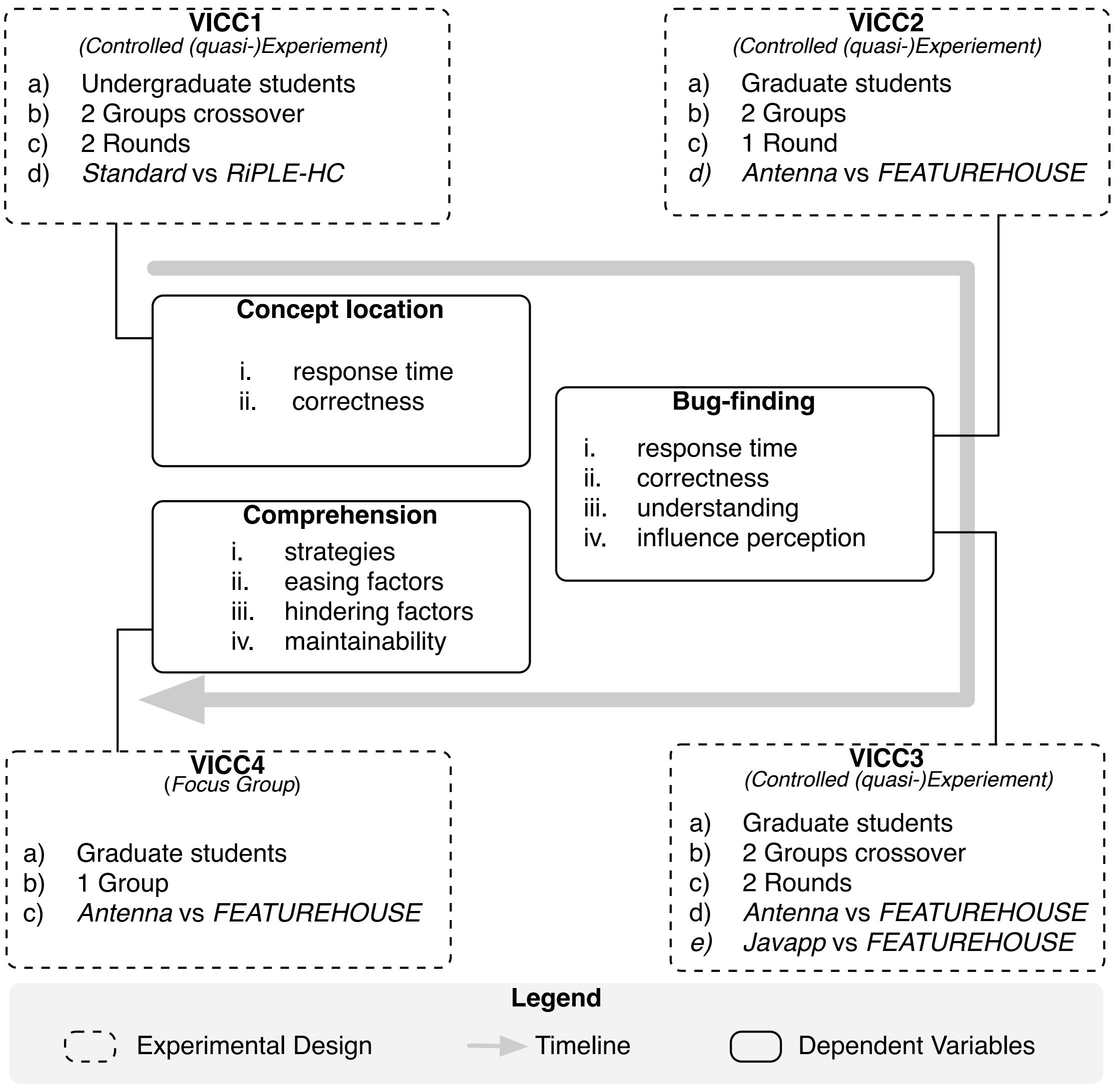


Figure 6.5 Variations in the experimental setup of the VICC family.

Chapter

7

VICC1: ON THE IMPACT ON CONCEPT LOCATION

This chapter reports the first controlled experiment (VICC1). We present each of its phases, as well as we discuss the results of this empirical evaluation. The idea is to gather evidence on the effort demanded by RiPLE-HC from students to accomplish concept location tasks for maintenance purpose. We compare our approach to the current *state-of-the-practice* – which is based on control-flow structures and parametrization (*i.e.*, variables controlling the data flow). A controlled experiment is the cheaper – since we carried it out with Software Engineering students – and yet a reasonable evaluation method to analyze one factor by measuring the effect of two different treatments.

This chapter consists of five main sections. Section 7.1 presents the planning of VICC1, including tasks, metrics, participants, support material, experiment design and variables. Section 7.2 shows how we performed the experiment, which includes the participants characterization and the preparation. Section 7.3 discusses the results of the experiment regarding response time, correctness. Section 7.4 highlights participants' feedback. Section 7.5 presents the threats to validity identified during the evaluation.

7.1 PLANNING

In this section, we present the controlled experiment planning. The factors involved are described as well as the actions used to control the environment for the experiment. This investigation followed the guidelines for an experimental setup described by Wohlin *et al.*[18]. Moreover, this empirical study was backed up with the Goal-Question-Metric paradigm [69]. This is used as a mechanism for formalizing the characterization, planning, construction, analysis, learning and feedback of our experiment. We can define the GQM statement for this experiment:

Analyze *variability representations* for the purpose of *characterization* with respect to their *influence on the concept location effectiveness* from the point of view of the *researcher* in the context of *undergraduate students working with source code.*

More specifically, the goal of this empirical study was to compare the impact of two approaches to organize the source code in feature location from the point of view of novice developers, regarding *response time* and *correctness*: the *ad-hoc* approach, *i.e.*, tacit knowledge of the software engineers, hereinafter referred to as *Standard* – with no systematic way to organize the code – and the RiPLE-HC– with a feature-oriented code organization. Therefore, we pursue the answers for the following research questions:

*RQ*1**:** Does the code organization based on RiPLE-HC approach reduce the *time* required for feature code location in maintenance tasks?

*RQ*2**:** Does the code organization based on RiPLE-HC approach improve the *correctness* of feature code location in maintenance tasks?

Each question embraces a couple of hypotheses. Table 7.1 presents the *null* (H_0) and *alternative* (H_1) hypotheses. In the former, the observation is that RiPLE-HC (R) code organization approach does not affect the time needed (H_{01}) to locate a feature, i.e., *Standard* (S) yields better results. The same rule applies to *f*1*-score* calculations (H_{02}), explained next.

Table 7.1 Hypotheses tested in the controlled experiment.

	Null hypotheses		**Alternative hypotheses**
H_{01}	$\mu(Time_S) \geq \mu(Time_R)$	H_{11}	$\mu(Time_S) < \mu(Time_R)$
H_{02}	$\mu(F1_S) \geq \mu(F1_R)$	H_{12}	$\mu(F1_S) < \mu(F1_R)$

7.1.1 Metrics

To measure the performance of the participants, and to test the hypotheses, we leveraged four metrics: *response time*, *precision*, *recall*, and f_1*-score*. The **response time** relates to the effort spent by the participant to accomplish each task, **precision** relates to correctness and indicates how the student assigned a piece of code to the feature of a given task. The **recall** also relates to correctness and indicates how much from the source code that belongs to a given feature the student managed to find in a given task. Finally, **f$_1$-score** is an harmonic mean of precision and recall and it subsumes the results achieved by the participants regarding the perspectives of both metrics.

Precision and *recall* were obtained by checking the participants answers to an oracle. The oracle was manually built by the author using the code *shadowing* technique [26], which consists of coloring each line of code belonging to a feature with a specific color. The answers were either correct or incorrect whether they match with the shadowed oracle. Despite the hard work on manually shadowing the code, th use of such a technique contributes to improve the reliability of the measurement procedure, as it avoids double judgment in similar cases for different participants. Regarding the f_1*-score*, it depends only on the precision and recall values. The time values were measured in seconds, by using the PROPHET[1] tool [41].

7.1.2 Subjects

Nineteen senior undergraduate students enrolled in a Software Engineering course took part in the experiment. We designed a questionnaire to gather background information regarding their programming experience. Although the target systems were written in JavaScript, we also included questions about programming experience in other languages. The design followed the guidelines from Siegmund *et al.* [41], in which authors observed that programmers holding skills in varying programming languages could yield better results in program comprehension tasks. Appendix B describes the questionnaire.

The answers showed that 32% of the students had previous industry experience. All of them had been enrolled in the university for at least three years. Before joining the experiment, they had taken at least five programming courses. Their programming experience was evaluated with a 5-point likert scale (1 to 5, in which 1 is the lowest value and 5 is the highest one). More than 70% of the participants ranked themselves as 4 or higher experience in C programming; regarding Java programming, over 60% of them reported as being experienced programmers; and a small set of about 33% had previous experience in JavaScript programming.

7.1.3 Tasks

We considered two versions of the open-source JavaScript systems we presented in the Chapter 6: algorithms.js and video.js. They were chosen because they belong to different domains and have different sizes. We designed 21 *static* feature location tasks, *i.e.*, without counting on a running system. Locating feature code for maintenance purposes is a typical task for a developer – which helps developers became aware of the system codebase – and perhaps it is one of the most time-consuming maintenance activities. Although it is not representative of the entire effort a maintenance request demands, it can surely present helpful insights on which direction the RiPLE-HC support should follow, as well as developers who decide for a different approach during their project development.

In the tasks, the participants had to find the code of both, *modular* (when the feature is implemented in a single file or in set of files placed together) and *scattered* (when the code of a single feature is spread over several source files) features. Then, the codebase was organized following both approaches. Figure 7.1 illustrates how each observed approach – namely *Standard* (A) and RiPLE-HC (B) – organizes the code. Next, the participants were asked to find the code implementing a given feature of the target system, and fill in a text field with the names of the files containing the code of the given feature.

Tables 7.2 and 7.3 show some data about the features used in the experiment. For each target system, there is column that identifies the task, the feature addressed, its type, size (LOC), and *scattering degree (SD)*. As *scattering degree*, we consider the number of files containing source code of the feature, which is the same in both used variability

[1]<https://github.com/feigensp/Prophet/>

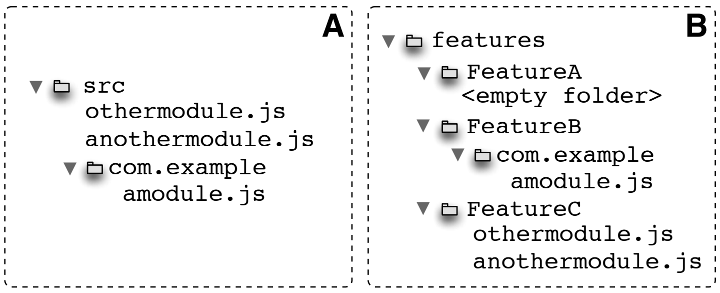


Figure 7.1 Code organization examples.

representations. Features are then defined as either `modular`, if $SD = 1$, or `scattered`, otherwise.

Table 7.2 `algorithms.js` (Round 1) features characterization.

Task	**Feature**	**Type**	**Size** (LOC)	**SD**
Task 1	`KarpRabin`	Modular	57	1
Task 2	`BellmanFord`	Modular	43	1
Task 3	`PriorityQueue`	Scattered	34	2
Task 4	`Fibonacci`	Modular	28	1
Task 5	`BinarySearch`	Modular	13	1
Task 6	`Dijkstra`	Modular	33	1
Task 7	`Heap`	Scattered	73	3
Task 8	`InsertionSort`	Modular	16	1
Task 9	`MergeSort`	Modular	24	1
Task 10	`Stack`	Scattered	13	2
Task 11	`CountingSort`	Modular	35	1

LOC: Lines of Code; **SD:** Scattering Degree.

7.1.4 Support Material

In order to avoid bias from the familiarity with the IDE environment, we used the `PROPHET`[2] [40] infrastructure in our experiment. It allows planning the experiment in a way that researchers can control the additional tools provided, such as the search (global or local) and the possibility to go back and forth with a clean and HTML fashioned user interface. For the experiment tasks, the artifacts provided to the students were the system feature model and their codebase.

[2]`PROPHET` is free and open-source and it is available at: <https://github.com/feigensp/Prophet/>

Table 7.3 `video.js` (Round 2) features characterization.

Task	**Feature**	**Type**	**Size** (LOC)	**SD**
Task 1	`AutoSetup`	Scattered	35	2
Task 2	`FullScreen`	Scattered	173	7
Task 3	`PlaybackRate`	Scattered	88	3
Task 4	`Mute`	Scattered	59	5
Task 5	`WebKit`	Modular	11	1
Task 6	`OldWebKit`	Modular	11	1
Task 7	`Mozilla`	Scattered	19	2
Task 8	`Microsoft`	Modular	10	1
Task 9	`BigPlayButton`	Scattered	13	3
Task 10	`LoadingSpinner`	Scattered	23	3

LOC: Lines of Code; **SD:** Scattering Degree.

7.1.5 Experiment Design and Variables

Figure 7.2 illustrates the experiment design, which consisted of "one factor (code organization) with two treatments (*Standard* and RiPLE-HC)". The experiment was composed of two rounds, in which all participants could use each tool. In **round #1 (R1)**, the *Group A* (n=11) addressed the system using *Standard*, and the *Group B* (n=8) with RiPLE-HC. In **round #2 (R2)**, the groups were then exchanged, so that the *Group A* addressed the system with RiPLE-HC and the *Group B* the other way round. From the planned 21 tasks, 11 were addressed in **R1** and 10 in **R2**. Each task involved only one feature of the target system. Tables 7.2 and 7.3 show each of them. Hereinafter, ***round #1*** and ***round #2*** will be referred to as R1 and R2, respectively.

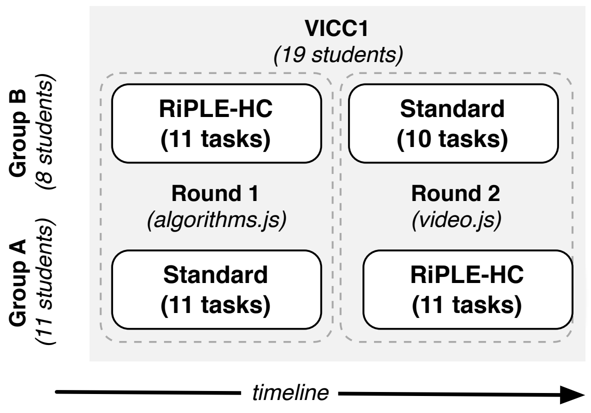


Figure 7.2 Experiment design.

Three groups of variables were considered in this experiment: *independent*, *dependent*, and *confounding variables*. The first one comprised the approaches used in this study, namely ***Standard*** and **RiPLE-HC**. The second group considered the ***time*** – as a measure of effort – and the ***correctness*** – as a measure of effectiveness. The latter encompassed different variables that may affect the task analysis, as follows: ***the level of modularity, each round, the target systems,*** and ***the size of each system***. By *level of modularity* we mean the nature of the feature (modular or scattered); *the rounds* stand for the order in which a participant addressed the system with a given approach; *target system* stands for the familiarity of the participants with them (it might be the case that participants are familiar with `algorithms.js` but not with `video.js`); and, finally, the *size of the system* stands for the extent to which the differences in the target systems influenced the analysis of the source code.

7.2 PREPARATION AND EXECUTION

This Section describes the participants characterization and the preparation for the experiment execution. This phase in this experiment comprised the planning and the execution of a pilot study. The pilot study consisted of the execution of the experiment with two participants not involved in the actual execution to identify bottlenecks in both preparation and execution of the study. Both participants answered the characterization form, none of them had previous knowledge of neither the RiPLE-HC approach nor `JavaScript` programming; and the feedback form, they reported positively for the time, size, and content of the tasks. We identified some opportunities for improving the study, thus we made some adjustments in the experiment materials, such as reducing the amount of tasks and updating the training presentation.

In the experiment session itself, the training session took about 60 minutes. It consisted of establishing a common vocabulary, explanations on the environment where they had to report the results, and the forms the participants had to fill out. Next, both rounds were performed. *R1* took about 50 minutes and *R2* took about 30 minutes. The confounding variables, such as the "maturing effect" and "fatigue" may explain the observed difference in execution time between the rounds. After the execution session, the participants reported the answers and fill out a feedback form (Appendix C).

7.3 RESULTS AND DISCUSSION

In this section, we present and discuss raw data, and the impact of both approaches on the dependent variables *time* and *correctness*. While the former produces evidence on the cost of maintenance tasks, the latter produces evidence on whether developers may or may not benefit from using RiPLE-HC over *Standard*.

We started the analysis by applying the *Shapiro-Wilk* test to verify the normality of each sample, namely *Time* and F_1_score in both, *R1* and *R2*. Table 7.4 shows the results of the test considering each round and treatment. Results pointed out normality in the samples generated in *R1*, concerning to time values for both treatments (RiPLE-HC and *Standard*). Besides, we carried out a data transformation on the values from *R2*, by applying a logarithmic function to adjust the statistical differences found. Then,

Table 7.4 Shapiro-Wilk normality test data.

Round 1				**Round 2**			
Time		***F_1-score***		***Time***		***F_1-score***	
RHC	*STD*	*RHC*	*STD*	*RHC*	*STD*	*RHC*	*STD*
before log transformation							
Yes	Yes	No	Yes	No	No	Yes	No
after log transformation							
Yes	Yes	No	No	No	No	No	No

RHC: RiPLE-HC; **STD:** *Standard*

(a) *Round 1* (b) *Round 2*

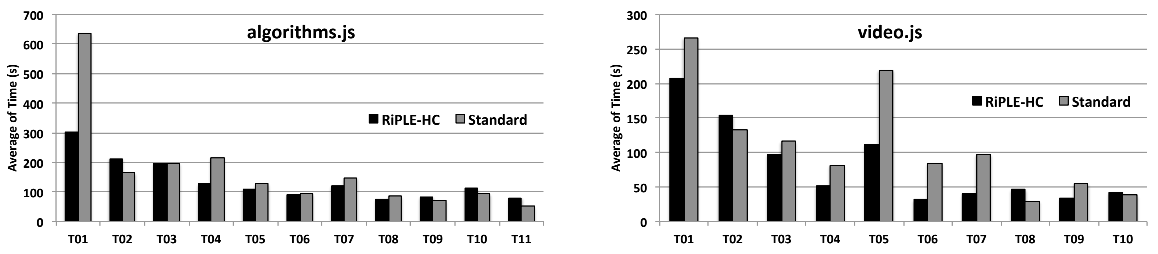


Figure 7.3 Average time spent in each task in the VICC1.

we carried out the hypothesis testing, by applying the *independent T-Test* to assess *Time* (*R1*), and the non-parametric *Mann-Whitney U* test was used for *Time* (*R2*) and F_1_score (in both *R1* and *R2*).

7.3.1 Execution Time

Figure 7.3 shows the average time spent in each task, in seconds. We may observe similar results between tasks carried out by the groups A and B. In both, the earlier tasks demanded more time to produce the results. The lack of familiarity with the tools may explain those values. As the participants gained confidence on the source code, the time spent decreased. Indeed, the similarity in time spent refutes the arguments in favor of the harmfulness of the code scattering. To a certain extent, the scattered code produced using RiPLE-HC approach did not demand extra effort.

Although both target systems are small, there is a significant size difference among them. However, such a difference does not affect the effort to locate features. Participants spent less time analyzing the second system – Figure 7.3(b), than the preceding one – Figure 7.3(a). The lower values in *R2* can be a result of the likely maturation effect, given that participants were already familiar with the activity.

Most participants spent less time on average to perform feature location when the target system was organized with the RiPLE-HC. RQ1 is primarily interested in an-

Table 7.5 Mann-Whitney U Test of hypothesis for *Time* spent.

	Round 1		**Round 2**	
Approach	**Mean Rank**	***p*-value**	**Mean Rank**	***p*-value**
RiPLE-HC	137.48	.539*	9.30	.36
Standard	171.05	.542**	11.70	

*: Equal variances assumed; **: Equal variances not assumed.

Table 7.6 Mann-Whitney U Test results for F_1*_score*.

	Round 1		**Round 2**	
Approach	**Mean Rank**	***p*-value**	**Mean Rank**	***p*-value**
RiPLE-HC	13.27	.20	10.80	.82
Standard	9.73		10.20	

alyzing whether the RiPLE-HC approach reduces the time needed to locate features. The hypothesis test was performed in both rounds by considering the average time spent by the participants in each task. Table 7.5 shows that the participants that used the RiPLE-HC spent less time to perform their tasks. The significant difference on the mean values is due to the mentioned data transformation. However, with a *p*-value higher than .05, it is impossible to refute the null hypothesis (H_{01}) in any rounds.

7.3.2 Correctness

The results indicate that both approaches produced similar impact on feature location for modular and scattered features. In most cases in *R1*, the F_1*_score* of both approaches were higher than 50%. (Figure 7.4(a)). However, both approaches had worse results in *R2* (Figure 7.4(b)). Although the RiPLE-HC does not excel `Standard` results in tasks T01, T03, and T04, the results were good in all the other tasks. Subjects inspecting source code organized with the RiPLE-HC yielded slightly better results when compared to the *Standard* approach. In fact, in *R1*, while the median of RiPLE-HC was around 0.8, in the *Standard* approach was around 0.6. In *R2*, the difference was around 20%. We believe that participants might have been affected by the novelty on how the code is organized in the RiPLE-HC prior to the training section of the experiment. Such an impact might explain the perceived reductions in gains concerning to the source code organization. Besides, we could not indentify any errors pattern in the participants errors. Compositional and annotative approaches yielded good and bad results depending on the task addressed by participants. In addition, we did not carry any analysis regarding only those who had correct answers due the size of the remaining sample and to avoid fishing for results during the analysis.

Regarding the analysis of correctness, as RQ2 stands out, the hypothesis testing considered the average of the F_1*_score* of the participants in every task. Table 7.6 shows the

(a) *Round 1* (b) *Round 2*

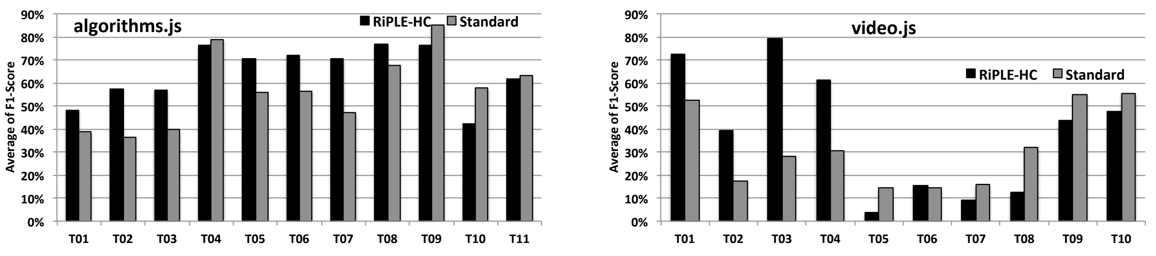


Figure 7.4 Average f_1-score in each task in the VICC1.

observed results. The participants who used the RiPLE-HC approach got better results in both rounds. However, we see that p-value is greater that .05 in both rounds, thus, we cannot refute null hypothesis (H_{02}) in any of them.

7.4 PARTICIPANTS' FEEDBACK

By the end of the experiment, the participants were asked to fill out a feedback form, only 13 finished such a request. They were supposed to describe their point of view regarding the likely benefits and drawbacks of each approach. About 67% of the participants reported that the RiPLE-HC approach fits best for `JavaScript` development and 33% still preferred the *Standard* approach. Next, we provide a detailed description on the participants' reported standpoints.

Standard: The benefits reported by the students about this approach was that it is similar to the way they learned how to program. It may facilitate the approach learning, if compared to the experimental one, and give freedom to developers to program as they are used to. On the other hand, some participants pointed out some drawbacks. For instance, 38% (5 out of 13) of them reported it was difficult to locate features, which reportedly demands a good searching tool. In addition, they mentioned that using *Standard* makes it harder to carry out a feature driven approach and might increase the project complexity.

***RiPLE-HC*:** The participants (53% – 7 out of 13) found out *RiPLE-HC* to be simpler to use than *Standard*. They also reported that associating features and folders might improve feature location, and that a strong point in favor is that the RiPLE-HC approach exercises the so-called hybrid composition and annotation. This characteristic could improve the way `JavaScript` developers program to different platforms. However, some participants believe that the RiPLE-HC is messy and that the novelty on the way of developing features may find resistance to be adopted in practice.

7.5 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of this empirical study. We believe that presenting such detailed information may contribute to further research and replications of this study [18], which may be built upon the results presented herein. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion* validity.

7.5.1 External Validity

We identified some threats that may limit the ability to generalize the results. For example, the study was carried out in an *in-vitro* setting, which means a sample selected pseudo-randomly. In addition, most of the participants were characterized as inexperienced with industry projects, which poses a threat to the study. We attempted to mitigate such a threat by characterizing and reporting the environmental settings of the experiment, since it is unfeasible to reproduce a realistic environment.

7.5.2 Internal Validity

There are possible threats that may happen without the researcher's knowledge affecting individuals from different perspectives, such as *(i)* the maturation and learning effects, *(ii)* the testing repetition since several tasks were carried out, and *(iii)* the experiment instrumentation. These threats were mitigated by choosing different features for each task, as well as by randomizing the sequence of task's execution to omit possible relationships. Finally, the only artifacts used were the source code which participants were already familiar with, and the feature models and the `PROPHET` tool, explained in the training session.

7.5.3 Construct Validity

Confounding constructs may affect the findings. For instance, the presence or the absence of knowledge about a particular programming language may not explain the causes of failures in the feature location tasks. In fact, the differences may depend on the participants' experience, which was controlled with the characterization form, to ensure that participants had substantial experience to accomplish the tasks. Besides, the manual construction of the oracle by shadowing and analysis of the answers might also have affected the results.

7.5.4 Conclusion Validity

We observed from the results a likely *low statistical power*, which concerns to the power of used tests to reveal a true pattern in the data. Employing well-known measures mitigated such a threat. Another observed threat is the *fishing* for a specific result, which we mitigated by relying the analysis only on the gathered data. There is a threat on the *reliability of treatment implementation*, when participants are treated differently, which was minimized by avoiding communication with the participants and leaving time for discussion of the experiment between the training and the experiment sessions. Finally, the *random heterogeneity of participants*, which was measured by the characterization form and presented, but no additional actions were taken to control it since the experiment took place in the context of an academic course.

7.6 CHAPTER SUMMARY

This chapter presented the first controlled experiment of the VICC family, which addressed the impact of the new constructs of RiPLE-HC in the feature code location tasks in `JavaScript`-based systems. We detailed the planning and execution of the VICC1, while discussing metrics, participants, tasks, support material, experiment design, and variables. We also highlighted the experiment execution and presented a detailed discussion on the results of the experiment.

In all the four scenarios defined for the experiment (Rounds 1 and 2 regarding both hypotheses: *time* to complete the task – 2 scenarios – and *correctness* of the answers – 2 scenarios), the mean of the results of the subjects indicated slightly better result in favor of our approach. However, the data points did not allow us to statistically refute the null hypotheses. Therefore, it is not possible to generalize that developers addressing feature location tasks in current *state-of-the-practice* code organization take longer, neither that they make more errors than those addressing the code structured with the proposed approach. In addition, the study produced evidence on the benefits of systematic code organization. However, the feedback of the participants suggested points to further investigation *(i)* the RiPLE-HC provides better code organization regarding the systems functionalities; *(ii)* the composition can ease the product development for different platforms; and *(iii)* that their unfamiliarity with the approach may have hindered better results.

Finally, we enumerated a number of threats to validity identified during the different phases of the experiment. Next chapter presents the planning, execution, results of the VICC2 and VICC3.

Chapter 8

VICC2 AND VICC3: ON THE INFLUENCE ON BUG-FIXING

This chapter reports the second and the third controlled experiments (VICC2 and VICC3). We present each of its phases, as well as we discuss the results of this empirical evaluation. We extend the evidence corpus of the influence of the use of different variability representations on program comprehension by comparing CONDITIONAL COMPILATION against FEATUREHOUSE. This time, the experiments rely on bug-fixing tasks, another common activity of the SDLC.

The chapter consists of five main sections. Section 8.1 describes the planning, preparation, and execution of the (*quasi-*)experiments, including the addressed research questions and hypotheses. Section 8.2 presents the results of the controlled experiments and Section 8.3 discusses the results of the confounding parameters survey with the participants. Section 8.4 discusses the influence of participants' motivation and difficulty perception concerning their results in our replications comparing the current to the original study, as well as the answers to our research questions. Finally, Section 8.5 discusses threats to validity of this study.

8.1 STUDY SETTINGS

In this section, we present the VICC2 and VICC3 studies, which were performed as replications [52] and also followed Wohlin *et al.*guidelines [18]. In summary, both studies can defined with the following GQM statement:

Analyze *variability representations* for the purpose of *characterization* with respect to their *influence on the bug-fixing effectiveness* from the point of view of the *researcher* in the context of *graduate students working with source code.*

We proceed by presenting research questions, hypotheses, planning, and execution of them.

In these studies, we used systems written in `Java`, considering the following variability representations: FeatureHouse and Conditional Compilation annotations. We choose them, because FeatureHouse is a language-independent approach [1] and Conditional Compilation is likely the most widely used variability mechanisms [6]. In addition, FeatureHouse is representative of the group of techniques that *physically* separate the implemented features, whereas Conditional Compilation is representative of the group of approaches that *virtually* separate the implemented features.

8.1.1 Research Questions, Hypotheses, and Variables

In this section, we present the 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 each research question, the associated hypotheses, and how we proceeded to gather data to test them.

RQ1: What are the differences in the developers' effectiveness while performing program-comprehension tasks addressing software systems implemented with FeatureHouse and Conditional Compilation?

The main claim in favor of variability representations, such as FeatureHouse is its supposed benefits regarding the improved modularity [10]. Therefore, in the context of maintenance tasks, feature-oriented systems written using FeatureHouse might facilitate the *understanding* of problems triggering software bad functioning. This allow us to state the following hypothesis:

$\mathbf{H_1}$ *It is easier to understand* FeatureHouse *code rather than* Conditional Compilation *code.*

For hypothesis $\mathbf{H_1}$, we asked the participants to answer "why the problem was occurring" and "how to fix such an issue" 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 coded the answers in a three-point scale. *Complete understanding* (2) when participants answered both questions correctly, *partial understanding* (1) when either the problem or the solution was described correctly, and *no understanding* (0) 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 functionalities of a software system. Besides, Conditional Compilation has proved its value as a suitable strategy to implement variability in software systems [6], there is evidence regarding optional features that FOP adheres more closely to the Open-Closed principle than Conditional Compilation [61, 81]. However, we can say little about the likeliness of FOP yields more *correct* answers in comparison to Conditional Compilation annotations. We capture this rationale in the following hypothesis:

> **H_2** *Developers addressing change requests using* FEATUREHOUSE *provide more correct answers compared to developers using* CONDITIONAL COMPILATION.

For hypothesis **H_2**, we asked the participants in each task to describe the “class”, the “line of code”, and, in case of a FEATUREHOUSE task, also the “feature folder” where the error occurs. Again, we coded the answers in a three-point scale. *Complete correctness* (2) when the participants correctly answered both questions, *partial correctness* (1) when a participant only described the class (and the feature folder in case of a FEATUREHOUSE task) correctly, and *no correctness* (0) 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 on the program comprehension while addressing an issue during a maintenance task could lead teams to undesired costs. This fact makes the *response time* a factor worth to investigate. 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 led us to state the next hypothesis:

> **H_3** *Developers addressing change requests with* FEATUREHOUSE *code can finish their tasks faster than those with* CONDITIONAL COMPILATION *code.*

For hypothesis **H_3**, the PROPHET tool [82] recorded the time spent by the participant in each task in milliseconds. We then converted the time measure to minutes to ease the analysis.

Next, we present our second research question. The rationale we present serves a twofold purpose: *(i)* the better understanding of the results of the experiments and the whole process of program comprehension; and *(ii)* to aid in the design of future comprehension-focused experiments.

RQ2: What are the differences between software systems implemented with FEATUREHOUSE and CONDITIONAL COMPILATION regarding developers’ perception of the influence of confounding parameters while performing comprehension tasks?

There are numerous confounding parameters when performing controlled experiments either with students or professionals [15]. In fact, these parameters hinder the complete understanding of how practitioners actually comprehend the source code. Therefore, we believe that the understanding of individual perception of such confounding parameters may help researchers to better design follow-up research studies addressing program comprehension. We wrap up this in our last hypothesis:

> **H_4** *Developers addressing change requests with* FEATUREHOUSE *code perceive the influence of confounding parameter differently than when they use* CONDITIONAL COMPILATION.

Table 8.1 Measures, their descriptions, and the associated hypothesis.

Measure	**Description**	**Hypotheses**
	Independent variable	
`variability representation`	The type of variability representations used by a participant in a given task.	$\mathbf{H_1}$-$\mathbf{H_4}$
	Dependent variables: Tasks measures (RQ1)	
`understanding`	Describes to what extent a participant understood a given task.	$\mathbf{H_1}$
`correctness`	Describes to what extent a participant answered a given task correctly.	$\mathbf{H_2}$
`response time`	Describes how long it took to finish a given task.	$\mathbf{H_3}$
	Dependent variables: Feedback measures (RQ2)	
`influence rating`	Perceived severity of the influence of a confounding parameter in a task by the participants.	$\mathbf{H_4}$

Finally, for hypothesis $\mathbf{H_4}$, we conducted a survey with the participants after they finished the experiment session. In the survey, they rated in a 4-point Likert-scale [83] ranging from "no influence" (1) to "high influence" (4). We used the ratings to identify the differences regarding the perceived influence of such aspect.

In summary, each hypothesis concerns a main aspect: ($\mathbf{H_1}$) understanding; ($\mathbf{H_2}$) correctness; ($\mathbf{H_3}$) response time; and ($\mathbf{H_4}$) developers' perception of confounding parameters. The variability representations used in the target systems are our two independent variables. Thus, we are going to refer to the group of developers addressing the CONDITIONAL COMPILATION code as "Group CC" and to the group addressing the FEATUREHOUSE as "Group FH". Table 8.1 summarizes the measures, their descriptions, as well as their association with the addressed hypotheses.

8.1.2 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.

8.1.2.1 Design. We carried out the experiments in the form of two replications, consisting of three rounds. All rounds were inspired by the pilot carried by Siegmund et al. [28]. The participants were all Computer Science graduate students from the Federal University of Bahia's (UFBA). Figure 8.1 shows the design of the replications. In fact, the *first* replication was executed in one round (Round 1) as an *exact replication* [52] of Siegmund *et al.*'s pilot. The *second* replication is a *conceptual replication* [52], executed in two rounds (Round 2 and 3). Round 2 is exactly the same as Round 1; in Round 3, we designed the tasks trying to reproduce the same level of difficulty of the Siegmund's pilot in a second system, which was executed one week after Round 2.

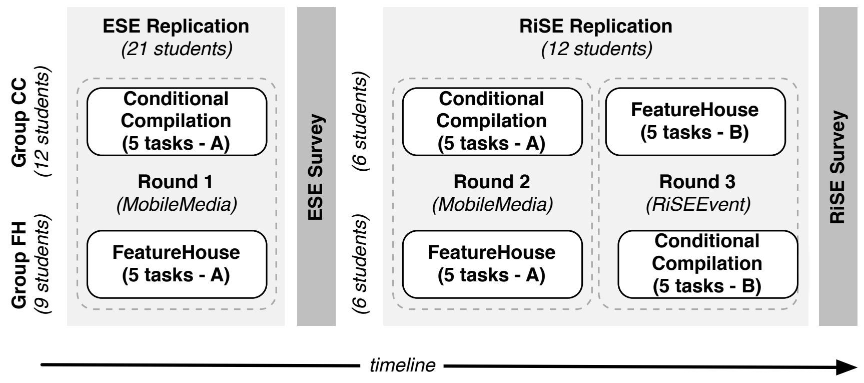


Figure 8.1 Experiment design.

In total, 33 students took part 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[1] (RiSE) research group to participate in Rounds 2 and 3. Each participant worked at an individual workstation and they were not allowed to communicate among themselves.

In both replications (ESE and RiSE), we arranged the participants in two groups addressing a different variability representation, which we are going to call throughout this chapter the CC group and the FH group. We created homogeneous groups according to their programming experience by using balancing. In the ESE replication, the grouping is unbalanced by three students in the group FH due to 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 we did in ESE replication.

8.1.2.2 Target Systems. We used `MobileMedia` and `RiSEEvent`. Both systems were described in Chapter 6.

8.1.2.3 Tasks. We reused the Siegmund *et al.* [28] pilot study tasks in the first and second rounds of our replications, because we believe they are well balanced and had a feasible difficulty level for one experiment session. Each round consisted of *five* bug-finding tasks. We introduced five bugs of equivalent difficulty level in the second target system, relying on the task design of the original study. It is worth notice that all annotations of the systems are disciplined [27, 84].

Table 8.2 describes the tasks defined for each target system used in the experiment. In addition, Figure 8.2 shows a code snippet from the `RiSEEvent` target system from which the Task 1 was created. The error was introduced in the Line 17 by using the variable `statement` instead of `idActivity` in the delete SQL query. The bugs were chosen randomly. Table 8.3 shows the example of a correct answer.

[1]<www.rise.com.br>

Table 8.2 Experiment tasks defined for each target system used in the experiment.

	`MobileMedia`	`RiSEEvent`
Task 1	Instead of setting the counter to the actual value, it is set to 0.	The code uses the wrong activity ID variable in the delete query.
Task 2	A false identifier is used (`SHOWPHOTO` instead of `PLAYVIDEO`).	The code calls "gerarCarne(...)" instead of "gerarBoleto(...)".
Task 3	Instead of showing a list of favorite items when requested by clicking in the "View Favorites" menu, the application shows nothing.	The notification was not implemented yet.
Task 4	Instead of show a list of pictures ordered by the number of visualizations, they appear unordered.	The Register menu was added a second time instead of adding the Reports menu.
Task 5	A wrong label for deleting a picture is used, such that the check for the user rights provided by the access control feature is never executed and a user can delete a picture without according rights.	The option "JFrame.DO_NOTHING_ON_CLOSE" was used instead of "JFrame.EXIT_ON_CLOSE".

8.1.2.4 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 target system. All these items were provided through the PROPHET[2] experimental workbench [82]. We extended the tool by translating the complete environment to Brazilian Portuguese to carry out the experiments in the participants mother language.

Figure 8.3 shows the two screens of the PROPHET workbench. Figure 8.3(a) shows in the left the tasks description window, which guided a participant throughout the experiment displaying the tasks descriptions, the appropriate place to hold the participant's answers, including those regarding the feedback form. In the other side, Figure 8.3(b) shows the source-code inspector, which the participants used to search through 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 in right tabbed panel (④).

8.1.3 Preparation and Execution

Next, we present details on the preparation and the execution of the experiments. For the purpose of preparation, we conducted training sessions. For the purpose of execution,

[2]PROPHET is free and open-source and available at <https://github.com/feigensp/Prophet/>

```java
package rise.splcc.repository;

// <several imports>

public class ActivityRepositoryBDR
            implements ActivityRepository {
//  <70 more lines>
    @Override
    public void remove(int idActivity)
            throws ActivityNotFoundException,
            RepositoryException {
        try{
            Statement statement =
                        (Statement) pm.getCommunicationChannel();
            int i = statement.executeUpdate(
                        "DELETE FROM Activity "+
                        "WHERE idActivity = '"+ idActivity+"'");
//  <3 more lines>
        } catch(PersistenceMechanismException e){
//  <8 more lines>
}
//  <300 more lines>
}
```

Figure 8.2 Original `RiSEEvent` code snippet used in the Task 1.

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.

8.1.3.1 Training. We carried out different training sessions to cope with the different knowledge of the sample in each replication, ESE and RiSE. In the first replication, we presented a seminar about variability with practical examples, including differences and peculiarities about each of the variability representations, FEATUREHOUSE and CONDITIONAL COMPILATION. As for the second, since the participants were already familiar with the concept of variability, we developed and provided the participants with written material for training prior to the actual experiment session for the purpose of familiarization with the experiments object of study.

In both training sessions, we included references to real software projects (other than those used either in the training sessions) using each technique, such as the Linux kernel for the CONDITIONAL COMPILATION and other projects available at the SPL2Go repository (<http://spl2go.cs.ovgu.de/>) for FEATUREHOUSE. Apart from the references and the recommendation to inspect any of the referenced projects, no activity was required prior to the warm-up task.

Table 8.3 `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 the variable idActivity instead of statement.

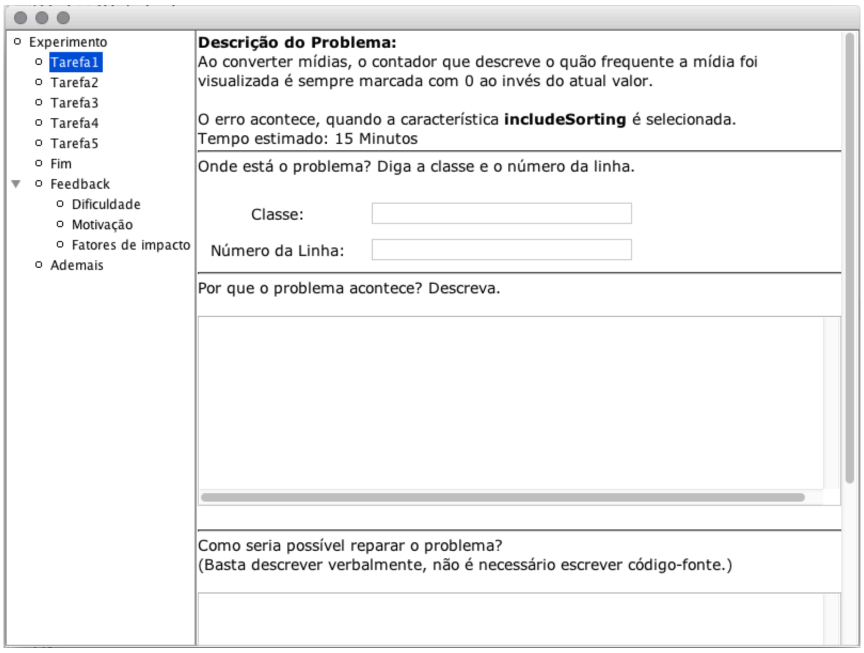
*: the actual line number.

8.1.3.2 Participants Characterization. Appendix B shows the questionnaire used to measure the programming-experience in both replications, except a couple of specific questions regarding participants' knowledge of FeatureHouse and Conditional Compilation to the RiSE participants. This different treatment is justified because the RiSE group of participants have a solid knowledge about variability, which might include the addressed variability representations, which in turn would influence in the balance of the groups distributions.

The questionnaire serves the purpose of gathering basic information and details on the programming experience of each participant. More specifically, we asked about their industrial experience, as well as their background knowledge in different programming paradigms and languages. Table 8.4 summarizes the characterization of the participants. Half of them were female in both replications. The age medians were 33 in the ESE replication and 29 in the RiSE. 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. In a 5-point Likert scale, 16 of the ESE group and all the RiSE group have at least mediocre level (3 out of 5 points) experience with the object-orientation paradigm. 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 working experience.

We asked the participants to compare themselves against their classmates and professional developers with 20 years of experience. Figure 8.4 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 8.4(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 8.4(b) shows a similar pattern in the RiSE replication, except that no one judged him/herself as less experienced as 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 heterogenous groups with sufficient experience for the replications.

(a) Tasks description window



Descrição do Problema:
Ao converter mídias, o contador que descreve o quão frequente a mídia foi visualizada é sempre marcada com 0 ao invés do atual valor.

O erro acontece, quando a característica **includeSorting** é selecionada.
Tempo estimado: 15 Minutos

Onde está o problema? Diga a classe e o número da linha.

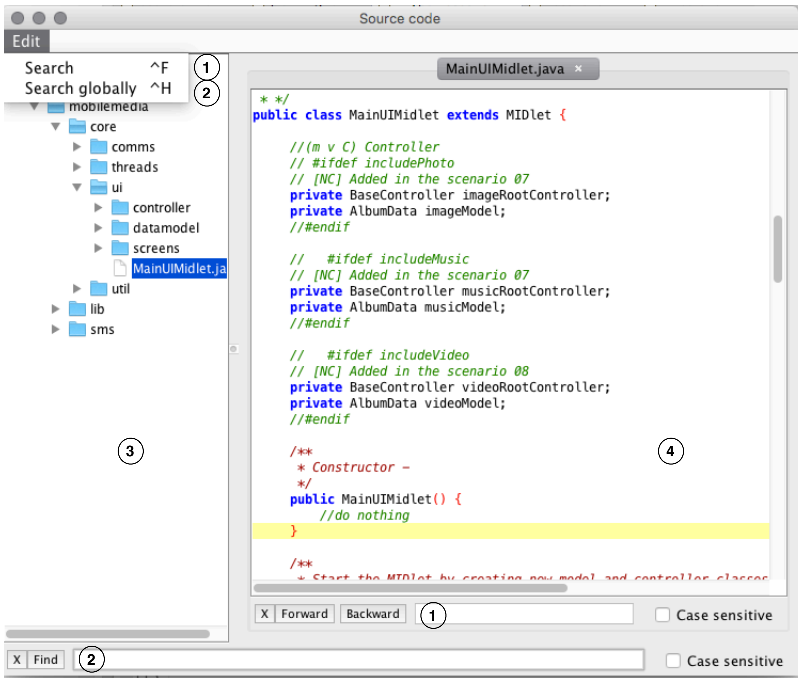
Classe:

Número da Linha:

Por que o problema acontece? Descreva.

Como seria possível reparar o problema?
(Basta descrever verbalmente, não é necessário escrever código-fonte.)

(b) Source code inspector



```
 * */
public class MainUIMidlet extends MIDlet {

    //(m v C) Controller
    // #ifdef includePhoto
    // [NC] Added in the scenario 07
    private BaseController imageRootController;
    private AlbumData imageModel;
    //#endif

    //   #ifdef includeMusic
    // [NC] Added in the scenario 07
    private BaseController musicRootController;
    private AlbumData musicModel;
    //#endif

    //   #ifdef includeVideo
    // [NC] Added in the scenario 08
    private BaseController videoRootController;
    private AlbumData videoModel;
    //#endif

    /**
     * Constructor -
     */
    public MainUIMidlet() {
        //do nothing
    }

    /**
     * Start the MIDlet by creating new model and controller classes
```

Figure 8.3 PROPHET workbench: the two screens used by the participants to solve the assigned tasks.

8.1.3.3 Example Task.

All participants completed a warm-up task just before the actual experiment session in each round with two main purposes: *(i)* for the familiarization with the experiment environment (PROPHET Workbench); *(ii)* a first contact with

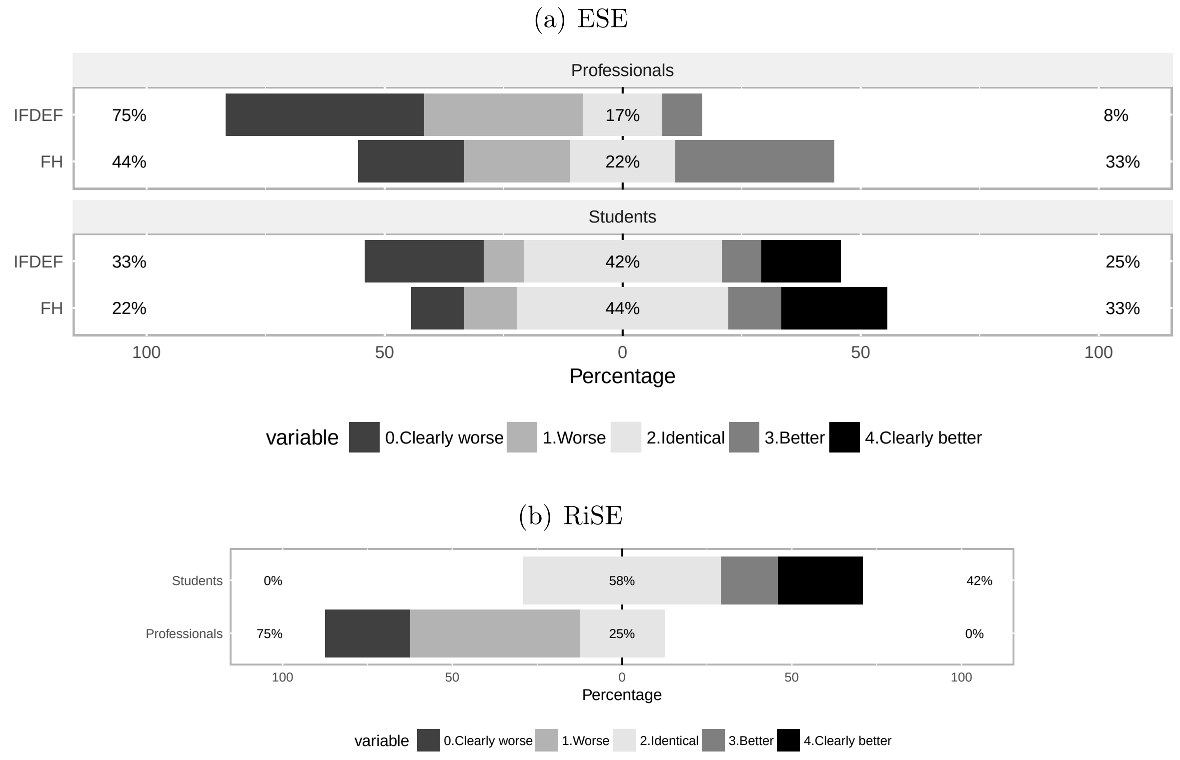


Figure 8.4 Participants' programming experience (self assessment) against their classmates and professional developers with 20 years of experience.

the source-code of the target system. The task required the participants (FH group) to identify in how many features a class ("PhotoViewScreen") had been refined; or (CC group) to identify in how many files a feature ("includeFavourites") had been implemented to.

8.2 RESULTS RQ1

RQ1: What are the differences in the developers' effectiveness while performing program-comprehension tasks addressing software systems implemented with FEATUREHOUSE and CONDITIONAL COMPILATION?

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

Table 8.4 Participants' experience summary.

Characteristic	**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

H_1 ***It is easier to understand*** **FeatureHouse** ***code rather than*** **Conditional Compilation** ***code.***

To complete this task, participants should describe why a problem occurs and how this problem could be fixed. Figure 8.5 shows raw data for the three rounds of replication. The scale ranges from *(0)* no understanding; *(1)* partial understanding; *(2)* complete understanding. In all three rounds, most participants failed at locating the problem and proposing a solution.

However, in some tasks of the RiSE replication rounds, half of the group reached a complete understanding. If we compare Rounds 1 and 2, in which groups with different background performed the same tasks, we can say the CC group from the RiSE replication had slightly better results, whereas the results are inconclusive for the FH groups. On the other hand, by 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 looks slightly worse than Round 2.

We tested $\mathbf{H_1}$ regarding `understanding`. We then used the *Shapiro-Wilk Test* to discover whether the raw data sample had a non-normal distribution. Besides, since we have one independent variable (`variability representation`) with two levels ("FH" and "CC") and one ordinal dependent variable (`understanding`), we used the *Mann-Whitney U Test* to test the $\mathbf{H_1}$. To test $\mathbf{H_1}$, we considered all observations (*i.e.*, from the three rounds) regarding `understanding` from each group ("FH" and "CC") as our sample. The resulting p-value from the test was 0.9947 with 95% of confidence, which indicates both groups yielded identical populations.

$\mathbf{H_1}$: Rejected. There was no statistical difference in the understanding and in the task of describing the bugs and eventual solutions between the populations using either FeatureHouse or Conditional Compilation code.

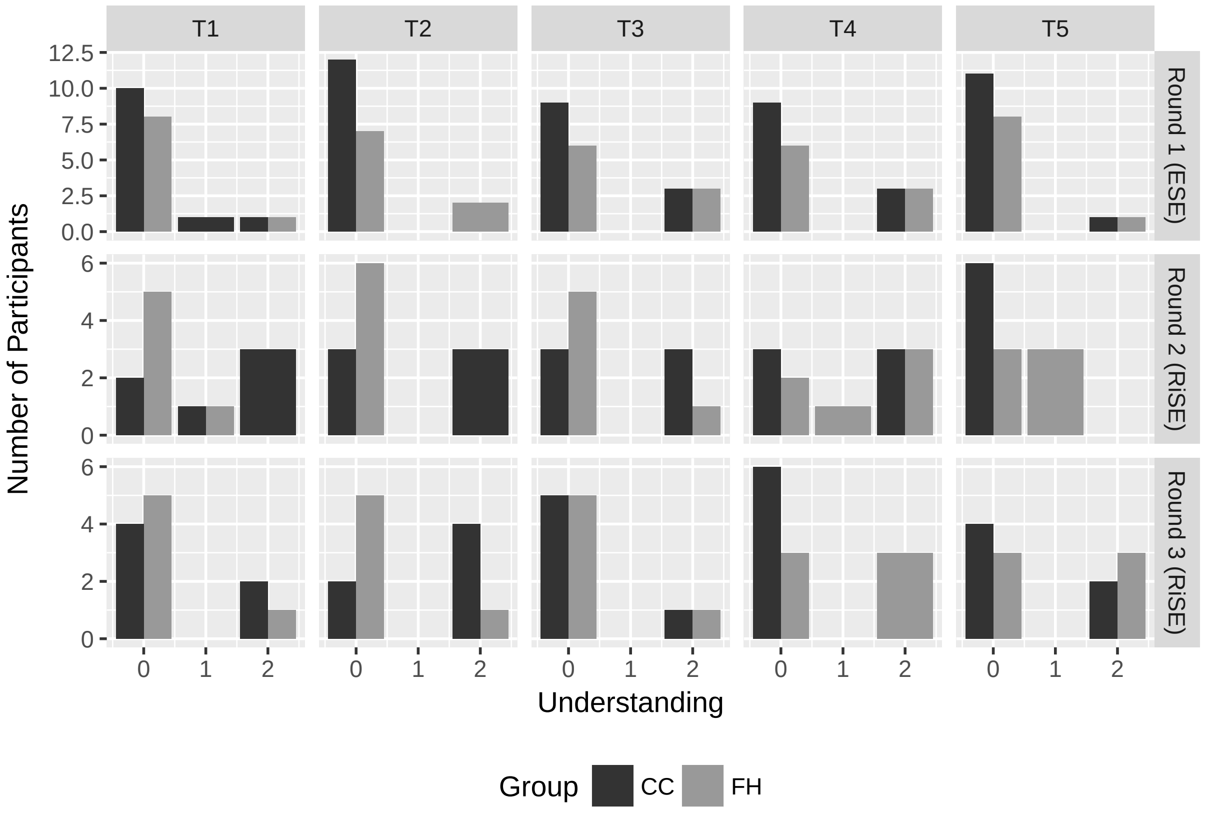


Figure 8.5 Overall participants understanding of "why the problem happened?" and "how to solve" in each round of the experiments. Scale: *(0)* no understanding; *(1)* partial understanding; *(2)* complete understanding.

H_2 *Developers addressing change requests using* FeatureHouse *provide more correct answers compared to developers using* Conditional Compilation*.*

To complete this task, participants should describe what class, line of code, and for those in the FH group the feature folder where they had located the error. Figure 8.6 shows the raw results for 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 partially locate at least the class in which the problem occurs.

If we compare Rounds 1 and 2, we can state 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 $\mathbf{H_2}$ with regard to `correctness`. We then used the *Shapiro-Wilk Test* to discover whether the raw data sample had 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

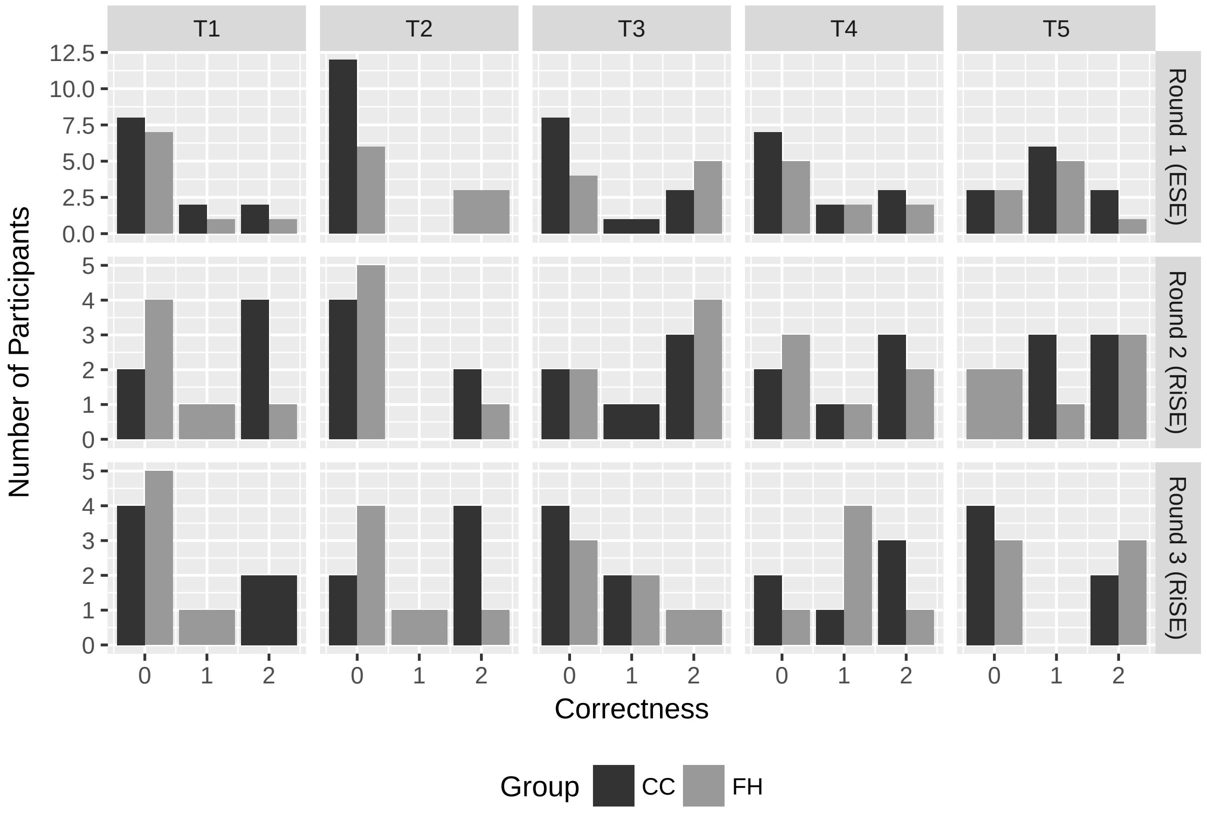


Figure 8.6 Overall participants correctness of the description of what "class", "line of code", and "feature folder" in each round. Scale: *(0)* no correctness; *(1)* partial correctness; *(2)* complete correctness.

the *Mann-Whitney U Test* to test the $\mathbf{H_2}$. To test $\mathbf{H_2}$, we also considered all observations (*i.e.*, from the three rounds) regarding `understanding` from each group ("FH" and "CC") as our sample. The resulting p-value from the test was 0.7579 with 95% of confidence, which indicates both groups yielded identical populations.

$\mathbf{H_2}$: Rejected. There was no statistical difference in the locating bugs between the populations using either FeatureHouse or Conditional Compilation code.

H_3 ***Developers addressing change requests with*** **FeatureHouse** ***code can finish their tasks faster than those with*** **Conditional Compilation** ***code.***

Figure 8.7 shows the raw results for three rounds regarding the response time of the participants. It can be seen that all tasks yielded similar time ranges for both variability representations, regardless of the experience with variability of the group (ESE and RiSE) or the addressed target system. The response times ranged (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 following tasks were performed in a smaller amount of time.

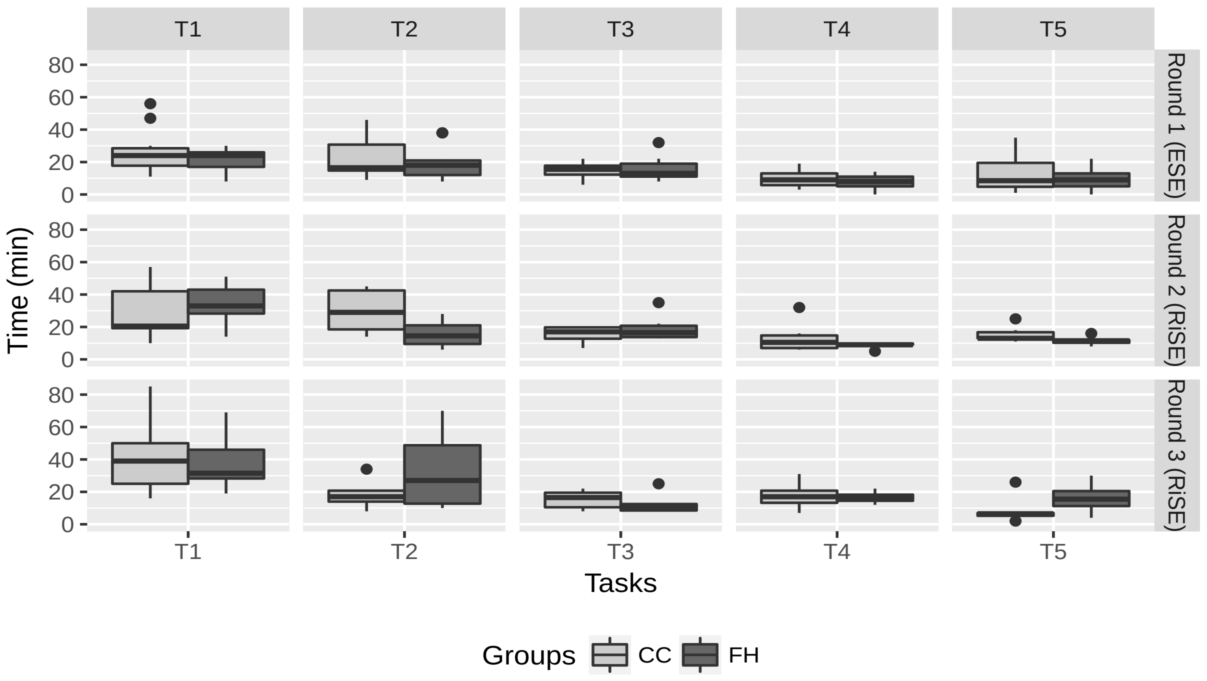


Figure 8.7 Overall response time of the participants in each round.

Finally, we also tested $\mathbf{H_3}$ with regard to the `response time`. We then used the *Shapiro-Wilk Test* to discover whether the raw data sample had a non-normal distribution. Besides, since 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 *Mann-Whitney U Test* to test the $\mathbf{H_3}$. To test $\mathbf{H_3}$, 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 resultant from the test was 0.6011 with 95% of confidence, which indicates both groups yielded identical populations.

$\mathbf{H_3}$: Rejected. There was no statistical difference in the response time of the populations using either FEATUREHOUSE and CONDITIONAL COMPILATION code.

8.3 RESULTS RQ2:

RQ2: What are the differences between software systems implemented with FEATUREHOUSE and CONDITIONAL COMPILATION regarding developers' perception of the influence of confounding parameters while performing comprehension tasks?

In this section, we discuss developers' perception of the influence of confounding parameters on the activities during the experiments tasks regarding each of the variability representations.

8.3.1 Confounding Parameters Classification

We start by presenting the confounding parameters considered in the survey and its classification. According to Siegmund *et al.* [15], they can be separated into three groups regarding individuals: *(i)* background; *(ii)* knowledge; and *(iii)* circumstances. The first group concerns "color blindness", "gender", "culture", and "intelligence", which we believe are inappropriate to our analysis. However, we took into consideration the other two groups and also added study-specific confounding parameters. Table 8.5 shows the confounding parameters considered from each one, which we detail next.

Table 8.5 Confounding parameters took into consideration for the comprehension analysis.

Type	**Confounding parameters**
Individual knowledge	Ability, Domain knowledge, Education, Familiarity with study object, Familiarity with tools, Programming experience, and Reading time.
Individual circumstances	Fatigue, Motivation, and Treatment preference.
Study specific	Programming language (`Java`) knowledge, Code organization, Experience in the programming paradigm.

8.3.1.1 Individual knowledge parameters.

This first group describes parameters that are influenced by learning and experience [15]. We are aware of some of these characteristics may concern different aspects of the comprehension process. However, it is important to measure such general parameters to explore their relationship with the more specific ones.

Ability (**CP01**) in our context can be understood as a participant's skill level, in which he/she relied on to accomplish an assigned task.

Domain knowledge (**CP02**) in our context can be understood as a participant's knowledge about mobile phone's application (`MobileMedia` Tasks) or scientific event management systems (`RiSEEvent` Tasks).

Education (**CP03**) describes what they learned in their studies. Participants may have taken different kinds of courses and subjects prior to their participation in the study, which could have influenced their performance.

Familiarity with study object (**CP04**) refers to the target software systems, *i.e.*, their experience with `MobileMedia` or `RiSEEvent`. Those with few or no experience in software maintenance are expected to report differently from experienced ones.

Familiarity with tools (**CP05**) refers to how experienced participants are with the tools used in the evaluation. In this particular case, we explore how well the participants are familiarized with the PROPHET environment, as well as its capabilities.

Programming experience (**CP06**) refers to the previous experiences the participants had so far with writing and understanding source code . In our study, we asked them about their experience in the background questionnaire applied prior to the experiments.

Reading time (**CP07**) refers to how fast the participants can read.

8.3.1.2 Individual circumstances parameters.

This second group describes how participants feel during the time of the experiment [15].

Fatigue (**CP08**) refers to how participants get tired and lose concentration. This parameter was also measured in the feedback questionnaires. In our case, the only measure taken was the participants subjective judgment of how this could influence in their performance.

Motivation (**CP09**) refers to how motivated the participants were during the execution of each task. This parameter was also measured in the feedback questionnaires.

Treatment preference (**CP10**) refers to the preference of the participants for any of the used techniques or particularly in this study for any of the programming paradigms. This parameter was also measured in the feedback questionnaires.

8.3.1.3 Study-specific parameters.

This last group describes those factors particular to this study, which are mainly associated to the variability representation techniques.

Programming language (`Java`) knowledge (**CP11**) refers specifically with their knowledge of the language of the target systems. We asked about their knowledge in the background questionnaire.

Code organization (**CP12**) refers to the organization of the target system's source code. More specifically, the systems version using conditional compilation has the usual object-oriented organization and the other one using feature orientation was organized in containment hierarchies [10]. This parameter was also measured in the feedback questionnaire.

Experience in the variability representation (**CP13**) refers to the experience either in feature-orientation with CONDITIONAL COMPILATION or FEATUREHOUSE. This parameter was also measured in the background questionnaire.

8.3.2 Participants' Perception of Confounding Parameters

In this section, we present the results of the survey with the participants. To the best of our knowledge, this the first exploratory study on the influence of confounding parameters in the feature-oriented software comprehension. We inspired ourselves by the Santos and Mendonça work [85] – who analyzed decision drivers of guiding god class detection – to conduct such an analysis of confounding parameters based on participants' perception. We used the survey method to gather participants' feeling about the influence of each confounding parameters in the accomplishment of their experiments tasks. We carried out the surveys differently in each replication of the experiment.

In the ESE replication, we asked the participants to order four confounding parameters – namely, familiarity with tools (**CP05**), Programming knowledge (**CP11**), Code organization (**CP12**), and their Programming experience (**CP13**) regarding their influence on the complete `understanding` of why the error occurred and how it could be solved ($\mathbf{H_1}$), as well as the `correctness` of their answers of where the error was located

($\mathbf{H_2}$). In the RiSE replication, we asked the participants to judge the influence of each of the confounding parameters described in the previous section with a four-point Likert-scale [83]: *(i)* No influence; *(ii)* Little influence; *(iii)* Moderate influence; and *(iv)* High influence. Next, we proceed with the analysis of the gathered data.

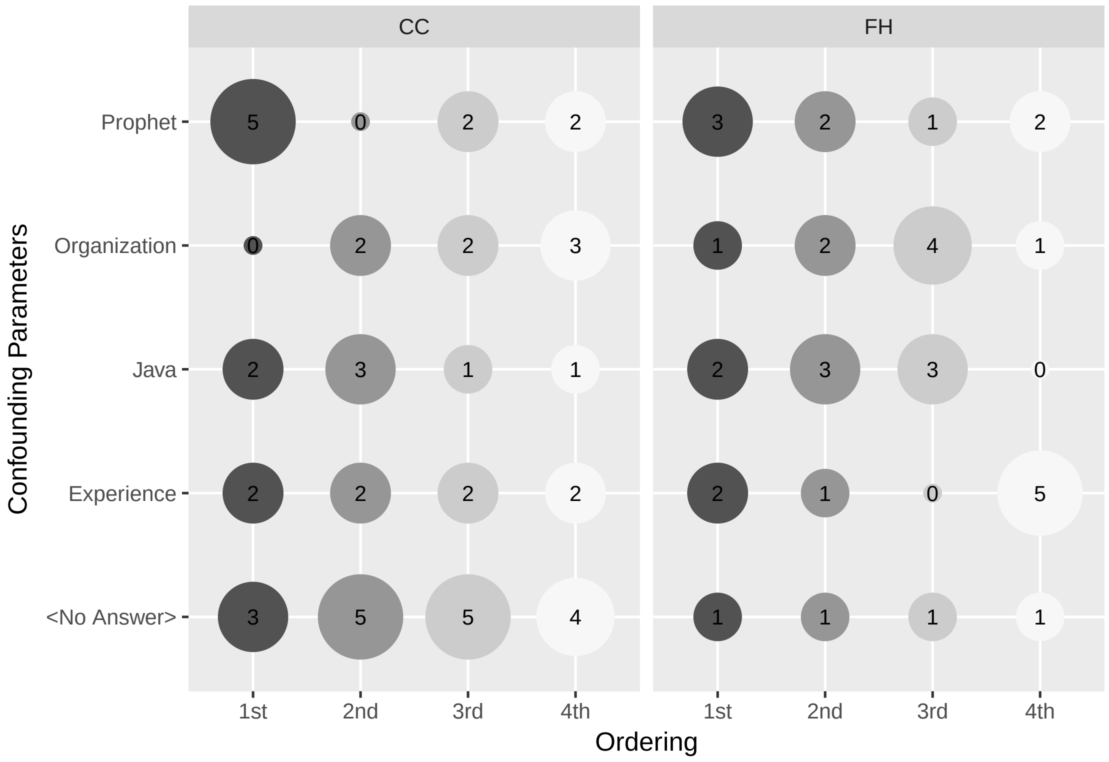


Figure 8.8 Overall confounding parameters in each task of the ESE experiment.

Figure 8.8 shows a bubble plot from the ordering provided by the 21 participants of the ESE execution. We split the plot by the variability representation group ratings. Each participant could mention a parameter only once and should, ideally, answer with a *4*-tuple. Unfortunately, some participants did not answer the feedback properly, which forced us to include fifth line in the plot ("<No Answer>") to represent the missing ratings. The columns are ordered by the participants influence ratings ($1st$ had higher influence and $4th$ had lower influence). While the bubbles' size indicates how many times they cited the confounding parameter, the darkest their coloring is, the higher was the influence parameter in the activity.

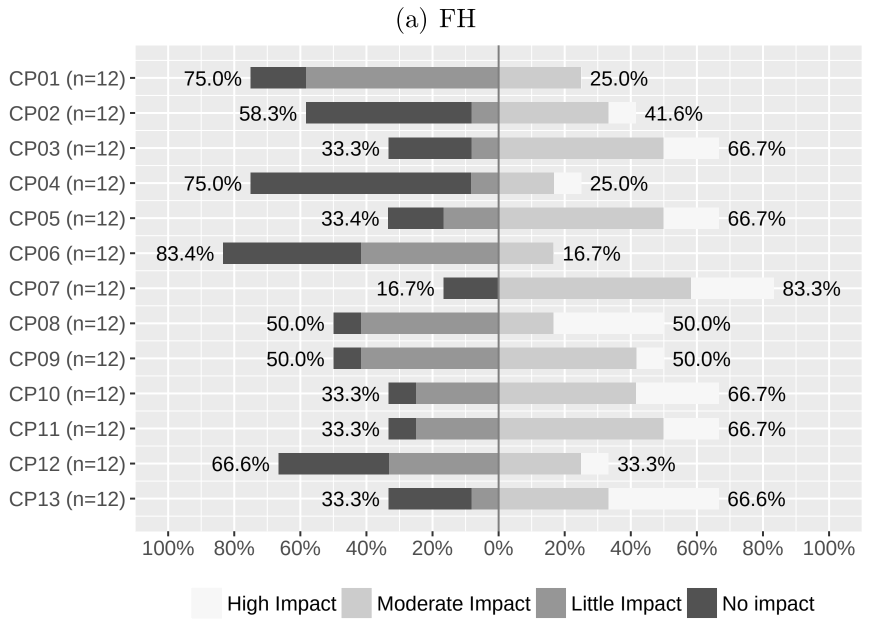
In both groups (FH and CC), the lack of knowledge or the lack of functionalities in the PROPHET workbench played higher influence on their activities, which eventually might have hindered their success. In addition, the "Code Organization" is the less cited as the greater influence, which corroborates with the results of the hypotheses $\mathbf{H_1}$, $\mathbf{H_2}$, and $\mathbf{H_3}$. Both, their "knowledge of `Java`" and their "Programming Experience" were cited the same amount of times for either variability representation. On the other hand, the "Programming Experience" was the most cited by the FH group of participants as the one with less influence, whereas in the CC group the "Code Organization" assumed this place.

We then proceeded with the feedback collection differently in the RiSE replication. Table 8.6 shows the questionnaire used to collect the rating feedback of programming experience. Instead of ordering the confounding parameters, we elaborated more by using a structured survey questionnaire to collect more complete information from the participants.

Table 8.6 Questionnaire used for measuring the influence of the confounding parameters on the comprehension tasks of the experiments.

	Influence			
Confounding Parameter	**No**	**Little**	**Moderate**	**High**
(CP01) Ability	❑	❑	❑	❑
(CP02) Domain knowledge	❑	❑	❑	❑
(CP03) Education	❑	❑	❑	❑
(CP04) Familiarity with study object	❑	❑	❑	❑
(CP05) Familiarity with tools	❑	❑	❑	❑
(CP06) Programming experience	❑	❑	❑	❑
(CP07) Reading time	❑	❑	❑	❑
(CP08) Fatigue	❑	❑	❑	❑
(CP09) Motivation	❑	❑	❑	❑
(CP10) Treatment preference	❑	❑	❑	❑
(CP11) Programming language (Java) knowledge	❑	❑	❑	❑
(CP12) Code organization	❑	❑	❑	❑
(CP13) Experience in the programming paradigm	❑	❑	❑	❑

Figure 8.9 shows the plots of the ratings by the 12 participants of both groups – 6 using FH in Round 2 and 6 using FH in Round 3, similarly for CC– in the RiSE replication. It is worth mentioning that each participant completed the feedback questionnaire twice (in Rounds 2 and 3), with each one concerning a different variability representation, which explains the $n = 12$ for each confounding parameter. The darkest colors represent "No" or "Little influence" in the comprehension tasks – the bars in the left-hand side of the axis –, whereas the lighter colors "Moderate" or "High influence" – the bars in the right-hand side of the axis. 50% or more of the participants rated the confounding parameters *Familiarity with the study object* (**CP05**), *Reading Time* (**CP07**), *Fatigue* (**CP08**), *Treatment preference* (**CP10**), *Programming language knowledge* (**CP11**), and *Experience in the programming paradigm* (**CP13**) of moderate to high influence while using either of the variability representation. In fact, most participants rated only the formal *Education* (**CP03**) with the opposite level of influence between the two groups. In other words, while most of the FH group participants rated **CP03** with moderate to high influence in the comprehension, most of the CC group participants rated it having low to no influence in their tasks.



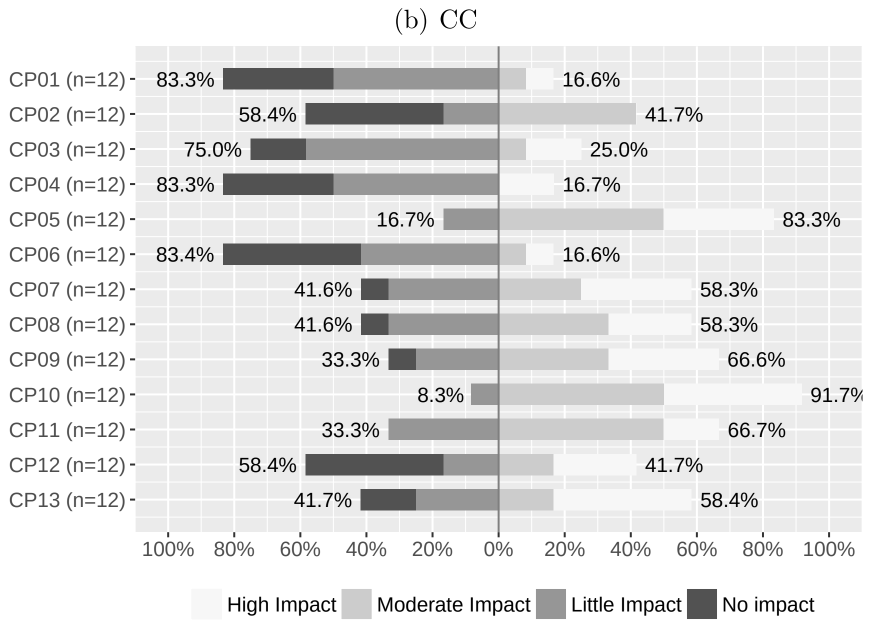


Figure 8.9 Participant's perception of the confounding parameters during the tasks execution.

In addition, some participants reported the fact they are used to have proper debugging tools to performing such bug-finding tasks. In fact, in Round 1, the "Familiarity with the tools" was reported in the feedback survey to have the highest influence on their comprehension tasks regardless of the group. The FH group members mentioned "Experience" and "Code organization" as having little influence in the comprehension, whereas the CC participants had a heterogeneous rating regarding the remaining parameters. In Rounds 2 and 3, "Familiarity with the tools", "Treatment preference", and "Programming language knowledge" were the most cited as having moderate to high influence in the comprehension and consequently in the successfully accomplishment of the assigned tasks.

We conducted a more in-depth analysis of the relationship among the confounding parameters by calculating the correlation (cor) and (dis)similarity ($d = 1 - cor$) among the addressed confounding parameters. We choose dendograms [86] – a tree-like hierarchical clustering plot – to show the similarity and dissimilarity among the parameters. In our case, the (dis)similarity says how different/similar the parameters are regarding the influence ratings of the participants.

Figure 8.10 shows the dendograms built from the feedback of participants when using each of the variability representation addressed in this study. Each leaf is more similar to those leaves located in its same branch, whereas the distance among branches shows the dissimilarity among groups of leaves. For instance, in Figure 8.10(a), which shows the FH feedback, the left-most branch shows CP13, CP10, and CP11, which indicates that they are more similar among each other than to those in the right side of the branch. Yet, at the same time, CP10 and CP11 are more similar among each other than to CP13. In a lowest level of granularity, when we consider the participants using FeatureHouse, the pairs (CP10, CP11), (CP05, CP09), (CP03, CP08), (CP04, CP12), and (CP01, CP06) are the more similar, whereas when we consider the participants using Conditional Compilation, the pairs (CP05, CP08), (CP07, CP12), (CP09, CP10), (CP03, CP11), and (CP01, CP04) are the more similar.

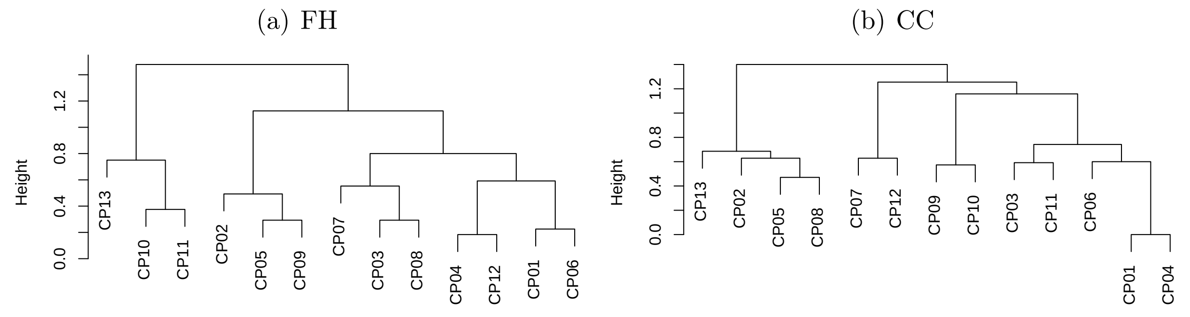


Figure 8.10 Dendograms showing the hierarchical clustering of the confounding parameters regarding their (dis)similarity.

Figure 8.11 shows the correlation matrix of how the participants perceive the influence of the confounding parameters in the comprehension tasks while using the different variability representations. This matrix complements the analysis of the dendograms. In fact, the ordering of the matrix follows the similarity order presented by them. The darker coloring of the bubbles in the plot shows the participants' feeling was more similar among them while using FeatureHouse.

For the FH group, there are several pairs of parameters with moderate correlations with values above 0.6, such as (CP10, CP11), (CP05, CP09), (CP07, CP03), and (CP03, CP08). Only (CP04, CP12) had a strong correlation. In contrast, for the CC group' ratings, there are several weak correlations, with the exception of the strong correlation between the pair (CP01, CP04). For both groups, there were no strong negative corre-

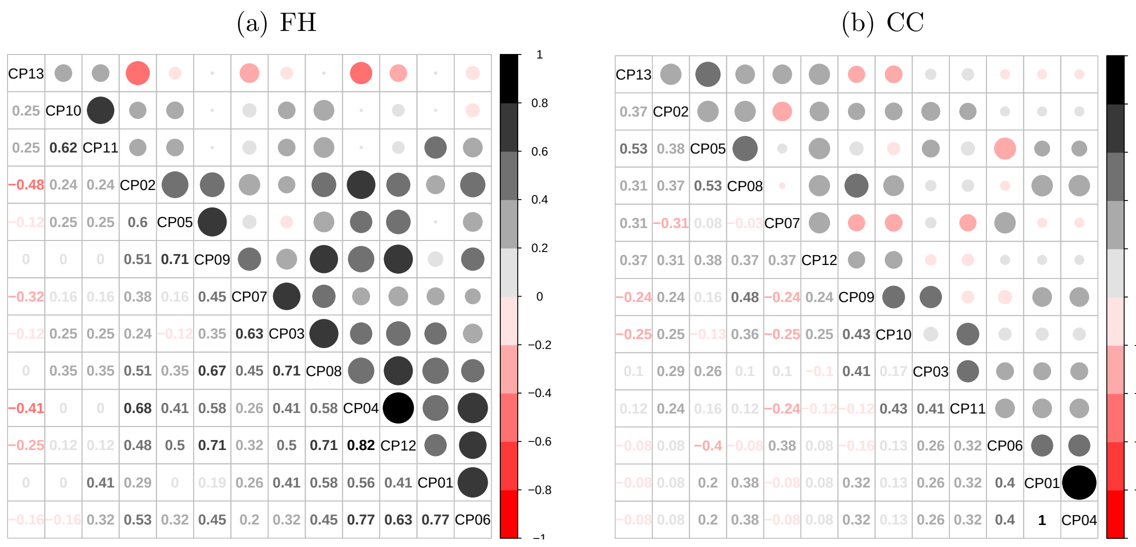


Figure 8.11 Correlations matrices of the participant's perception of the confounding parameters.

lations. In summary, FH and CC groups differ in the strength of positive correlations, which it indicates evidence of the different perception of the confounding parameters by the participants.

H_4: Accepted. There was evidence of different perception of confounding parameters by participants while engaging maintenance tasks either in FeatureHouse or Conditional Compilation code.

8.4 DISCUSSION

First, in this section, we present the participants' perception of their motivation and tasks' difficulty during our study and compare whether they are replicate results from Siegmund et al. pilot [28]. Then, we present how the replications' results correlate to the participants' motivation and the tasks difficulty. Later, 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 tasks' difficulty.

8.4.1 On the Participants' Motivation, Tasks' 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 are related since harder tasks can easily decrease or increase participants' motivation. We use a five-points Likert scale [83] 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 8.12 shows the participants' feedback regarding difficulty and Figure 8.13 shows their feedback regarding their motivation. In addition, we are going to use quotes from the original study [28] to guide our discussion.

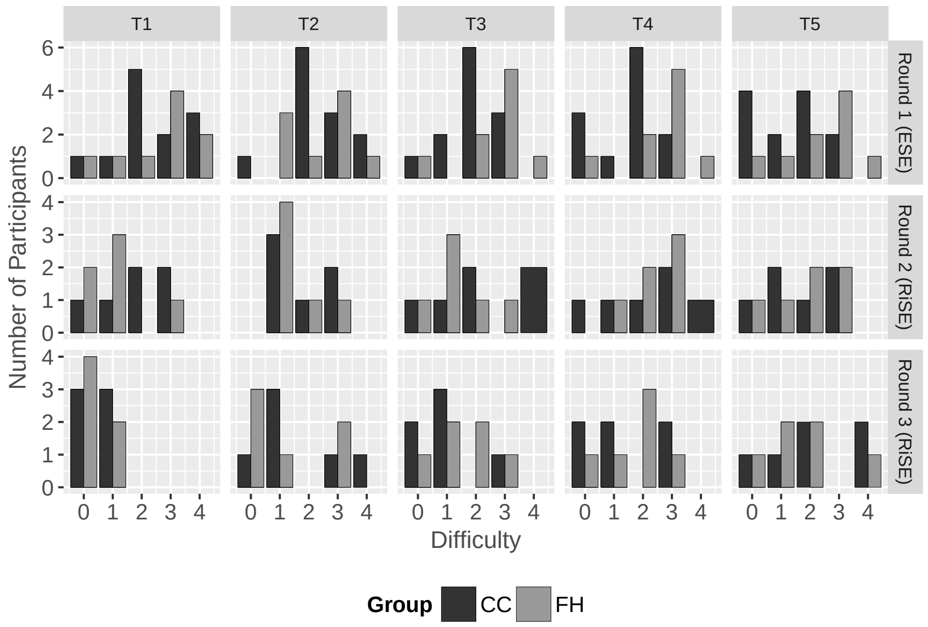


Figure 8.12 Overall feeling of difficulty of the participants in each round. Scale: *(0)* Very difficult; *(1)* Difficult; *(2)* Normal difficulty; *(3)* Easy; and *(4)* Very easy.

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

Siegmund et al.[28] "

In Round 1, both groups of participants' responses of the ESE replication had similar tendency. Both groups responses concentrated the rating of the tasks difficulty as "normal" or "easy". Thus, the FH group answers did not follow Siegmund et al. [28] observations. In Round 2, where the RiSE replication participants performed the same tasks of the Round 1, their responses were closer to the results observed by Siegmund et al. [28]. For the FH group, the three first tasks mostly (very) difficult, whereas only Task 2 was rated as difficult level by the CC group, which shows the FH group felt the tasks harder than the CC group. In Round 3, in which the participants targeted at a different system, their responses seemed to be homogeneous, regardless of the variability representation in the addressed code, rating the tasks mostly (very) difficult. This results indicate that the RiSE group might have an equivalent background knowledge to the those participating in the original study, whereas the ESE group show signs of having a deficient background.

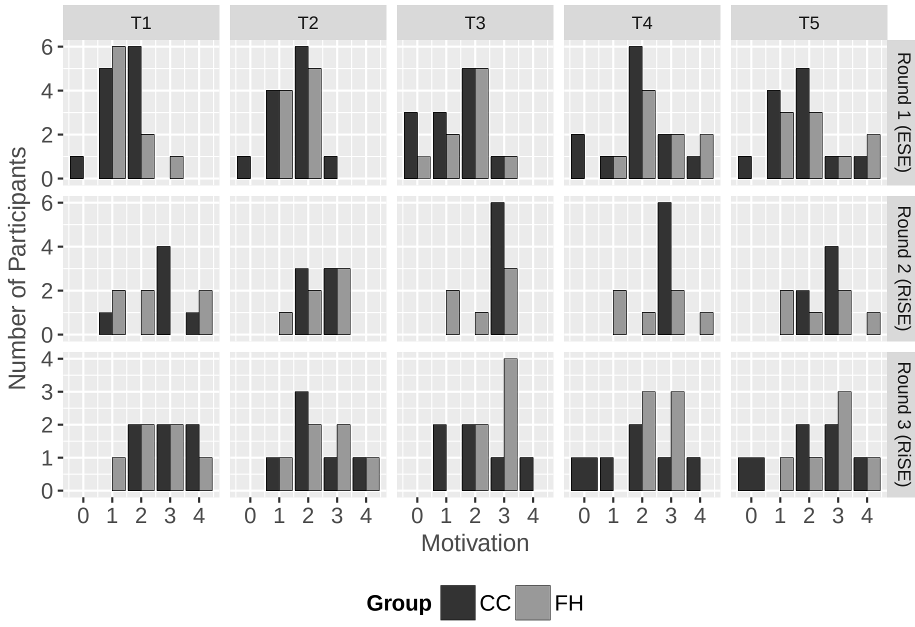


Figure 8.13 Overall feeling of motivation of the participants in each round. Scale: *(0)* Very unmotivated; *(1)* Unmotivated; *(2)* Normal motivation; *(3)* Motivated; and *(4)* Very motivated.

> “ For motivation, there is a tendency that participants of the CC group are more motivated to solve a task. This tendency might be caused by the fact that two participants of the FH group were unhappy to be in that group (as they told us). Thus, the FH version appears more difficult to participants and they did not like it. This can affect their performance, such that they work slower.
>
> *Siegmund et al.[28]* ”

Regarding motivation, both groups in all three rounds had equivalent motivation results, with the exception of Tasks 3 and 4 in Round 3. In Round 1, most ESE group participants' motivation reached the scale values from unmotivated to normal, whereas in the Rounds 2 and 3 the RiSE participants's motivation reached the scale values from normal, with the exception of the Task 3 in Round 3 – in which they felt unmotivated. Some of the participants from both groups mentioned they increased their motivation 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. On the other side, the RiSE participants (Rounds 2 and 3) felt the tasks mostly hard to solve, but kept motivated during their tasks, which show the consciousness of the participants of the situation. We believe this happened because their familiarity with the context of the experiment activities.

We now present Spearman correlations among the participants' *motivation*, tasks' *dif-*

ficulty perception and the results of our replications in terms of three dependent variables (`correctness`, `understanding`, and `response time`) to compliment the analysis. Table 8.7 shows in the first row the correlations between participants' *motivation* and the experiment dependent variables, whereas the correlations between the *difficulty* perception and each variables are shown in the second row. The correlations coefficients show significant ($p < .05$) moderate correlations between the participants' motivation and both the `correctness` and the `understanding` variables. This makes sense, since the motivation of the participants is what keeps them focused in their activity and helps them extract most of their abilities to transform in the effort needed to successfully complete the tasks.

Table 8.7 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`	0.589	0.705	0.758	0.676	0.136	0.359
`difficulty`	0.135	0.322	- 0.109	0.320	- 0.254	0.177

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

8.4.2 On the Answers to the Research Questions

Research Question 1: *"What are the differences regarding developers' effectiveness while performing program-comprehension tasks addressing software systems implemented with FeatureHouse and Conditional Compilation?"*

Regarding the first research question, we can enumerate two main findings, which we discuss in the following.

> **Finding 1:** there was no difference between the results from FH and CC groups in none of the rounds regarding understanding.

This means that the variability mechanism does not seem to have an effect in the overall comprehension of the source code and the difficulty in the maintenance task may have reasons other than how the variability is realized. In addition, both partial correctness and understanding had few occurrences in any of the rounds. This fact 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 on the program comprehension of unfamiliar code, specially when it comes to bug-finding tasks. The Prophet tool functionalities are 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.

Moreover, the plots of the replications results are not clear enough, so we could 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. On the other hand, the raw data about the participants' perception of the tasks' difficulty might help to explain why they fail to complete understand the code. While participants with superficial variability knowledge (ESE Group) mostly found tasks easy, the participants with strong foundations of variability (RiSE Group) weren't that excited. Our claim is that, although most participants failed to complete understand, those with the specific background required to the maintenance task are the ones who had the conscious commitment to the experiment. This fact may point out to the importance of the background knowledge of the sample 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.

> **Finding 2:** the response time range clearly decreased after the second task in all three rounds.

Despite the decreasing of the response time demanded to finish follow-up tasks, we observed neither increasing nor decreasing in the understanding and correctness levels. We did observed the increase of the motivation after initial tasks in the RiSE participants, which can explain the the decrease of the range of time spent to answer the final tasks. Surprisingly, we found a weak correlation between motivation and the response time. We claim that the low motivation of the ESE participants may have influenced in this result, since they are bigger in number of participants. The differences in the time range of the between the beginning and ending tasks may point out to the time needed for the participants to familiarize with the source code and also be associated to little unmotivation in the first tasks.

Research Question 2: *"What are the differences between software systems implemented with FeatureHouse and Conditional Compilation regarding developers' perception of the influence of confounding parameters while performing comprehension tasks?"*

Regarding the second research question, we can enumerate one main finding, which we discuss in the following.

> **Finding 3:** participants estimated the influence of confounding parameters differently depending on the group.

First, on the side of the contrasting perceptions, we believe participants estimated the influence of the confounding parameter "Education" differently because participants associated the training of each variability representation to their education. Since 92% of the RiSE group had at least one year of working experience and all of than at least mediocre level of knowledge of the object-orientation, the Conditional Compilation can more similar to the programming environment (*e.g.*, programming languages and code organization) they used to work with. In fact, the different constructs of FeatureHouse might have influenced most, once none of them have had contact with it prior to the experiment, whereas 58% (7 out of 12) had prior contact with Conditional Compilation in academia previously.

Second, on the side of the agreeing perceptions, Participants seemed to agree, under the addressed conditions, about the no or little influence of three of confounding parameters (the *Ability* – CP01 –, the *Familiarity with study object* – CP04 –, and the *Programming experience* – CP06) on the program comprehension. Indeed, the programming experience characterization of the participants have highlighted their confidence regarding their programming skills, which are definitely an important asset in these kind of maintenance tasks. Perhaps, their failure to understand and provide correct answers have more to do with the amount of cognitive effort demanded by the tasks, which were expected to be finished in a short time range due to the experimental time constraints. Indeed, this fact was highlighted in FH group feedback, in which the confounding parameter *Reading time* (CP07) was perceived as having the higher influence on the complete understanding and accomplishment of the tasks.

Regarding the confounding parameters of higher influence, participants seemed to agree about the moderate or high influence of three confounding parameters (*Familiarity with tools* – CP05 –, *Treatment preference* – CP10 –, *Programming language knowledge* – CP11). All these confounding parameters have something do to with the training in the specific abilities demanded during the tasks assigned to the participants. This fact may point out shortcomings in the training of the tools and the addressed variability representations; or even to the lack of functionalities the participants are used to work with, as discussed above.

8.4.3 Implications and Lessons Learned

The evidence raised in these replications are preliminary due to the exploratory nature of the study, but bring the opportunities of rethinking the way we are going to carry out future experiments. They highlighted the peculiarities of the experimentation on the influence of differences of variability representations on program comprehension. For instance, it is hard to assure equivalent knowledge of both variability representations under evaluation by training the participants prior to the experiment session, which is important to assure the results are comparable. Especially because FeatureHouse is an emerging technology, most developers are about to have the first contact during the experiment activities.

8.5 THREATS TO VALIDITY

In this section, we discuss threats to the validity of this empirical study. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion validity.*

8.5.1 External Validity

We identified some threats that may limit the ability to generalize the 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 [79]. In this sense, we recruited students with different experience levels, as showed by the characterization questionnaire applied prior to the experiments. This fact, makes our sample an important population.

8.5.2 Internal Validity

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

Confounding constructs may affect the findings. For instance, the motivation and the difficulty of each task might have affected the participants perception of the influence of the variability representations on the tasks comprehension. The issue here is that we cannot assure the tasks were not too hard to be solved by the group of participants we recruited for the experiments. However, we believe this threat was mitigated in our study by the attempt of balance the difficulty with pilot studies and using the same artifacts from previous experiments. Regardless of such threat, results are still important to research community as it can guide future research on the matter.

8.5.3 Construct Validity

Construct validity refers to the fact that the construct (i.e., developers' effectiveness) was not operationalized correctly. Perhaps, the way we coded the understanding and correctness regarding the answers of the participants places a threat, because of the used pseudo-ordinal measure scale. The fact that only a few partial understandings/correctness might be a sign of such threat. As the participant's understanding cannot be measured easily by others, we tried to mitigate the threat by being as objective as possible to have a clear idea of what was being measured for each one.

8.5.4 Conclusion Validity

Our discussions were based on a rather small sample, which limits the power of used tests to reveal a true pattern in the data. We mitigated such a threat by employing well-known measures to conduct our analysis. Another observed threat is the fact that the results of the recruited groups (ESE and RiSE) might not be comparable, since they have different background. We are aware of this fact and tried to alleviate it by comparing the results both inter and inner groups. In addition, we believe to have exploited the available data from different statistical analysis, which strengthens the carried analysis.

8.6 CHAPTER SUMMARY

This chapter presented the second and the third controlled experiments of the VICC family, which addressed the influence of the use of two variability representations (Conditional Compilation and FeatureHouse) on program comprehension. We detailed the planning and execution of the VICC2 and VICC3, while discussing metrics, participants, tasks, support material, experiment design, and variables. We also highlighted the experiment execution and presented a detailed discussion on the results of the experiment.

The experiments investigated two research questions. The first one, regarding the differences in participants' effectiveness while addressing bug-fixing tasks in feature-oriented software using such variability representations. The second one, regarding participants' perception of the influence of confounding parameters on program comprehension. The results showed no statistical difference on the understanding and correctness of participants' answers, whereas different perception of the influence of the confounding parameters whether the participants addressed the experiments tasks in the CONDITIONAL COMPILATION or the FEATUREHOUSE group.

Finally, we enumerated a number of threats to validity identified during the different phases of the experiment. Next chapter presents the planning, execution, results of the VICC4.

Chapter 9

VICC4: ON THE DEVELOPERS PERCEPTION OF DEMANDED EFFORT

This chapter reports the last experimental study of the VICC family (VICC4). We present each of its phases, as well as we discuss its results. We intend to gather evidence on the aspects driving the developers in the process of understanding variable code implemented with CONDITIONAL COMPILATION and FEATUREHOUSE. We used the focus group qualitative approach [19] to pursue such goal, while merging the participants assessment with the gathered evidence from the previous studies. In addition to the Chapters 6, 7, and 8, this one covers the **Research Goal 2** (pg. 5).

This chapter consists of six main sections. Section 9.1 presents the study setting of VICC4, including the planning, and the execution. Section 9.2 presents the data collection procedures. Section 9.3 presents and discusses the results of the study regarding the individual feedback collected from the participants. Section 9.4 discusses participants' answers to the focus group questions. Section 9.5 discusses the research questions with regarding to the gathered data. Section 9.6 presents the threats to validity identified during the evaluation.

9.1 STUDY SETTINGS

This section presents the pursued research questions, the planning, and the execution of this final experimental study.

9.1.1 Research Questions

In the following, we present the research questions guiding our experimental study. Unlike the previous studies we do not state any hypotheses, since our goal was to gather evidence from a qualitative study and add them to the observations from the prior empirical evaluations performed. In this sense, no metric is previously defined, once data was gathered from the coding of the textual transcripts from the focus group interviews and the applied feedback questionnaire. Next, we describe each research question.

RQ: Which aspects impact the developers comprehension of variability implementation in the maintenance of feature-oriented software?

This is rather a general research question aiming at covering all aspects to be considered while analyzing the comprehension bottlenecks involved in the maintenance process. In order to go deeper in the analysis of such aspects, we split this general question into three research questions.

RQ$_a$: How do developers approach the variability implementation comprehension?
Rationale: since the code are rearranged from CONDITIONAL COMPILATION to FEATUREHOUSE and there is a different set of tools available for each of them, our hypothesis is that developers might have to change their strategy to address unfamiliar code in the different variability representations.

RQ$_b$: Which aspects hinder variability implementation comprehension?
Rationale: given the different nature of how developers program either in CONDITIONAL COMPILATION or FEATUREHOUSE, in this question, we investigate specific factors from either of them hindering comprehension.

RQ$_c$: Which aspects ease variability implementation comprehension?
Rationale: conversely, the differences in the programming models may incur in different positive factors, each. In this question, we look for those aspects playing in favor of the variability representation.

9.1.2 Planning

Alike previously reported empirical studies in this thesis, the planning concerns the experimental 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.

9.1.2.1 Design. The design of this study was a mix of controlled experiment with a focus group in the end. More specifically, the experimental study was performed in three phases:

Phase 1 where the participants performed three technical tasks using CONDITIONAL COMPILATION and FEATUREHOUSE (the tasks are described later). This phase serves the purpose of familiarizing the participants with the novelty of the FEATUREHOUSE and its capabilities.

Phase 2 where they fulfilled a feedback form. This phase serves the purpose of getting an individual perception of the participants. We thought this might be needed in case some of them kept wordless during the follow-up phase, what actually happened.

Phase 3 was the actual execution of the focus group. The idea behind the focus group is to identify aspects impacting the comprehension of both variability representations (CONDITIONAL COMPILATION and FEATUREHOUSE) through the coding of the dialogs' transcriptions.

In total, 10 graduate students took part in our experimental session. All of them recruited from a Software Reuse course, which is an optional course in the Federal University of Bahia's Computer Science Graduate program. Each participant worked on an individual workstation until they finish *Phase 2*. Afterwards, we get them together to *Phase 3*.

9.1.2.2 Tasks. Table 9.1 describes each of the three tasks the participants had to answer prior to the actual focus group session. They took around 50 minutes to finish the tasks. These tasks were designed in order to force the participants to understand the source-code as a whole, including aspects of data-flow, which was needed to perform the Tasks 2 and 3. Task 1 is probably the easiest one and it was used to allow the participants to familiarize with the system.

9.1.3 Execution

In the experimental session, we introduced the study, as well as its goals and procedures to the participants. This phase took around 35 minutes. Next, we described the tasks they had to perform and defined 40 minutes as the desired time we expected they to finish it. In fact, we did not use this as a hard deadline and some participants took a couple of minutes more to finish.

9.1.3.1 Subjects Characterization. Appendix B presents the questionnaire used to characterize the subjects programming experience. As the participants were at the time taking part in a Software Reuse course, which includes the implementation of software product line in the syllabus, we did not include questions regarding this topic.

We asked the participants to compare themselves against their group-mates and professional developers with 20 years of experience. Figure 9.1 shows the answers of the group.

Table 9.1 Tasks defined for `MobileMedia` in the focus group session.

Tasks	**Description**
Task 1	Find the exact place – Class, Line of Code, (and Feature) – where the video controller is being initialized in both implementations (CONDITIONAL COMPILATION and FEATUREHOUSE);
Task 2	Analyze the source-code in order to identify which features must be selected in order to make this part of the code (video controller initialization) available in a final product;
Task 3	Is there any precedence between/among the features? Please, justify.

The comparison with their group-mates are identified by the key "Students", whereas the comparison with the developers with the key "Professionals". This comparison shows approximately one third the participants see themselves as less or more experienced as their mates and 40% as equally experienced. In addition, only 10% self-assessed as more experienced than the those professionals.

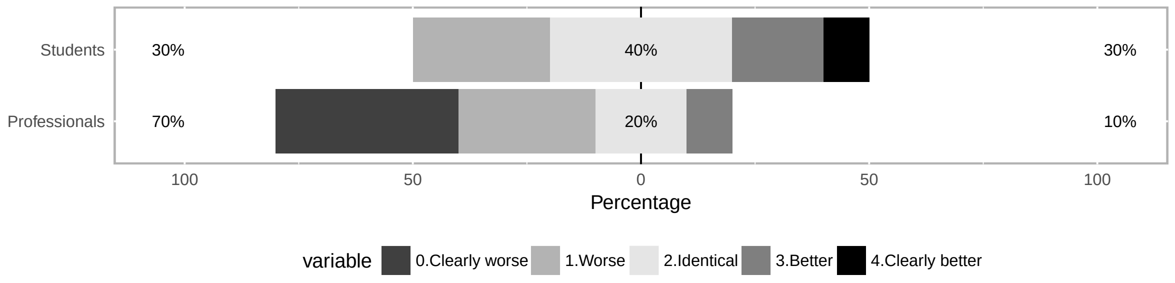


Figure 9.1 Subjects' programming experience self-assessment.

9.1.3.2 Support Material. From the experience acquired from the previous experiments, we decided this time to provide the complete FeatureIDE infrastructure to the participants. This fact contributes to the observation of the actual impact of available tools in the participants comprehension tasks. The participants also had the feature model of both versions in hands while they analyzed the target system during the session.

9.1.3.3 Pilot. The original idea was to carry out a (*quasi-*)experiment [18] addressing data-flow tasks. In this sense, in order to balance the study in a way we could extract the best from it, we performed a pilot with three Master students from the *Technical School of Würzburg-Schweinfurt* (FHWS) located in Germany, while they were working at UFBA in a research collaboration project between both institutions. The german students had only superficial knowledge of SPL engineering, conditional compilation, and FeatureHouse in opposition to the background of those recruited from the Software Reuse course, which led us to believe we would have low to no success given the complexity of the tasks. In fact, they found the tasks to be hard. Therefore, we decided to maintain the tasks with lower importance in the study in favor to add a new phase, the focus group. In addition, they contributed with the factors that might have impacted in their success in the assigned tasks, which we considered while refining the study and helped in the construction of the individual feedback form.

9.2 DATA COLLECTION

This section presents how we collected the data to proceed with the analysis. We anticipated in the design session, we collected data by three different ways: *(i)* the answers form in the Phase 1; *(ii)* the feedback form in the Phase 2; and *(iii)* the transcription of the focus group available in Portuguese language in Appendix E. However, as the sample is small and we would be unable to draw any strong conclusions, we decided then to focus on the last two phases of the study.

9.2.1 Individual Feedback Collection

This section describes how the feedback was individually collected from the participants after the tasks. One of the primarily reason we considered to construct such a questionnaire is that we anticipated some of them with few or no contribution during the focus group session. In this sense, we elaborated one question – *What was the impact of the following factors in the comprehension in source-code?* – with a set of predefined impact factors concerning each of the paradigms addressed in this task (Conditional Compilation and FeatureHouse). Additionally, we asked two questions to an overall evaluation of how hard it was for them to perform the tasks. Some of the predefined factors were already addressed in the previous studies, some of them are new. Next, we discuss each of these groups of questions.

Tables 9.2 and 9.3 show the questions regarding the pre-defined factors concerning the Conditional Compilation and the FeatureHouse paradigms, respectively. All the questions were rated in a five-point Likert-scale [83] with values ranging from "too easy/no impact" to "too hard/high impact". We summarized these factors based on two main sources of evidence. The first, the experience acquired by the author of this thesis while refactoring the Conditional Compilation version of `RiSEEvent` to FeatureHouse. The second, the evidence raised in the previous experimental studies.

Table 9.2 Predefined impact aspects used for the participants individual feedback.

ID	**Conditional Compilation Factors**
(CC-Q1)	this factor concerns the "quantity of variation points", *i.e.*, the number of the annotations present in the source-code;
(CC-Q2)	perhaps, the existence of "variations points scattered" throughout the entire project might also impact in the tasks of dealing with unfamiliar code;
(CC-Q3)	the existence of "logical expressions in the variation points", can also impact in the comprehension;
(CC-Q4)	we are interested in the analysis of the impact of "long fragments of variable code", which is the code between the openings (#ifdef X) and the endings (#endif) of annotations blocks.
(CC-Q5)	we are also interested in the comprehension of the "data-flow" in the different variability representation code;
(CC-Q6)	the "tool support" was one of the factors most mentioned in the previous studies. Therefore, we added this it here one more time check whether the ratings would change.

Table 9.3 Predefined impact aspects used for the participants individual feedback.

ID	**FeatureHouse Factors**
(FH-Q1)	One intrinsic characteristic of the feature-oriented programming with FeatureHouse is the "class refinements", which produces duplicated classes and eventually the high number might impact in the comprehension.
(FH-Q2)	Another characteristic is the feature-based "source-code organization". We asked the participants about the impact of this factor.
(FH-Q3)	At first, the "features precedence" might not be noticed at first sight, the participants perceived impact can help on the understanding of how hard it is to get this information from the context being addressed.
(FH-Q4)	We are also interested in the comprehension of the "data-flow" in the different variability representation code;
(FH-Q5)	As well as the class refinements, the "method overloading" might also be hard to detect while programming.
(FH-Q6)	In addition to the method overloading, the"method refinements" might also be hard to detect while programming.
(FH-Q7)	the "tool support" was one of the factors most mentioned in the previous studies. Therefore, we added this it here one more time check whether the ratings would change.

9.2.2 Focus Group Data Collection

This section describes the questions used to collect data during the focus group session. Table 9.4 shows the questions. They are both *closed-* and *open*-ended questions. Since the closed-ended questions lack of stimulus for the participant to give a long and elaborated answer, we did not include them among the questions while planning the questionnaire. However, they did happen in the focus group session to complement the open-ended questions during the discussion.

We transcribed the audio recording of the focus group session to resort on the coding technique. More specifically, we are going to use two coding levels [19]: *(i)* open coding – where the main rule is to segment the transcription text into similar groupings in order to identify preliminary categories of information about the phenomenon under analysis –; *(ii)* axial coding – the open coding allows the researcher to concatenate similar ideas concerning specific aspects under a more general statement or concept/themes.

9.3 INDIVIDUAL FEEDBACK RESULTS

This section presents the raw data and discuss the individual feedback of the participants collected with the feedback form in the experimental study.

Table 9.4 Focus group questions.

Order	Questions
Question 1	What were your first impressions of each paradigm?
Question 2	What could make you more effective in your tasks?
Question 3	What were your feelings about each paradigm? Fatigue, tiredness, paradigm preference.
Question 4	What tool or strategy have you adopted during the tasks to comprehend unfamiliar code? Have you changed the strategy to address the FEATUREHOUSE code?
Question 5	The worse thing about CONDITIONAL COMPILATION is...
Question 6	The worse thing about FEATUREHOUSE is...
Question 7	The best thing about CONDITIONAL COMPILATION is...
Question 8	The best thing about FEATUREHOUSE is...
Question 9	While using CONDITIONAL COMPILATION, the first thing I did was...
Question 10	While using FEATUREHOUSE, the first thing I did was...
Question 11	Would you like to elaborate a bit more about the maintenance difficulty of each approach?
Question 12	Have you noticed precedence among features, regarding the methods refinements?

9.3.1 Influence Drivers

Figure 9.2 summarizes the subjects rating of the impact the predefined factors detailed previously (Tables 9.2 and 9.3). We can identify the rating for each factor in the figure by the identifiers from these tables (ID). The bars are ordered in decreasing order of impact, *i.e.*, the predefined factors with highest impact comes first.

If we look to each paradigm individually, we can see the "Long fragments of variable code" (CC-Q4) was rated as the most impactful factor among those from the CONDITIONAL COMPILATION paradigm, whereas the "quantity of class refinements" (FH-Q1) is by far the most impactful among those from the FEATUREHOUSE paradigm. On the other hand, the "Tool support" for CONDITIONAL COMPILATION showed up with the lowest ratings, whereas only 10% of the participants found the FEATUREHOUSE "Code organization" a factor of several impact. In fact, the participants rated the impact of most factors listed in the FEATUREHOUSE tasks mostly as indifferent, with the exception of "Code organization" (FH-02) and "Tools support" (FH-07), which most participants rated with few to no impact in the comprehension.

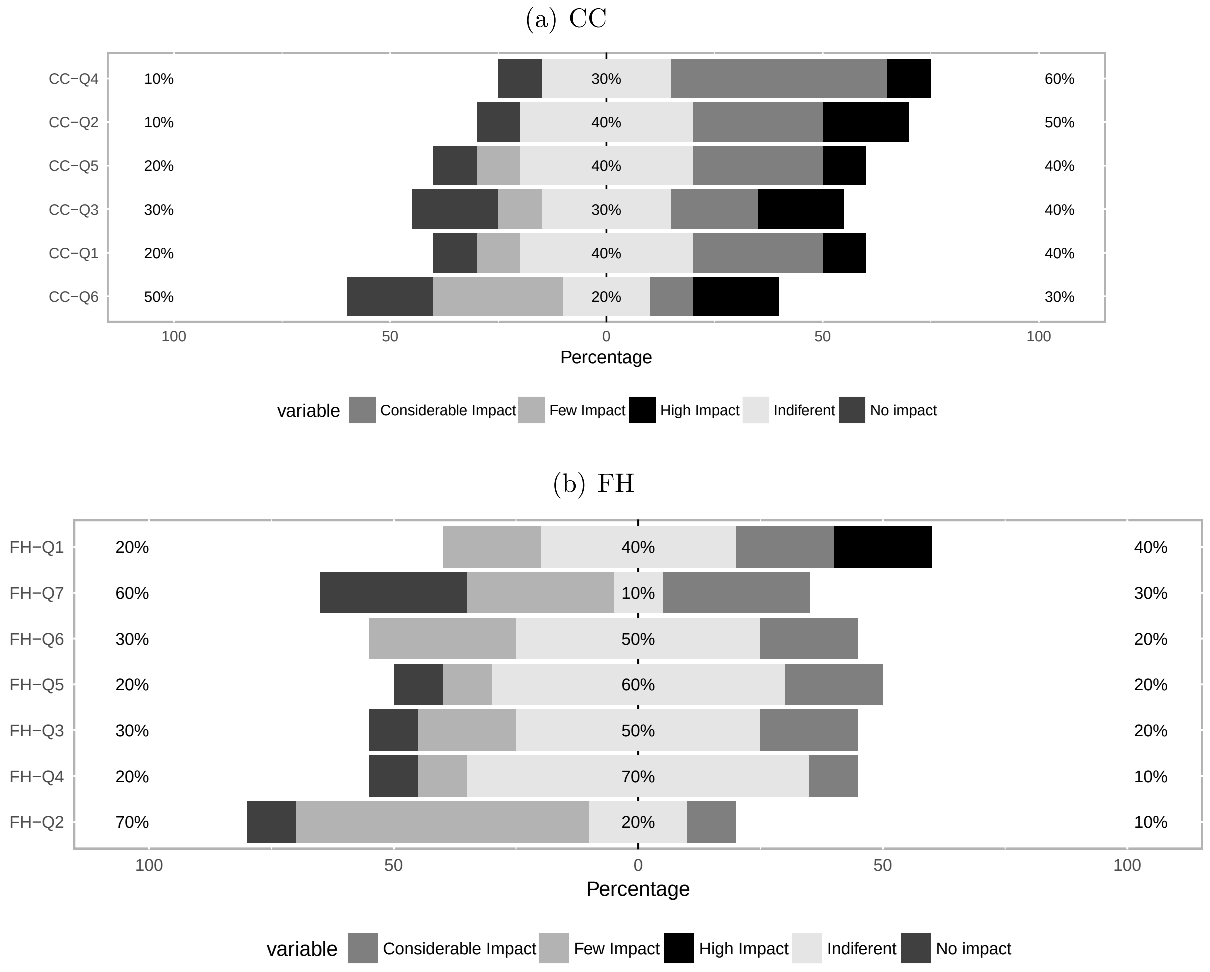


Figure 9.2 Subjects rating of the impact the predefined aspects.

9.3.2 Tasks' Difficulty Perception

We relied on two general questions to assess the overall difficulty of the executed tasks: *(i)* "Generally speaking, how difficult were the tasks?" and *(ii)* "Generally speaking, how difficult was to understand the system source-code?". Figure 9.3 shows the answers to them regarding each variability representation CONDITIONAL COMPILATION and FEATUREHOUSE. The answers show an equivalence in terms of difficulty. In addition, half of the participants found the tasks to have usual difficulty both in a general way or in terms of comprehension effort.

9.4 FOCUS GROUP QUESTIONS ANSWERS

In this section, we proceed with the analysis of the focus group session transcriptions. The focus group questions can be subset in four main group of questions: *(i)* the questions concerning the strategies adopted by the software engineers while addressing code using CONDITIONAL COMPILATION and FEATUREHOUSE; *(ii)* the questions concerning

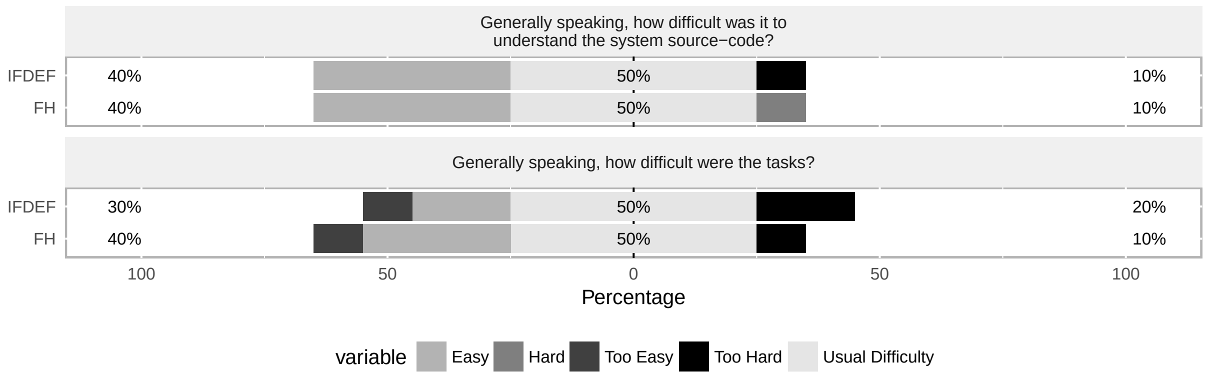


Figure 9.3 Answers to the questions to assess the overall difficulty of the tasks.

the factors hindering the best performance of the software engineers; *(iii)* the questions concerning factors easing the maintenance work with each variability representation; and *(iv)* the general observations questions. Tables 9.5 and 9.6 enumerate all *benefits* and *drawbacks* identified in the participants' speeches during the focus group. We discuss the more important of them in the following while presenting the focus groups questions answers.

9.4.1 Group 1: Comprehension Strategies

Three questions are in this group: questions 4, 9, and 10. Each of these questions concern the way developers handle the comprehension of unfamiliar source code. Depending on the strategy they used to accomplish their task, different kind of tools would yield different results in terms of effectiveness. Next, we discuss the finding of each of these questions.

Question 4 explicitly asked the participants about the tools and strategies they adopted while addressing the comprehension tasks. In addition, we asked whether they had to change their usual strategy to address unfamiliar code, because of the implementations using CONDITIONAL COMPILATION and FEATUREHOUSE. Question 9 and 10 asked the participants what were their first attitude to comprehend the source code using either CONDITIONAL COMPILATION or FEATUREHOUSE, respectively. The idea was to identify whether the first action follows a different pattern depending on the variability representation.

We identified mainly two groups of participants, those who relied only on the search tools and those who resort of additional tools. In the first group, a participant stated to have addressed both versions code with the same strategy, as shown by the following quote.

“ I ended up by using the same strategy I used in the other. (. . .) To search and than identify the artifacts. I repeat for both.

Focus Group Participant ”

Other two participants of this first group made use of their previous game development

Table 9.5 Benefits identified in the answers to the focus group questions.

Conditional Compilation
To read the code is enough to understand the variability.
It is good in cases of several variation points in the same file.
The search is enough to find the feature annotations.
Background knowledge was useful to the working strategy.
Modularization and architecture may alleviate the annotation scattering problem.
The programming model is simple.
FeatureHouse
Variability, constraints, and configurations visualization are straightforward.
The configuration management tool had some use to understand features constraints and boosted effectiveness.
Background knowledge was useful to the working strategy.
The configuration management helps in the troubleshooting and debugging.
Collaboration diagram is faster than search for annotations.
Valid configurations visualization.
Code readability (*i.e.*clean code)
It is better than Conditional Compilation in cases of scattered variability.
The visual organization is interesting.
Collaboration diagram provides a good overview about each feature
Code obfuscation is not a problem.
FeatureIDE views are useful to novice programmers.
It has good traceability.
To locate the maintenance-target code for maintenance seems straightforward
The feature code traceability can produce gains in the development.
The error propagation seems to be lower than Conditional Compilation due the feature-based modularization
The collaboration diagram is more comfortable to work with than a traceability matrix.

experiences, which can be seen in the following quote.

> “ I have already worked a bit with game development, so it was straightforward to me to search for a “init” method. In the way that, in the place I found a variable declaration with this method call, than it should be the one i was looking for. This means I did not look to feature or other stuff, just looked for the declaration.
>
> *Focus Group Participant* ”

On the other hand, participants of the second group approached the code using differ-

Table 9.6 Drawbacks identified in the answers to the focus group questions.

CONDITIONAL COMPILATION
Code obfuscation is a huge problem (Excessive amount of annotation).
Lack of visualization tools supporting annotations (*e.g.*, colors, block folding)
Lack of traceability tools (justified with lack of training)
Maintenance of CONDITIONAL COMPILATION might be bottleneck for both large and small systems.
Good programming practices may also alleviate the annotation scattering problem.
Annotations turns the maintenance harder.
The use of feature tags in comments hinder comprehension (*e.g.*, searching process) when the annotation mechanism is not a native construct, such as in Java.
OO does not require a feature-oriented mindset, which turns up to be too flexible to implement variability and requiring additional effort from developers to handle such construct.
Although CONDITIONAL COMPILATION works, it is probably not the best solution to use in large systems.
To locate the maintenance-target code requires global search.
CONDITIONAL COMPILATION is error propagation prone
FEATUREHOUSE
It is a FH problem the amount of clicks to reach the source code files.
Lack of traceability tools (justified with lack of training).
Lack of visualization tools support.
Too much duplicated classes.
It is hard to keep a big picture of the implementation (*e.g.*, memory allocation problem, method overriding.)
It is required to understand the composition problem in order to comprehend how the variability is being implemented.
Feature precedence is harder to notice for novices.
Big effort needed to manage the shared resources.

ent strategies. While using using CONDITIONAL COMPILATION, they relied on the search tools only, whereas they resort of the feature model/configuration management and the collaboration diagram tools to solve the tasks with FEATUREHOUSE. The quote below shows the behavior of one of the participants.

> “ As soon as I discovered the FEATUREHOUSE had that small tree[1], I preferred to create a configuration, unselect all features non-mandatory, than I verified, selected the features I did not want and saw that everything built automatically. This made me solve my problem infinitely faster.
>
> *Focus Group Participant* ”

Regarding the first action, we did not identify any different behavior other than the use of the global search tool by the participants addressing CONDITIONAL COMPILATION code. On the other hand, we identified that some participants prefer to use first the collaboration diagram, others the feature model while using FEATUREHOUSE. None of the participants mentioned the change of strategies to address unfamiliar code. However, it is clear by the use of these new tools the deviation from the usual strategies.

9.4.2 Group 2: Hindering Factors

Three questions are in this group: questions 3, 5, and 6. Each of these questions aimed at the identification of aspects that might hinder the comprehension of the source code. It is important to distinguish aspects of each variability representation hindering comprehension, because it would be possible to mitigate the effect of such factors in the daily activity of developers.

Question 3 asked the participants to elaborate on the different feelings (*e.g.*, Fatigue, tiredness, variability representation preference) during the execution of the technical tasks using each of the variability representations. Question 5 and 6 asked the participants what was the worse thing they found while working either with CONDITIONAL COMPILATION or FEATUREHOUSE during the comprehension tasks, respectively.

The following quote regarding the superiority of FEATUREHOUSE concerning to the maintainability of the addressed variability representations points to the overall feedback of the participants majority.

> “ I have the feeling that it is unanimous that FEATUREHOUSE is better than CONDITIONAL COMPILATION. By thinking on the maintenance of that CONDITIONAL COMPILATION code, it would be an headache to me.
>
> *Focus Group Participant* ”

Later, the same participant claimed the maintenance issue would appear regardless of the system's size. Although none of the other participants raised objections, some raised arguments in favor of a balance between both CONDITIONAL COMPILATION and FEATUREHOUSE highlighting the importance of the best use of each one, as shown by the following quote.

> “ The CONDITIONAL COMPILATION code is highly obfuscated. However, if you use the search tool to look for "annotations[2]" it makes your life easier. You go straight to the point. You click *Find* and all the annotations you expect to find will be in your hands. At the same time, when using FEATUREHOUSE you can identify them in the (collaboration) diagram. You go faster to the right point. Therefore, I believe you need to know how to use the available resources. It is a balance.
>
> *Focus Group Participant* ”

Regarding the hindering aspects itself, the participants mentioned mostly factors from the CONDITIONAL COMPILATION representation. Besides the hindering aspects already known (*Code obfuscation*, *Excessive amount of annotations*), we also identified the following factors.

The use of feature tags in the comments: according to the participants, this practice can both cause misunderstandings and turn the search for annotations in time of maintenance harder.

The lack of feature-orientation mindset: according to them, differently from the FEATUREHOUSE, the object-orientation paradigm does not force the software engineer to think in terms of features, which requires additional effort for them to handle such additional construct.

Regarding the hindering aspect identified in the FEATUREHOUSE variability representation, we extracted the following from participants observations.

There is too much duplicated classes: according to them, the amount of classes with the same name turns it harder to understand and follow the code. For instance, it is hard to keep track memory allocation and deallocation –, which may not be a problem in `Java`, but certainly is in `C/C++` – or method overriding.

It is hard to keep a big picture of the implementation: according to them, still the number of duplicated classes, in conjunction with the amount of clicks needed to get to the actual code in the features folder hinder comprehension and consequently the execution of maintenance tasks.

9.4.3 Group 3: Facilitators Factors

Three questions are in this group: questions 2, 7, and 8. Each of these questions aim at the identification of aspects that might ease the comprehension of the source code. It is important to enumerate the aspects of each variability representation hindering comprehension, because it would be possible to use them as inspiration to improve the supporting tools of the daily work of developers. Question 2 asked the participants about what could make their work more effective while using either CONDITIONAL COMPILATION or FEATUREHOUSE. In addition, the questions 7 and 8 asked them about the best thing they found about working with CONDITIONAL COMPILATION and FEATUREHOUSE, respectively.

The participants pointed out the lack of tools supporting annotations as a set of effective-oriented enhancements that might help in the comprehension and consequently in the maintenance activities. Among these tools, the participants mentioned visualization tools and also tools to help with the identification of annotations of interest.

> “ A software visualization tool for this (to be more effective) would help. For both variability representations.
>
> *Focus Group Participant* ”

Additionally, the available tools are too Eclipse dependent and the participants who do not use it in daily work had difficulties to address their tasks.

> “For me, to both variability representations, as I do not use Eclipse, I found difficulties to use the tools functionalities and to look for the files.
>
> *Focus Group Participant* ”

9.4.4 Group 4: General Observations

Three questions are in this group: questions 1, 11 and 12. Question 1 asked the participants about their first impressions of each variability representation, whereas the Question 11 asked the participants to elaborate more about the difficulty to maintain code written in each variability representation. Question 12 asked whether the participants noticed any kind of precedence among features, regarding the methods refinements in the variability representations.

Regarding their impressions, although the participants did not present any strong objections to the use of CONDITIONAL COMPILATION, it seems they would prefer to use the FEATUREHOUSE to implement variability. They claimed the tool support available to FEATUREHOUSE helped them to best approach the tasks. The following quotes show this feeling of them.

> “The variability representation with CONDITIONAL COMPILATION is interesting when we have a concentration of variation points inside a single source code file. (. . .) However, when the variability starts to be scattered, then the FEATUREHOUSE seems to be interesting, because we can easily see the project structure. The bad side is the many clicks we need to reach the source code files.
>
> *Focus Group Participant* ”

> “There is also that tool, which shows the feature “Base” and where it was implemented. . . [Collaboration Diagram] Exactly. I think it helps to get an overview of how the feature is implemented. On the other hand, while using CONDITIONAL COMPILATION it just use the search to find the annotations, which is also interesting and you can also walk through the source code.
>
> *Focus Group Participant* ”

Regarding to answers to the maintainability (Question 11), the participants pointed out the benefits of the improved traceability of FEATUREHOUSE in comparison with the CONDITIONAL COMPILATION. Besides, the fact that the development team changes constantly, according to the participants this quality of FEATUREHOUSE would be beneficial in such situations, which demands maintenance knowledge management – usually accomplished with traceability matrix.

> “Traceability is the biggest need. Because something that worries me is how scattered is the source code, since the more scattered the code, the bigger

the error proneness of the source code. [With FEATUREHOUSE,] I would go straight to that package and perform the needed changes.

”

Focus Group Participants

Moreover, none of the participants claimed to have perceived the existing precedence among features in FEATUREHOUSE in binding time. In fact, one of them clearly stated he did not perceive, instead he pointed out to potential issues while programming with FEATUREHOUSE. As in FEATUREHOUSE several duplicated classes might exist throughout the entire project, the management of shared resources (*e.g.*, memory) might be problematic, as discussed in the RQ_b.

9.5 RESEARCH QUESTIONS DISCUSSION

In this section, we discuss our macro research question by answering their sub-questions. The answer to this general question together with their sub-questions can help future research to pursue the enhancement of processes and tool support for maintenance tasks of software in the presence of variability. To this end, we take into consideration the raw data presented in the previous sections and the possible implications of the identified influence factors on program comprehension to future research. We now use each of those defined subsets to answer both the sub-questions and the macro research question.

9.5.1 RQ$_a$: How do developers approach the variability implementation comprehension?

According to the presented answers for the questions of Group 1, participants rely on the search tools regardless of the variability representation. More specifically, global search to find all the CONDITIONAL COMPILATION annotation tags and local search inside the files of the target feature folder. In addition, we can point out that they also made use of visualization tools available in the FEATUREIDE to support FEATUREHOUSE development, such as the “Collaboration Diagram”, the “Feature Model", and the “Configuration Management”. Moreover, the background knowledge of each participant has heavily influenced in the way they approached the tasks. While some participants with game development background just searched for similar keywords, the one with traceability matrix experience preferred to use the collaboration diagram.

9.5.2 RQ$_b$: Which aspects hinder variability implementation comprehension?

According to the presented answers for the questions of Group 2, the factors making the tasks accomplishment harder are different regarding the variability representations. In CONDITIONAL COMPILATION, the scattering of feature annotations, the code obfuscation caused by the excessive amount of annotations, as well as the use of annotation tags inside the comments are the most mentioned factor hindering variability comprehension. Conversely, in the FEATUREHOUSE, the number of occurrences of the duplicated classes and the amount of clicks needed to reach the source code files were the ones more emphasized as the factors hindering program comprehension in this variability representation.

Regarding the bottlenecks of each variability representation, the focus of participants' complaints have shifted slightly in comparison to our preceding experiments in the VICC family. While in our previous studies with bug-finding tasks, the participants complained about the lack of proper debugging tools, in this study where the tasks demand basically comprehension effort, they suggested an improvement in the set of visualization and search tools available. They raised these enhancements as a mean to improve their effectiveness in the maintenance tasks.

9.5.3 RQ$_c$: Which aspects ease variability implementation comprehension?

According to the presented answers for the questions of Group 3, the factors making the tasks accomplishment easier are also different regarding the variability representations. In CONDITIONAL COMPILATION, participants were unanimous that the search tool is the first tool they have in mind to accomplish their tasks. They also found the programming model simpler than the FEATUREHOUSE option, since it was enough to read the code to become aware of its purpose. On the other hand, in FEATUREHOUSE, the code organization was pinpointed as an interesting factor of this variability representation. Additionally, participants found the collaboration diagram and the configuration management tools highly useful to accomplish the tasks.

9.5.4 RQ: Which aspects impact the developers comprehension of variability implementation in the maintenance of feature-oriented software?

Each of the sub-questions have led us to identify some of the referred aspects subject of investigation under this research question. As those aspects were already discussed previously, we now focus on those highlighted in the presented answers for the questions of the Group 4. This group concerned general observations regarding the maintainability of the code using each of the variability representation under investigation in this thesis.

The available tools supporting the maintenance activities have high impact on the first impressions of the participants of our focus group. In addition to the search tools and the feature models available of both representations, there were a couple of tools available only for FEATUREHOUSE. What might have favored this representation against the CONDITIONAL COMPILATION regarding the traceability of artifacts. Such difference was not a misplanned action, but rather a reproduction of the situation they would be exposed in the real world. To the best of our knowledge, apart from the configuration management used in the Linux Kernel build system (KBuild), which have functionalities similar to the configuration management of the FEATUREIDE, there is no equivalent tool to the collaboration diagram or the FEATUREIDE Outline[3] available to CONDITIONAL COMPILATION.

All these observed points are subject to further investigation to better understanding of the influence of such factors on the program comprehension of software in the presence of variability.

[3]The FEATUREIDE Outline is a tool that aggregates all the duplicated classes in only one tree view.

9.6 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of this empirical study. We believe that presenting such detailed information may contribute to further research and replications of this study [18], which may be built upon the results presented herein. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion validity*.

External validity. As a qualitative study, the results cannot be generalized to all software engineers and practitioners. However, this is the fourth study of a family of complementary studies providing evidence from different sources, which allow us to observe and make observations based on the whole experimentation experience context. Indeed, the approach of this study enabled deep insight to emerge from participants' perspectives that can be later tested for generalization in other contexts.

Internal validity. There are possible threats that may happen without researcher's knowledge affecting individuals from different perspectives, such as *(i)* the maturation effect, which we mitigated disregarding their answers to the tasks and focusing on their feedback and free speeches data; *(ii)* the tasks instrumentation – concerning the artifacts and forms used during the study session –, which we mitigated with the review of the study design and artifacts by more experienced researchers of our research group.

Construct validity concerns generalizing the result of the experiment to the concept or theory behind the experiment [18]. It might have to do with the preoperational explanation of constructs, which means that the concepts are not well defined before translated to measures or treatments. We mitigate this threat by using the same concepts throughout the whole family of studies and using only consolidated experimental designs already used in the literature. Confounding constructs may affect the findings. For instance, the difficulty of each task might have affected the participants perception of the impact of the variability representations on the tasks comprehension. We believe this threat could be mitigated since participants felt mostly the same level of difficulty.

Conclusion validity threats are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [18]. As the goal of this study was to identify a set of aspects worth of further investigation rather than draw any conclusions about them, we found this validity not applicable to our context.

9.7 CHAPTER SUMMARY

This chapter presented a focus group on the influencing factors on program comprehension in the presence of variability regarding CONDITIONAL COMPILATION and FEATURE-HOUSE. More specifically, we discussed the influence of predefined factors extracted from previous studies on the focus group participants program comprehension. Afterwards, we presented the results of the factors extracted from the free speech session of the focus group. Next chapter presents the concluding remarks and future directions of this thesis.

PART V

CONCLUSIONS

Chapter 10

CONCLUSIONS AND FUTURE WORK

The main cost of the software life cycle is the maintenance phase and one of its most time-consuming and tedious tasks is understanding source code, which makes software engineers avoid it whenever it is possible [39]. This fact raises the importance of ease such tasks as much as possible. On top of that, the presence of variability in feature-oriented software makes program comprehension even harder. Thus, research on program comprehension can help both researchers and practitioners to understand how software engineers approach unfamiliar code, specially in the presence of variability. Next, we present the contributions made by this thesis and directions to future work.

10.1 THESIS CONTRIBUTIONS

In this thesis, we pursued a twofold, but complementary goals:

- First, we extended the umbrella of feature-oriented software development to cover `JavaScript`-based software systems.
- Second, we contributed to the evidence corpus on the understanding of the influence of different variability representations (namely, *compositional* and *annotative* approaches) on program comprehension tasks.

Each of these goals were addressed individually and reported in two parts separately (Parts III and IV), in which we made the following contributions to fulfill our goals:

1. *A hybrid composition approach tool support for* `JavaScript` *product lines engineering* (RiPLE-HC– Chapter 4).
   RiPLE-HC relies on FeatureIDE capabilities to support software engineers who eventually decide to engineer a `JavaScript` product line. At this point, the RiPLE-HC workbench together with the external plug-ins works in conjunction providing several features: feature modeling; preprocessing annotations and code composition accordingly to a given variant configuration; annotations scattering location support

through a tree visualization; and feature collaboration graph visualization to each existing variant configuration. In addition, it can be extended with functionality due to its Eclipse plug-in architecture. Thus, we could identify more requirements by literature survey or by asking software engineers what functionality they need.

2. *Evaluation of* RiPLE-HC *with empirical case studies* (Chapter 5).
   We conducted two *proof-of-concept* empirical studies evaluating the proposed approach considering both, industry and academia settings. The industry case study showed considerable reduction of the time needed to built a learning objects product line for K-12 education. On the other hand, the academic empirical study carried out with free open-source `JavaScript` systems showed the ability to handle the implementation of crosscutting features with annotations and the scalability of the approach to systems sized up to 122KLOC.

3. *Extended empirical evidence of how feature-oriented software development affects program comprehension* (Chapters 7, 8, and 9).
   We presented a family of empirical studies focused on program comprehension tasks in feature-oriented software. In Chapter 6, we introduced each of the four studies and presented the variations in the experimental setup among them. Next, we enumerate the main findings in each of them.

   VICC1 (Chapter 7) addressed the impact on feature location while using either the Standard or RiPLE-HC. The study recruited undergraduate students and showed no statistical difference among the groups using each approach regarding `response time` and `correctness` of their answers.

   VICC2 and VICC3 (Chapter 8) addressed the lack of empirical evidence on the difference of the influence of traditional (Conditional Compilation) and emerging (FeatureHouse) variability representations. To the best of our knowledge, there was only one pilot study carried out so far addressing this particular context of such an important aspect in feature-oriented software.

   Besides, this was the first exploratory study on the influence of confounding parameters on feature-oriented software comprehension. To this end, we explored in different directions, such as *(i)* trying to identify the differences imposed by such variability representations and *(ii)* trying to establish a relationship among confounding parameters while maintaining software using either FeatureHouse or Conditional Compilation.

   The results of our study allowed us to make the following conclusions:

   - There is evidence of the equivalence regarding the *(i)* demanded effort to understand, *(ii)* to find flawed source-code, and *(iii)* the time demanded to finish bug-finding maintenance tasks while using FeatureHouse and Conditional Compilation paradigms;
   - There is evidence of a perceived high influence of the following confounding parameters in the program comprehension and maintenance of software devel-

oped using either FeatureHouse or Conditional Compilation: *Familiarity with the study object*; *Reading Time*; *Fatigue*; *Treatment preference*; *Programming language knowledge*; and *Experience in the programming paradigm*.

- There is evidence of a different perceived influence of the confounding parameter formal *education* on the comprehension and maintenance of software developed using either FeatureHouse or Conditional Compilation.
- There is a moderate correlation between the motivation of software engineers and their efficiency to perform comprehension tasks correctly.

VICC4 (Chapter 9) used a qualitative method called focus group to identify potential aspects to produce initial insights and guide future research on the program comprehension of software in the presence of variability. More specifically, to compare and understand the differences and implications of the use of the variability representations such as those used in our studies, Conditional Compilation and FeatureHouse.

We can group our main findings in four main categories regarding the aspects that might have influence on the comprehension and consequently the maintenance of feature-oriented software. These groups are enumerated in the following and may point out the direction of the next steps of the research community.

Comprehension Strategies: by understanding how the software engineers address the comprehension of unfamiliar code can produce insights on the construction of more effective methods, processes, and tools to support maintenance of feature-oriented software. Among our findings in this category are the use tools other than the search to address unfamiliar code. We conjecture that visualization tools, such as the "Collaboration Diagram" and the "Configuration management" can contribute to the comprehension code using both Conditional Compilation and FeatureHouse.

Hindering Factors: by understanding what makes the comprehension tasks harder, we can build tools to facilitate such an important process in the software maintenance. Among our findings in this category are already well known [6] by the practitioners using annotations, the excessive amount of annotations and the highly scattering of them. Regarding the composition approach, we found the number of duplicated classes, as well as the amount of clicks to reach the source code as bottlenecks of the existing code organization tools and are worth further investigation and improvements.

Facilitators Factors: by understanding what makes the comprehension tasks easier, we can concentrate the effort on research to make these factors of some use to enhance the already available tools to comprehend software. Among our findings in this category are the simple programming model of Conditional Compilation and the good way of code organization of FeatureHouse. This finding corroborates with those in the hindering factors category, since although the participants like the way the code is organized, they were uncomfortable with the amount of clicks to get to it.

General Observations: by understanding the feelings of the software engineers regarding to the first contact with unfamiliar variability representations, we can also look for improvements in such aspects causing negative and lack of motivated of software engineers facing the decision of using one or another option. Among our findings in this category is the difficulty to novice developers using FeatureHouse to perceive the importance of the precedence among the features in the binding time. This fact should be more explicitly addressed in supporting tools. Additionally, the participants pointed out the traceability is an important asset in the comprehension of such kind of code, which we agree and suggest also further investigation in the facet.

4. *Reusable experimental designs.*
   All of our experimental settings were designed to be reusable. We used common guidelines to present our set up, and made all material available at the project website (<http://rise.com.br/riselabs/vicc/>).

Such contributions set new direction for new and complimentary research in the field. We discuss the limitations and future work next.

10.2 LIMITATIONS

We put a lot of effort in the planning and performing each of the presented studies. Although we pursued the best experimental setup in each one, we know some edges remained uncovered, in other words, we are aware of different aspects that limit our findings and are worth mentioning.

- **The programming language switch**. We believe the use of two different programming languages in our experimental studies can play both roles, strengthen the findings or threat the general conclusions. Specially, if we consider the differences in sintax and the peculiarities of each language.

- **The limited number of addressed maintenance tasks**. This is definetly a limitation, still it would be necessary several studies to cover each or at least most of these kind of tasks. A good point in favor of the generalization of our findings would be checking whether they are recurrent in different maintenance tasks.

- **The expressiveness of annotations in `Java` and `JavaScript`**. Some may question the expressiveness of the Conditional Compilation emulation in these languages. It is true the use of annotations through comments in code are rather limited if compared to the native `C/C++` annotations. Thus, the influence of the *"emulated"* annotations in our results might not reflect the influence of *native* annotations for different variability representation.

- **The discipline of the annotations**. Due to the limation of the expressiveness of the annotations emulation in `Java` and `JavaScript`, our experimental setup covered only disciplined annotations. This point should also be further investigated in future research.

10.3 FUTURE WORK

We are aware that these findings are far from describing the relationship among variability representation chose and the different confounding parameters associated to comprehension tasks in software engineering. As our studies were limited as discussed in the previous section, there is a number of other research directions that remain open. Thus, we enumerate a set of directions to further research in this topic.

RiPLE-HC tool support improvement – It is possible to enhance the current implementation of the RiPLE-HC tool support, such as the filtering the scattering graphs and to integrate the workbench to use the recent improvements of FeatureIDE [87]. There is a need to improve the tool support to better handle nested annotated blocks and to provide consistency checking of the annotations against the feature model constraints [7].

Empirical evidence on program comprehension feature-oriented software – It is also important to address other maintenance activities, such as additive and perfective maintenance in the software components. Another possible direction is the comparison of the influence of different tools supporting program comprehension in the different variability representations, such as collaboration diagram and the FeatureIDE Outline [88]. Additionally, it would be interesting to observe professional programmers working with code using Conditional Compilation and/or Composition to observe how they work with different variability mechanisms and compare with the students results.

Case Studies/Survey – Additional case studies and/or surveys with developers to better investigate the synergy of the hybrid composition with the dynamic nature of `JavaScript`, such as global scope, function redefinition, weakly typing, as well as the usage of different current available frameworks such *angular* and *react* are possible directions to further investigation. Furthermore, it is also possible to assess the fine-grained feature dependencies [89] impact on the comprehension of feature-oriented software systems.

REFERENCES

1 APEL, S.; KäSTNER, C.; LENGAUER, C. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, v. 39, n. 1, p. 63–79, 2013.

2 APEL, S.; LEICH, T.; ROSENMüLLER, M.; SAAKE, G. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In: *Proceedings of the 4th Generative Programming and Component Engineering*. Berlin, Germany: Springer-Verlag, 2005. p. 125–140. ISBN 978-3-540-29138-1.

3 BATORY, D. S. Feature-oriented programming and the AHEAD tool suite. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. p. 702–703. ISBN 0-7695-2163-0.

4 KäSTNER, C.; APEL, S. Integrating compositional and annotative approaches for product line engineering. In: *Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*. Nashville, Tennessee: Universität Passau, 2008. p. 1 – 6.

5 SPENCER, H.; COLLYER, G. #ifdef considered harmful, or portability experience with C news. In: *Proceedings of the USENIX Summer 1992 Technical Conference*. San Antonio, TX: USENIX, 1992. p. 185–197.

6 MEDEIROS, F.; KÄSTNER, C.; RIBEIRO, M.; NADI, S.; GHEYI, R. The love/hate relationship with the C preprocessor: An interview study. In: *Proceedings of the 29th European Conference on Object-Oriented Programming*. Prague, Czech Republic: Dagstuhl Publishing, 2015. p. 495–518.

7 SANTOS, A. R.; ALMEIDA, E. S. Do #ifdef-based variation points realize feature model constraints? *Software Engineering Notes*, ACM, New York, NY, USA, v. 40, n. 6, p. 1–5, 2015.

8 APEL, S.; KäSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, jul 2009. ISSN 1660-1769.

9 BATORY, D. Feature models, grammars, and propositional formulas. In: *Proceedings of the 9th International Software Product Lines Conference*. Berlin, Germany: Spring-Verlag, 2005. p. 7–20.

10 APEL, S.; BATORY, D.; KäSTNER, C.; SAAKE, G. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Germany: Springer-Verlag, 2013. ISBN 978-3-642-37521-7.

11 SIEGMUND, J. *Framework for Measuring Program Comprehension.* Tese (Doutorado) — Otto-von-Guericke-Universität Magdeburg, 2012.

12 SILVA, L.; RAMOS, M.; VALENTE, M. T.; BERGEL, A.; ANQUETIL, N. Does javascript software embrace classes? In: *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering.* Québec, Canada: IEEE, 2015. p. 73–82.

13 KITCHENHAM, B.; CHARTERS, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering.* [S.l.], 2007.

14 KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, P. *Evidence-Based Software Engineering and Systematic Reviews.* [S.l.]: Chapman and Hall/CRC, 2015. 399 p. ISBN 9781482228656.

15 SIEGMUND, J.; SCHUMANN, J. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, Springer-Verlag, v. 20, n. 4, p. 1159–1192, 2014.

16 SIEGMUND, J. Program comprehension: Past, present, and future. In: *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* Washington, DC, USA: IEEE Computer Society, 2016. v. 5, p. 13–20.

17 WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering.* New York, NY, USA: ACM, 2014. p. 38:1–38:10. ISBN 978-1-4503-2476-2. Disponível em: <http://doi.acm.org/10.1145/2601248.2601268>.

18 WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. *Experimentation in Software Engineering.* Berlin, Germany: Springer-Verlag, 2012. ISBN 978-3-642-29044-2.

19 SHULL, F.; SINGER, J.; SJØBERG, D. I. *Guide to Advanced Empirical Software Engineering.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN 184800043X.

20 MACHADO, I. C.; SANTOS, A. R.; CAVALCANTI, Y. a. C.; TRZAN, E. G.; SOUZA, M. M. a. de; ALMEIDA, E. S. Low-level variability support for web-based software product lines. In: *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems.* New York, NY, USA: ACM, 2014. p. 15:1–15:8. ISBN 978-1-4503-2556-1.

21 SANTOS, A. R.; MACHADO, I. C.; ALMEIDA, E. S. RiPLE-HC: Javascript systems meets SPL composition. In: *Proceedings of the 20th International Systems and Software Product Line Conference.* New York, NY, USA: ACM, 2016. p. 154–163. ISBN 978-1-4503-4050-2.

22 SANTOS, A. R.; MACHADO, I. C.; ALMEIDA, E. S. RiPLE-HC: Visual support for features scattering and interactions. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. New York, NY, USA: ACM, 2016. p. 320–323.

23 SANTOS, A. R.; MACHADO, I. C.; ALMEIDA, E. S. Aspects influencing feature-oriented software comprehension: Observations from a focus group. In: *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse*. New York, NY, USA: ACM, 2017. p. 1–10.

24 SANTOS, A. R.; MACHADO, I. C.; ALMEIDA, E. S.; SIEGMUND, J.; APEL, S. Exploring the influence of variability representations on program comprehension. *Empirical Software Engineering*, Springer US, x, n. x, p. 1–34, 2018. ISSN 1382-3256.

25 SANTOS, A. R.; OLIVEIRA, R. P.; ALMEIDA, E. S. Strategies for consistency checking on software product lines: A mapping study. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: ACM, 2015. p. 5:1–5:14. ISBN 978-1-4503-3350-4.

26 FIGUEIREDO, E.; CACHO, N.; SANT'ANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; FILHO, F. C.; DANTAS, F. Evolving software product lines with aspects: An empirical study on design stability. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008. p. 261–270. ISBN 978-1-60558-079-1.

27 LIEBIG, J.; KäSTNER, C.; APEL, S. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: *Proceedings of the 10th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2011. p. 191–202. ISBN 978-1-4503-0605-8.

28 SIEGMUND, J.; KäSTNER, C.; LIEBIG, J.; APEL, S. Comparing program comprehension of physically and virtually separated concerns. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012. p. 17–24. ISBN 978-1-4503-1309-4.

29 PREHOFER, C. Feature-oriented programming: A fresh look at objects. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Berlin, Germany: Springer-Verlag, 1997. p. 419–443. ISBN 978-3-540-63089-0.

30 BATORY, D. S.; SARVELA, J. N.; RAUSCHMAYER, A. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, v. 30, n. 6, p. 355–371, 2004.

31 KäSTNER, C.; APEL, S.; KUHLEMANN, M. A model of refactoring physically and virtually separated features. In: *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2009. p. 157–166. ISBN 978-1-60558-494-2.

32 KOENEMANN, J.; ROBERTSON, S. P. Expert problem solving strategies for program comprehension. In: *Proceedings of the 3rd Conference on Human Factors in Computing Systems.* New York, NY, USA: ACM, 1991. p. 125–130. ISBN 0-89791-383-3.

33 BROOKS, R. Using a behavioral theory of program comprehension in software engineering. In: *Proceedings of the 3rd International Conference on Software Engineering.* Piscataway, NJ, USA: IEEE Press, 1978. p. 196–201.

34 SHAFT, T. M.; VESSEY, I. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *Journal of Management Information Systems*, v. 15, n. 1, p. 51–78, 1998.

35 SOLOWAY, E.; EHRLICH, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, n. 5, p. 595–609, Sept 1984. ISSN 0098-5589.

36 PENNINGTON, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, v. 19, n. 3, p. 295 – 341, 1987. ISSN 0010-0285.

37 SHNEIDERMAN, B.; MAYER, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, v. 8, n. 3, p. 219–238, 1979. ISSN 1573-7640.

38 MAYRHAUSER, A. von; VANS, A. M. From program comprehension to tool requirements for an industrial environment. In: *Proceedings of the 2nd Workshop on Program Comprehension.* Washington, DC, USA: IEEE Computer Society, 1993. p. 78–86. ISSN 1092-8138.

39 MAALEJ, W.; TIARKS, R.; ROEHM, T.; KOSCHKE, R. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, New York, NY, USA, v. 23, n. 4, p. 31:1–31:37, set. 2014. ISSN 1049-331X.

40 FEIGENSPAN, J.; APEL, S.; LIEBIG, J.; KASTNER, C. Exploring software measures to assess program comprehension. In: *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement.* Washington, DC, USA: IEEE Computer Society, 2011. p. 127–136. ISSN 1949-3770.

41 SIEGMUND, J.; KäSTNER, C.; LIEBIG, J.; APEL, S.; HANENBERG, S. Measuring and modeling programming experience. *Empirical Software Engineering*, Springer US, v. 19, n. 5, p. 1299–1334, 2014. ISSN 1382-3256.

42 FEIGENSPAN, J.; SIEGMUND, N.; HASSELBERG, A.; KöPPEN, M. PROPHET: Tool infrastructure to support program comprehension experiments. In: *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement.* Washington, DC, USA: IEEE Computer Society, 2011. Poster Session.

43 FEIGENSPAN, J.; SIEGMUND, N. Supporting comprehension experiments with human subjects. In: *Proceedings of the 20th International Conference on Program Comprehension*. Passau, Germany: IEEE, 2012. p. 244–246. ISBN 1092-8138.

44 MAYRHAUSER, A. V.; VANS, A. M. Program comprehension during software maintenance and evolution. *Computer*, IEEE, Washington, DC, USA, v. 28, n. 8, p. 44–55, 1995.

45 STOREY, M.-A. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, v. 14, n. 3, p. 187–208, 2006. ISSN 1573-1367.

46 RAJLICH, V.; WILDE, N. The role of concepts in program comprehension. In: *Proceedings of the 10th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2002. p. 271–278.

47 BURKHARDT, J.-m.; DÉTIENNE, F.; WIEDENBECK, S. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, v. 7, n. 2, p. 115–156, 2002. ISSN 1382-3256.

48 PENTA, M. D.; STIREWALT, R. E. K.; KRAEMER, E. Designing your next empirical study on program comprehension. In: *Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2007. p. 281–285. ISSN 1092-8138.

49 SIEGMUND, J.; KÄSTNER, C.; APEL, S.; PARNIN, C.; BETHMANN, A.; LEICH, T.; SAAKE, G.; BRECHMANN, A. Understanding understanding source code with functional magnetic resonance imaging. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. p. 378–389.

50 KOSAR, T.; MERNIK, M.; CARVER, J. C. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, v. 17, n. 3, p. 276–304, 2012. ISSN 1573-7616.

51 FEIGENSPAN, J.; KäSTNER, C.; APEL, S.; LIEBIG, J.; SCHULZE, M.; DACHSELT, R.; PAPENDIECK, M.; LEICH, T.; SAAKE, G. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, Springer US, v. 18, n. 4, p. 699–745, 2013. ISSN 1382-3256.

52 SHULL, F. J.; CARVER, J. C.; VEGAS, S.; JURISTO, N. The role of replications in empirical software engineering. *Empirical Software Engineering*, Springer, Berlin, Germany, v. 13, n. 2, p. 211–218, 2008. ISSN 1573-7616.

53 ROBILLARD, M. P.; COELHO, W.; MURPHY, G. C. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, v. 30, n. 12, p. 889–903, 2004.

54 FRITZ, T.; BEGEL, A.; MüLLER, S. C.; YIGIT-ELLIOTT, S.; ZüGER, M. Using psycho-physiological measures to assess task difficulty in software development. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. p. 402–413. ISBN 978-1-4503-2756-5.

55 MELO, J.; BRABRAND, C.; Wąsowski, A. How does the degree of variability affect bug finding? In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. p. 679–690. ISBN 978-1-4503-3900-1.

56 ALIMADADI, S.; MESBAH, A.; PATTABIRAMAN, K. Understanding asynchronous interactions in full-stack javascript. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. p. 1169–1180. ISBN 978-1-4503-3900-1.

57 GALSTER, M.; WEYNS, D.; TOFAN, D.; MICHALIK, B.; AVGERIOU, P. Variability in software systems: A systematic literature review. *IEEE Transactions on Software Engineering*, v. 40, n. 3, p. 282–306, 2014.

58 PETTERSSON, U.; JARZABEK, S. Industrial experience with building a web portal product line using a lightweight, reactive approach. *Software Engineering Notes*, ACM, New York, NY, USA, v. 30, n. 5, p. 326–335, 2005. ISSN 0163-5948.

59 CAPILLA, R.; DUENAS, J. Light-weight product-lines for evolution and maintenance of web sites. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2003. p. 53–62. ISSN 1534-5351.

60 TRUJILLO, S.; BATORY, D.; DIAZ, O. Feature oriented model driven development: A case study for portlets. In: *Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. p. 44–53. ISBN 0-7695-2828-7.

61 FERREIRA, G. C. S.; GAIA, F. N.; FIGUEIREDO, E.; MAIA, M. A. On the use of feature-oriented programming for evolving software product lines - a comparative study. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, v. 93, p. 65–85, nov. 2014. ISSN 0167-6423.

62 FISCHER, S.; LINSBAUER, L.; LOPEZ-HERREJON, R. E.; EGYED, A. Enhancing clone-and-own with systematic reuse for developing software variants. In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE Computer Society, 2014. p. 391–400.

63 GAIA, F. N.; FERREIRA, G. C. S.; FIGUEIREDO, E.; MAIA, M. A. A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, v. 96, n. P2, p. 230–253, dez. 2014. ISSN 0167-6423.

64 LIU, J.; BATORY, D.; LENGAUER, C. Feature oriented refactoring of legacy applications. In: *Proceedings of the 28th International Conference on Software engineering*. New York, NY, USA: ACM, 2006. p. 112–121. ISBN 1-59593-375-1.

65 PIERCE, B. C. *Basic category theory for computer scientists*. Cambridge, Massachusetts, United States: MIT press, 1991.

66 THüM, T.; KäSTNER, C.; BENDUHN, F.; MEINICKE, J.; SAAKE, G.; LEICH, T. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, v. 79, p. 70–85, 2014. ISSN 0167-6423.

67 CUTSEM, T. V.; MILLER, M. S. Robust trait composition for javascript. *Science of Computer Programming*, 2012. ISSN 0167-6423. In Press, Corrected Proof.

68 QUEIROZ, R.; PASSOS, L.; VALENTE, M. T.; HUNSEN, C.; APEL, S.; CZARNECKI, K. The shape of feature code: An analysis of twenty C-preprocessor-based systems. *Journal of Software and Systems Modeling*, Springer-Verlag, p. 1–29, 2015.

69 BASILI, V.; ROMBACH, H. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, IEEE, v. 14, p. 758–773, Jun. 1988. ISSN 0098-5589.

70 COMMITTEE, L. T. S. *IEEE Standard for Learning Object Metadata*. Washington, DC, USA, 2002.

71 WILEY, D. A. Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. In: *The Instructional Use of Learning Objects*. Bloomington, IN, USA: Association for Educational Communications and Technology, 2000.

72 KRUEGER, C. W. Easing the transition to software mass customization. In: *Proceedings of the 4th International Workshop on Software Product-Family Engineering*. Berlin, Germany: Springer-Verlag, 2001. p. 282–293. ISBN 3-540-43659-6.

73 WEISS, R. C. T. L. D. M. *Software product-line engineering: a family-based software development process*. Boston, Massachusetts, United States: Addison-Wesley, 1999.

74 POHL, K.; BöCKLE, G.; LINDEN, F. J. van der. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer-Verlag, 2005. ISBN 3540243720.

75 MARCUS, A.; SERGEYEV, A.; RAJLICH, V.; MALETIC, J. I. An information retrieval approach to concept location in source code. In: *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. p. 214–223.

76 SIEGMUND, J.; BRECHMANN, A.; APEL, S.; KäSTNER, C.; LIEBIG, J.; LEICH, T.; SAAKE, G. Toward measuring program comprehension with functional magnetic

resonance imaging. In: *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012. p. 24:1–24:4. ISBN 978-1-4503-1614-9.

77 NETO, P. A. M. S.; SANTANA, T. L.; ALMEIDA, E. S.; CAVALCANTI, Y. a. C. RiSE Events: A testbed for software product lines experimentation. In: *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*. New York, NY, USA: ACM, 2016. p. 12–13. ISBN 978-1-4503-4176-9.

78 SEAMAN, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, IEEE, v. 25, n. 4, p. 557–572, 1999.

79 SALMAN, I.; MISIRLI, A. T.; JURISTO, N. Are students representatives of professionals in software engineering experiments? In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. p. 666–676. ISBN 978-1-4799-1934-5.

80 COMMITTEE, S. . S. E. S. *14764-2006 - ISO/IEC International Standard for Software Engineering - Software Life Cycle Processes - Maintenance*. Washington, DC, USA, 2006. 1-46 p.

81 MEYER, B. *Object-oriented software construction*. Upper Saddle River, New Jersey, United States: Prentice Hall, 1988. v. 2.

82 FEIGENSPAN, J.; SIEGMUND, N.; HASSELBERG, A.; KöPPEN, M. PROPHET: Tool infrastructure to support program comprehension experiments. In: *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2011. Poster Session.

83 RENSIS, L. A technique for the measurement of attitudes. *Archives of Psychology*, v. 22, n. 140, p. 5–55, 1932.

84 MALAQUIAS, R.; RIBEIRO, M.; BONIFáCIO, R.; MONTEIRO, E.; MEDEIROS, F.; GARCIA, A.; GHEYI, R. The discipline of preprocessor-based annotations - does #ifdef tag n't #endif matter. In: *Proceedings of the 25th International Conference on Program Comprehension*. [S.l.: s.n.], 2017. (ICPC'17), p. 297–307.

85 SANTOS, J. A. M.; MENDONçA, M. G. Exploring decision drivers on god class detection in three controlled experiments. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2015. (SAC '15), p. 1472–1479. ISBN 978-1-4503-3196-8.

86 EVERITT, B. S.; SKRONDAL, A. *The Cambridge dictionary of statistics*. 4. ed. Cambridge: Cambridge University Press, 2010. ISBN 9780521766999.

87 MEINICKE, J.; THüM, T.; SCHRöTER, R.; KRIETER, S.; BENDUHN, F.; SAAKE, G.; LEICH, T. Featureide: Taming the preprocessor wilderness. In: *Proceedings of the 38th*

International Conference on Software Engineering Companion. New York, NY, USA: ACM, 2016. p. 629–632. ISBN 978-1-4503-4205-6.

88 LEICH, T.; APEL, S.; MARNITZ, L.; SAAKE, G. Tool support for feature-oriented software development: featureide: an eclipse-based approach. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. San Diego, California: ACM, 2005. p. 55–59. ISBN 1-59593-342-5.

89 RODRIGUES, I.; RIBEIRO, M.; MEDEIROS, F.; BORBA, P.; FONSECA, B.; GHEYI, R. Assessing fine-grained feature dependencies. *Information and Software Technology*, Elsevier, v. 78, p. 27–52, 2016.

90 ANDERSON, T. W.; FINN, J. *The new statistical analysis of data*. New York City, New York, United States: Springer, 1996.

91 ANDERSON, T. W.; RUBIN, H. Statistical inference in factor analysis. In: *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 5: Contributions to Econometrics, Industrial Research, and Psychometry*. Berkeley, Calif.: University of California Press, 1956. p. 111–150.

92 COSTELLO, A. B.; OSBORNE, J. W. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research & Evaluation*, v. 10, p. 173–178, 2005.

Appendix A

Standing in the shoulder of giants.

LITERATURE VENUES

A.1 DBLP VENUES

Table A.1 describes each of the selected forums considered in the for the manual literature review of the published papers from 2011 until June 2016. The search was conducted completelly in the DBLP[1] library. We selected ESE as leading platform for empirical research in the field of software engineering. We included TOSEM and TSE as leading journals in software engineering. ICPC is the leading conference for research regarding program comprehension. ICSE and FSE are the leading conferences on software engineering. Additionally, we chose ESEM as platform in the empirical-software-engineering domain. From these venues, we selected all papers that reported evidence-based studies concerning program compreension on feature-oriented configurable problems.

A.2 SIEGMUND AND SCHUMANN SURVEY VENUES

Table A.2 describes each of the selected forums considered in the Siegmund and Schumann survey [15].

[1]The computer science library, available at <http://dblp.dagstuhl.de>.

Table A.1 Selected research forums from DBLP.

Venue	**Acronym**
JOURNALS	
Empirical Software Engineering	ESE
Transactions on Software Engineering	TSE
Transactions on Software Engineering and Methodology	TOSEM
CONFERENCES	
Software Product Lines Conference	SPLC
International Conference on Software Reuse	ICSR
International Conference on Generative Programming	GPCE
International Conference on Program Comprehension	ICPC
International Conference on Software Engineering	ICSE
SYMPOSIUMS	
Symposium on the Foundations of Software Engineering	FSE
International Symposium on Empirical Software Engineering and Measurement	ESEM
WORKSHOPS	
International Workshop on Variability Modelling of Software-intensive Systems	VAMOS
International Workshop on Product LinE Approaches in Software Engineering	PLEASE
International Workshop on Feature Oriented Software Development	FOSD

Table A.2 Selected research forums from Siegmund and Schumann survey [15].

Venue	**Acronym**
JOURNALS	
Empirical Software Engineering	ESE
Journal of Software: Evolution and Process	JSEP
Transactions on Software Engineering	TSE
Transactions on Software Engineering and Methodology	TOSEM
CONFERENCES	
Software Product Lines Conference	SPLC
International Conference on Software Reuse	ICSR
International Conference on Generative Programming	GPCE
International Conference on Program Comprehension	ICPC
International Conference on Software Engineering	ICSE
Working Conference on Reverse Engineering	WCRE
International Conference on Human-computer Interaction	CHI
International Conference on Software Maintenance and Evolution	ICSM
SYMPOSIUMS	
Symposium on the Foundations of Software Engineering	FSE
International Symposium on Empirical Software Engineering and Measurement	ESEM
Symposium on Visual Languages and Human-Centric Computing	VLHCC
Symposium on Human-Centric Computing Languages and Environments	HCC[2]
WORKSHOPS	
International Workshop on Program Comprehension	IWPC[3]
International Workshop on Cooperative and Human Aspects of Software Engineering	CHASE
International Workshop on Variability Modelling of Software-intensive Systems	VAMOS
International Workshop on Product LinE Approaches in Software Engineering	PLEASE

Appendix

B

CHARACTERIZATION QUESTIONNAIRE

Table B.1 Questionnaire for measuring the programming experience of the participants. Extracted from Siegmund's *et al.* work [41]

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 paradigm?
Q12	How experienced are you with the Imperative programming paradigm?
Q13	How experienced are you with the Objected-oriented programming paradigm?
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?

Appendix C

VICC1 FEEDBACK FORM

1. Which approach you find best for Javascript development? (STANDARD or RIPLE-HC)
2. Which are the points in favor of the STANDARD way of development?
3. Which are the points against of the STANDARD way of development?
4. Which are the points in favor of the RIPLE-HC way of development?
5. Which are the points against of the RIPLE-HC way of development?
6. Do you think RIPLE-HC can be generalized for other programming languages? Justify your answer

Appendix

D

MODELING PROGRAMMING EXPERIENCE

This appendix presents the replication of the Siegmund *et al.* [41] effort to model programming experience. In discussion with Janet Siegmund, she advised to left it out of this discussion due the size of the sample, as well as out of the scope of this study. As the work was done, we decided to include it in the Appendix D.

D.1 CORRELATIONS ANALYSIS

Siegmund *et al.* [41] 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 D.1 shows the variables actually used to measure their experience.

Table D.2 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 D.1). 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 [90]. 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.

Table D.1 Description of the variables used for measuring the participants' programming experience. Extracted from Siegmund's *et al.* work [41].

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-take`	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	`professional`	Programming experience self-assessment against programmer with 20 years of experience.	LikertB

ID: refers to the question identifier (Appendix B). **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.$\}$

Table D.2 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 ($p < .05$).

D.2 FACTOR ANALYSIS

We are aligned with Siegmund *et al.* [41] 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* [91] 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 D.3 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 [92].

The first factor of our analysis summarizes the variables `oo`, `java`, `other-l`, `large-proj`, `proj-size`, and `students`. This 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 the 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.* [41].

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.* [41].

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.* [41] model this variable appeared together with the programming years – our second factor. Indeed these variables are related, the fact were grouped separately here might be explained by the differences in the size of the sample.

Table D.3 Factor analysis of each characterization independent variable of the characterization.

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.

(in Portuguese)

VICC4 - FOCUS GROUP TRANSCRIPTION

0’00”

[Researcher] O início do focus group, então, está começando às 10:40h.

Eu tenho um conjunto de questões que desenvolvi e a gente vai discutir, mais no nível de conversa mesmo.

Eu queria ouvir de cada um, queria que vocês pudessem falar razoavelmente alto para que eu pudesse conseguir gravar razoavelmente bem, aqui.

Aí eu vou querer resposta individual sem ser em ordem pré-estabelecida. Quem quiser, levanta a mão e fala, sobre a experiência que vocês tiveram com as tarefas – com a análise do código, tanto com compilação condicional, quanto com FeatureHouse.

Todo mundo entendeu como vai ser o esquema, né? Eu vou fazer uma pergunta, quem quiser levanta a mão e fala, de preferência, a maioria falando. E sem avaliação, é exatamente o que você percebeu, o que você sentiu sobre a coisa.

A primeira pergunta: eu gostaria que vocês dissessem, por exemplo, qual foi as suas primeiras impressões com cada um dos paradigmas. Vocês podem falar em ordem não estabelecida, quem gostaria de começar?

[Focus Group Participant] O paradigma da compilação condicional é interessante quando você tem muita variação dentro de um arquivo só. Aí você consegue enxergar de maneira bem clara, onde estão os pontos de variação. Só que quando isso começa a espalhar, aí já a abordagem do FeatureHouse pareceu interessante, porque você enxerga direitinho, né, a estrutura do projeto. Só que, a parte ruim é que pra chegar lá tem muitos cliques. Tem lá “core”, tem “ui”, tem que ir clicando. Então, quando também o projeto vai ramificando muito, tem que clicar muito, mas em termos de organização visual ele pareceu bastante interessante.

[Researcher] Entendi.

[Focus Group Participant] E também tem a outra ferramenta que vai mostrando, por exemplo, tinha “Base”, e tinha onde ele tá implementado...

[Researcher] é o diagrama de colaboração!?

[Focus Group Participant] isso. Ali acho que dá pra você ter uma noção de como tá implementada aquela feature, no caso. Ajuda também. Agora no outro paradigma, tem um fator interessante, que você consegue dar um find lá e pesquisar pelas anotações. Então, você consegue também navegar. Então, no final das contas, se você souber usar essas duas formas você consegue localizar o nível de espalhamento. Então, basicamente, foi o que eu fiz, então, eu consegui encontrar. [(tirando algumas coisas, talvez)2' 50" está inaudível]

2' 53"

[Focus Group Participant] o paradigma da compilação condicional, na minha percepção, tem um grande problema, que você deixa o código muito sujo. Ele fica difícil de você entender o código, devido à quantidade de anotações que você tem. Já em comparação ao outro paradigma, ele fica [inaudível] essencialmente, com a parte funcional mesmo, e facilita a manutenção da aplicação.

[Focus Group Participant] nesse sentido, talvez, uma ferramenta de visualização de código, pra você conseguir compreender ali, [inaudível] pra mostrar, seja com cor, seja com identação, nesse código da compilação condicional, talvez seja interessante. Porque você olha no inicio do if e [inaudível] você tem que procurar, num tá identado, num tá nada. Isso acaba também que, mais uma coisa que você tem que procurar no sistema pra ver se algum outro [inaudível].

[Researcher] Entendi.

4' 00"

[Focus Group Participant] Agora acho que, nas tarefas que a gente fez, a ferramenta também ajudou. Pra mim, que já conhece também o FeatureHouse e não usou [inaudível] o eclipse também [inaudível]

[Researcher] Entendi.

[Focus Group Participant] Eu nunca usei e realmente eu tive, a segunda ferramenta, o segundo projeto, foi muito mais simples pra mim. Extremamente mais simples.

[Researcher] quando você fala que nunca usou, nunca usou o que? O eclipse?

[Focus Group Participant] não, nunca usei o eclipse pra... já usei o eclipse em nível profissional mesmo. E pra mim a segunda foi muito mais fácil. Porque eu demorei um pouquinho também pra ir lá no eclipse e buscar os ifdefs, eu também utilizei isso, e pra mim não foi tão simples. Mas no segundo, como já tem aquela arvorezinha já pra mim foi trivial entender ali.

[Researcher] Certo. Uma próxima questão, se vocês não tiverem mais nada à adicionar... Durante a execução das tarefas com o paradigma, seja compilação condicional, seja com o featurehouse... (acho que até alguém já levantou alguns pontos, mas talvez se a gente pudesse enumerar, assim, seria uma boa) O que que poderia ter auxiliado vocês a ser mais eficaz? Assim, na atividade. Vamo começar por compilação condicional, o que teria ajuda você a ser mais eficaz na tua tarefa?

[Focus Group Participant] uma ferramenta de visualização de software pra isso, ajudaria.

[Researcher] mais alguém queria pontuar alguma coisa?

5'27"

[Focus Group Participant] no meu caso, pra ambos os paradigmas, como eu não uso o eclipse, senti dificuldade no sentido dos recursos da ferramenta, pra poder buscar os arquivos. Saber onde que tá cada controlador. Isso, pra mim, gerou um pouco de dificuldade.

[Focus Group Participant] até a própria, automatização dessa busca, por meio também de uma ferramenta seria interessante.

[Researcher] e para FeatureHouse?

[Focus Group Participant] pra mim vale pros dois.

[Researcher] a ferramenta, no caso, de busca, pra os dois.

[Focus Group Participant] no caso, tava falando assim, dentro do eclipse né, eu fiquei sabendo que existe uma maneira de buscar lá fora, eu também não utilizo o eclipse, uso mais o netbeans.

[Focus Group Participant] eu falei aqui né? No começo.

[Focus Group Participant] não, eles tavam falando... você pode tá fazendo, tal e tal... aí o que que acontece? Uma maneira de você localizar é diferente de uma ferramenta pra outra. Aí você também fica acostumado a usar uma só, direto. Você termina tendo dificuldade. Uma coisa que também achei interessante no trabalho foi exatamente isso, né? Mas assim, você realmente colocou a informação à disposição no início, mas dentro do questionário em si, não tem nenhum campo pra identificar o conhecimento da ferramenta ou da linguagem [inaudível]. Isso também seria interessante, até pra conseguir medir o grau de conhecimento de cada um que tá participando do estudo, né? [inaudível] já conhece a ferramenta...

7' 17"

[Researcher] Assim, qual o sentimento de vocês na hora da tarefa, por exemplo, alguma das abordagens, a compilação conditional, a FeatureHouse, chegou a dizer: "Pô isso aqui tá me deixando muito cansado!" ou "Muita fadiga" ou "Eu prefiro esse a B, prefiro B a C". Vocês querem comentar um pouco sobre isso?

[Focus Group Participant] eu acho que pra mim, é unânime que a segunda abordagem é melhor que a [inaudível]. Eu acho que, eu pensando em manutenção daquele código ali, com aqueles ifdefs, pra mim seria uma dor de cabeça.

[Researcher] um sistema grande né?

[Focus Group Participant] um projeto grande... [inaudível]. Tanto nesse projeto em si acho que ia ser ruim [inaudível].

[Focus Group Participant] o código com ifdef fica muito sujo [inaudível]. Em contrapartida, se você vai usar o "Ctrl+F" lá pra buscar as coisas, ele simplifica. Você já vai direto ao ponto. Dá um find lá, ele lista todas visões que tem aquela anotação que você está esperando. Agora, a outra abordagem, quando você identifica pelo diagrama, aí você já vai no porto certo. Aí é mais rápido. Então, você precisa saber usar o recurso. Eu acho que é um equilíbrio, né?

[Focus Group Participant] outro ponto muito interessante pra mim da ferramenta, da segunda, é que você consegue entender se o modelo você tá gerando uma configuração ali, ele tá colocando como válida, aonde não é o que você queria. Por que você tem o visual, e você olha, "pô selecionei esta feature aqui e aqui tá liberado?" Então, se tem uma restrição que não tá funcionando. Você consegue ver muito mais fácil do com um ifdef. Você teria que gerar o produto, ver que deu erro. Coisas nesse sentido. Acho que isso daí é algo que melhora muito.

9'10"

[Focus Group Participant] ajuda. Ajuda na hora da configuração da linha de produto né!? Você consegue identificar as dependências de forma muito mais fácil do que com ifdef.

[Researcher] Acho que a gente já pontuou aqui bastante sobre ferramentas. Na hora que vocês estão tentando entender algum código que vocês, que é a primeira vez que vocês estão lhe dando com aquele código, que não é familiar pra vocês. Qual é a ferramenta ou estratégia que vocês utilizam e se vocês tiveram que mudar essa estratégia que vocês usam normalmente, pra utilizar, por exemplo, FeatureHouse, que é uma coisa mais nova pra vocês. Vocês tiveram que mudar essa estratégia ou vocês utilizaram a mesma estratégia que utilizam normalmente?

[Focus Group Participant] tem uma coisa. Esse código que você fez, ele tem algumas semelhanças com o que você aplicou no período passado. Então, acho que pra alguns participantes aqui, esse código não é totalmente novo, não. No outro experimento tinha algumas pessoas aqui, que já viram esse código antes. Porque parece que é o mesmo software, né? acho que já é um segundo contato com esse código-fonte.

tá. Mas e questão de estratégia? Assim...

[Focus Group Participant] eu acabei utilizando a mesma que utilizei no outro né... [inaudível]. Você fazer a busca lá. E a partir dessa busca fazer o rastreamento. Usei nas duas... [inaudível].

[Focus Group Participant] eu, depois que eu descobri que a segunda ferramenta tinha aquela arvorezinha, eu preferi criar um... desselecionar tudo... deselecionei todas as features que não eram obrigatórias, fui lá e verifiquei, selecionei as features que eu não queria, vi que gerava automaticamente, solucionei infinitamente mais rápido o meu problema.

[Focus Group Participant] eu, eu já mexi com alguma coisa de programação de jogos, então pra mim já foi automático procurar aquele init lá. Então, onde estivesse aquele init lá, tivesse uma variável inicializando a get, então pra mim aquele seria o lugar. Então eu não olhei feature, essas coisas assim, onde estava aquela declaração, eu fui atrás dela.

[Focus Group Participant] primeiro, eu também usei essa estratégia. [inaudível].

12'20"

[Researcher] Certo. A pior coisa que achei em compilação condicional foi?

[Focus Group Participant] o excesso de anotação. Eu acho o processo de anotação muito ineficaz.

[Researcher] ineficaz?

[Focus Group Participant] pra manutenção.

[Focus Group Participant] não é que ele seja ineficaz, porque ele funciona à muito tempo. Tem em muito lugar. Agora assim...

[Focus Group Participant] é isso. Veja em C né...o próprio kernel do Linux...

[Focus Group Participant] o problema é você dar manutenção nisso. Eu acho que é um caos, assim.

[Researcher] então eficaz, talvez ele seja, eficiente, talvez eficiente, não.

[particpantes] [inaudível]. Variável.

[Focus Group Participant] mas aí vocês tão falando do nível de espalhamento do concern, por exemplo, a feature A, o código que implementa ela tá espalhado ao longo de N partes no meu código.

[Focus Group Participant] é. aí se eu tiver que corrigir uma coisinha nisso daí, vou ter que ir em 300 classes. Talvez na abordagem... [inaudível].

[Focus Group Participant] mas assim... não é nem que você esteja errado. É que depende de como você modularizou o software na hora de construir e não o ifdef em si.

[Focus Group Participant] exatamente isso.

[Focus Group Participant] porque ele é utilizado aí, em vários artigo aí, o linux, o kernel, principalmente, como uma linha de produto enorme. Então, assim. Eu acho que talvez é a forma como você projetou aquela arquitetura pra ela ter o mínimo de difusão, de espalhamento lá do concern. Acho que é detalhe.

[Focus Group Participant] isso que gera um esforço a mais né?

[Focus Group Participant] se tiver coeso o código, eu acho que talvez não dê tanta complexidade. Porém você tem um código mais (não sei se a palavra correta é "sujo") por conta de anotações, que você tem que... agora também as ferramentas, algumas IDE's se você usar a pesquisa, ele lista, tipo de forma a você identificar toda aquela anotação. Se você dá o find por ela, ela vai listar todas ali, bem como os arquivos que elas estão. Se você está desenvolvendo em Java e você usa aquela prática de cada arquivo do Java só ter apenas uma classe, facilita a vida porque você vai ter todas ali e você sabe onde vai mecher. Agora se você não tem isso, você tem... como já vi alguns programas, que você abre um arquivo .java e tem 4, 5 classes dentro.

[Focus Group Participant] o que acontece ao utilizar a mesma tag, o mesmo nome lá... no comentário? Dá algum erro?

[Researcher] não, ali é só se você tiver a palavra reservada ifdef.

[Focus Group Participant] imagine que... a gente tá dando manutenção no código e o cara fala "olha... eu achei que essa feature aqui não é necessária." Você começa escrever alguns comentários lá utilizando... acaba tendo uma quantidade maior de informação, não sei... pode acabar confundindo um pouco. Até por esse processo de busca. Acho que é algo muito primário, assim... muito cru. Não tem como você fazer algo...

[Researcher] No caso do Java é porque não tem suporte a isso, né!? Isso foi meio adaptado. Mas pro C já tem a palavra reservada, já tem sintaxe na mesma natureza do código.

15' 53"

[Focus Group Participant] tem outra coisa que eu vi, talvez como o FeatureHouse talvez não tem como gerenciar é...

[Researcher] ...agora é compilação condicional, se guarda aí...

[Focus Group Participant] é da parte que... como o Glaydson tava falando, quando você tá trabalhando essa ideia de condicional, a gente não pode dizer que ela não tem o seu lugar, por que ela tem. Várias, a gente viu isso bastante. O problema mesmo é como a gente vai construir. Se você não tomar cuidado, você pode criar uma linha de replicações de código, que aí se torna inviável de qualquer maneira. Que o código... [inaudível]. você tem que tá fazendo as marcações. isso é um fato. E o código fica até, de certa forma, entre aspas “sujo”, né? que você tem que fazer as marcações. Mas no final das contas, mesmo no Java, como você disse, que é adaptado, não é uma coisa nativa ali. Mas o código não fica tão obscuro, pouco fácil de você entender o que ele tava sendo feito. Nesse caso! De repente, a gente pode tropeçar com um código que seja extremamente confuso. Ele falou, tem cara que coloca 4, 5 classes [inaudível] vai complicar bastante.

[Focus Group Participant] é... mas essa [inaudível] da compilação condicional, ela não te obriga a seguir features. Igual, por exemplo...

[Focus Group Participant] ela te obriga ou não te obriga?

[Focus Group Participant] compilação condicional não te obriga. Assim, como por exemplo, você pode... existem livros que falam que você pode programar orientado a objeto em uma linguagem estruturada, mas só que a própria linguagem não te força isso, isso não meio que natural. Acho que, o que fica meio que faltando é... o programador tem que tá ciente disso toda vez que ele tá implementando. Então, o gerenciamento é dele. Não tem nada da ferramenta que dá o suporte natural pra ele né!?

[Focus Group Participant] ou obrigue ele a seguir isso...

[Focus Group Participant] isso!

[Focus Group Participant] mas isso dá uma ideia, assim, ainda de uma coisa muito... uma adaptação muito inicial. É mais ou menos, como você programar Web, em Java, em Servlet. É muito baixo nível ali. Você gerar a sua html direto. É como se... pra mim a analogia correta é essa assim. Pra mim é como programar em Java utilizando servlet. É algo que funciona? Funciona. Se eu quiser fazer um sistema, vai funcionar. Mas, um sistema grande é inviável... trabalhar com servlet. No ifdef você tem técnicas de fazer como sua modelagem se adapte da melhor maneira a isso. Assim como tem gente que faz adaptação e trabalha com servlet.

[Researcher] e mudando agora para o FeatureHouse. A pior coisa que eu achei em FeatureHouse foi...

19’08”

[Focus Group Participant] acho que a replicação das classes, quando você tem diversas classes com o mesmo nome, acho que isso pode pode dificultar um pouco. [inaudível].

[Focus Group Participant] essa aplicativo era meio que um jogo, alguma coisa assim, né? aplicação de mídia.

[Researcher] uma aplicação de gerenciamento de mídia.

[Focus Group Participant] então, eu queria realmente ver, essa aplicação não tem problema né... a pessoa não tá trabalhando com gerenciamento de memória. Eu imagino se a pessoa tivesse que alocar e desalocar memória, com essas features todas espalhadas, ia ser um... [pedaço de inferno]. A pessoa ver onde ela alocou tal espaço, onde liberou. Nesse contexto, acho que seria muito complicado você fazer esse “rastreio” de memória.

[Focus Group Participant] a melhor coisa que eu vi em compilação condicional foi...

20’01”

[Focus Group Participant] visualização da variabilidade, da restrição, de várias configurações ali, pra mim

[Researcher] em compilação condicional...

[Focus Group Participant] ah! Eu acho que é mais simples de você [inaudível].

[Focus Group Participant] a pergunta seria mais assim... você tava falando sobre a replicação de features, ela é realmente necessária naquele modelo?

[Researcher] a gente tá falando de compilação condicional...

[Focus Group Participant] não, é isso. Porque assim... na compilação condicional, a anotação, ela já faz essa parte do trabalho, entendeu? Você consegue, lendo o código, você já vai entendendo ele todo. É diferente da outra, que é necessário compreender um pouquinho, o processo, pra você entender o que vai acontecer. A condicional não, ela tá lá dentro, [inaudível] você já consegue com uma certa experiência entender.

[Researcher] tá. Você tá querendo dizer que, à medida que você vai programando você já está vendo que a variabilidade vai acontecer ali. Enquanto que no outro lado, você tem que ver uma visão maior, uma big picture do projeto, pra conseguir ver a variabilidade acontecendo.

Interessante! A gente já falou do melhor ponto de compilação condicional e sobre o que foi a melhor coisa de FeatureHouse.

21'47"

[Focus Group Participant] visualizar as configurações válidas.

[Focus Group Participant] eu achei a legibilidade do código.

[Researcher] a legibilidade do código?

[Focus Group Participant] é... é mais limpo.

[Focus Group Participant] acho que como o rapaz tá dizendo, a legibilidade do código... é mais limpa.

[Focus Group Participant] mais direto ao ponto.

[Focus Group Participant] é. Preciso.

[Focus Group Participant] porque é a questão do tracking dos artefatos, né? Você tem o feature model e você tem o código e eles são correspondentes. Agora na compilação condicional não é direto, assim.

[Focus Group Participant] como já foi citado, nesse caso, a manutenção é direta. Se eu tenho que dar manutenção numa feature "X" lá, eu já vou direto nela. Mesmo que existam outros pontos que usem ela, mas eu só quero aquela feature li de fato, que eu vou dar manutenção. A gente tem ali, eu num preciso procurar, como se fosse na condicional, vendo onde ela tá sendo implementada num conjunto de classes. Tá lá, é justo o que eu quero, acabou. Ele é mais simples a manutenção.

[Researcher] quando eu tava usando condicional, a primeira coisa que eu fiz foi? Qual foi a primeira coisa que vocês fizeram? 23'06"

[Focus Group Participant] Ctrl+F (outros, concordam!)

[Researcher] buscar, local, global?

[Focus Group Participant] pesquisa no projeto, por todos os arquivos.

[Researcher] e FeatureHouse?

[Focus Group Participant] aí, no caso da FeatureHouse, eu fui no diagrama de colaboração mesmo, pra ver onde a feature estava sendo utilizada e procurei depois nas partes lá.

[Focus Group Participant] eu olhei o feature model pra saber se tinha alguma restrição àquela feature, depois fui no diagrama de colaboração, e aí dei um find, porque tinha me equivocado com a feature que eu tava procurando ou a linha lá do init. Foi esses três caminhos aí: o diagrama, o feature model e o find só pra ver se eu tava no lugar certo.

[Researcher] o colega de vocês falou aqui sobre manutenção. Vocês queriam elaborar mais sobre a dificuldade de manutenção entre uma abordagem e outra.

[Focus Group Participant] olha! Ao meu ver é importante, porque quando você pensa assim numa linha de produtos é claro que com o passar do tempo essa equipe muda. Então, a facilidade de manutenção dos paradigmas, pra mim, no meu ver, no dia-a-dia é importante. Tanto que eu citei o seguinte: se eu tiver que dar manutenção na feature de Photo, do lado, lá no FeatureIDE, vejo onde tá o pacote dela e já vou direto naquela feature e já dou manutenção. Quando a gente vai pro paradigma que usa anotação, eu tenho que dar um find pra saber, até porque eu não sei qual é o nível de espalhamento lá da feature nas minhas classes. Eu tenho que dar um find, saber onde é que tá todo implementado aquele concern, depois que eu entender, eu parto pra manutenção. Usando uma abordagem como a do FeatureIDE, eu já sei onde que tá aquela feature, eu vou direto nela e faço a manutenção que eu preciso. Então isso é mais direto ao ponto.

[Researcher] Você acha que isso, ao decorrer do tempo, isso vai trazer ganhos no desenvolvimento, no caso, essa rastreabilidade maior.

[Focus Group Participant] isso. Você já vai direto a linha da manutenção. Se você precisar fazer evolução naquela feature, você já vai de forma direcionada, você não precisa tá olhando o, a composição do concern inteiro, você só vai ali naquele pacote.

[Focus Group Participant] É a necessidade maior é a rastreabilidade.

[Focus Group Participant] porque uma coisa que me preocupa é o quanto é difundido este concern no seu código. Já que...

[Researcher] por difundido você quer dizer espalhado?

[Focus Group Participant] isso. Quanto mais... maior o espalhamento, maior é possibilidade de erro naquele lugar, o índice de erros naquele lugar. Essa é a minha visão. Eu vou direto naquele pacote, faço a correção e é mais simples. E isso, o nível de impacto é isolado. Eu só mexi ali. Qualquer coisa é mais fácil eu saber, se a minha correção inseriu erro, do que a outras correções que foram feitas.

26’ 24”

[Focus Group Participant] fora que ali também dá pra você ter a ideia de que você mexendo aqui, aonde mais... pode se tornar um problema. Se eu mexer aqui, pode acontecer um problema em tal local. No outro caso, onde você faz anotação, até isso você tem que ter anotado. A essa “Photo” aqui vai estar interagindo com tal feature. Então, você pra mexer aqui você vai ter sempre que estar prestando atenção se não vai gerar uma falha. Lá tem coisas que você não vai poder poder mexer de jeito nenhum. E nessa abordagem, você já consegue enxergar isso melhor. Ifdef já não, se você tem que ter o hábito de registrar tudo.

[Focus Group Participant] por exemplo, eu já participei de alguns projetos, que a gente tinha uma matriz de... uma planilha no Excel, que era a matriz de rastreabilidade. Se eu mexesse em tal funcionalidade, você tinha que olhar A, B, e C. Na marra, entendeu!? [inaudível] O cara dizia que não mexeu numa classe, quando vai olhar, você mexeu... você mexeu numa classe “X”, mas que essa classe tinha várias funcionalidades que um grupo de módulos utilizava. Na marra! Era a solução que nós encontramos pra contornar o problema, mas da forma que o FeatureIDE mostra o paradigma de [inaudível] acho muito mais

[Focus Group Participant] natural, né!?

[Focus Group Participant] natural.

27,'40"

[Researcher] aí você falou uma coisa interessante, que chama de "propagação de mudanças". Você chegou a dizer que no FeatureIDE isso talvez não ocorra, talvez isso não exista. Foi isso mesmo que você quis dizer? Eu entendi bem ou entendi mal?

[Focus Group Participant] era isso que eu tava falando. A minha visão inicial dela é que talvez... então, teria que ter um pouquinho mais de maturidade para puder afirmar se ocorre ou não. Mas pelo que a gente foi apresentado, não ocorre tanto quanto com a... [inaudível]

[Researcher] é... talvez... não sei se vocês perceberam, mas talvez a precedência das coisas seja um problema. É tanto que tem uma pergunta lá na tarefa. Por exemplo, se você está fazendo uma implementação e você resolve refinar aquele método muitas vezes, a ordem que você coloca a classe no código vai mudar totalmente a ordem que você tá programando. Isso talvez, a propagação não fique tão clara quanto... vocês perceberam isso ou não foi tão claro ao primeiro olhar? Vocês não chegaram a imaginar isso?

[Focus Group Participant] eu particularmente, não.

[Focus Group Participant] eu enxerguei assim sabe... tem alguns recursos ali que são compartilhados que se você não pensar muito bem na hora de programar usando a FeatureIDE, você pode ter alguns problemas. Por exemplo, você tem uma tela que você desenha as coisas, eu enxerguei isso, as vezes você pode ter desenhado alguma coisa e na outra feature, em outra parte da implementação você desenhou também, aí você dá um efeito que não quer. E eu acho que, talvez pra você resolver isso não fique tão direto assim. Você vai ter que pensar mais, aonde você pode ter criado algum desenho. Porque um recurso único, né!? que é a tela vai ser sempre a mesma. Não importa quantas features você tenha. Então, a pessoa, ela ganha a naturalidade na hora de mexer com as features, mas tem alguns recursos que são únicos, então tem que saber gerenciar isso.

[Focus Group Participant] mas isso você tá falando da metodologia de trabalho na manutenção?

[Focus Group Participant] Não, não. Eu tô falando assim... que você pode gerar resultados indesejados por, às vezes pensar de uma maneira tão espalhada, tão isolada. O que eu estou falando é que por exemplo, numa aplicação multimídia você tem uma tela. Aí você especializa várias vezes e aí depois coloca várias features, pode ser que tenha, sei lá... feature interaction, né!? e a pessoa tem que fazer as coisas mais espalhadas e isso pode ser mais difícil do que, por exemplo, se tivesse em compilação condicional você veria tudo no mesma tela, no mesmo código, é nessa tela, é nessa hora e esse aqui eu não desenho. Então, você vai ter 20, 30 features trabalhando...20 trechos de código trabalhando com a mesma tela...

[Researcher] acho que talvez o exemplo que ele dá... por exemplo, se você tem a especificação de uma feature por exemplo... a tela aqui vai ter que fazer, tem esse comportamento... aí na outra feture você diz ah! vai fazer o contrário... e aí na outra feature você diz ah! num vai fazer nenhum dos dois. Então, se você bota isso numa determinada ordem, isso vai mudar determinada o comportamento da aplicação. Acho que é isso que tu tá querendo dizer.

[Focus Group Participant] é. Esse exemplo também cabe.

[Focus Group Participant] ...e o problema é você não enxergar isso.

[Focus Group Participant] é que tenho mais uma pergunta, fiquei na dúvida, é o seguinte. Quando ele geraria as classes, ele gera um pacote com o nome das features e o conjunto de classes dele. Se eu vou lá e especializo, eu estou especializando aquela feature e não as anteriores mesmo que eu tenha dependência.

[Researcher] na hora que você vai gerar o produto, você vai ter que selecionar, colocar a ordem que você vai juntar aquelas coisas. É isso que eu estou querendo dizer. Talvez a precedência delas... não sei se ficou...

[Focus Group Participant] você não conseguiu visualizar porque eu tava pensando no nível de features...

[Researcher] é exatamente isso que ele tava dizendo. Não fica tão fácil você entender e você perceber isso.

[Focus Group Participant] agora deixa eu fazer aqui uma pergunta. Que na aula passada, a gente tratou um pouco disso aqui na sala, com o colega também... é... à respeito do seguinte. Vamos pensar aqui no sistema de instalação da Microsoft, que você tem um dvd só e você faz a seleção do produto ali na hora da instalação. Você tem do starter ao ultimate, passando por várias etapas. Aí dentro, pra cada etapa daquelas, com certeza são separados módulos que vão ser instalados e disponibilizados no equipamento. Neste caso, nesse caso, esse modelo de implementação porque uma vez que você consegue pegar dentro da abordagem o que é que é importante, o que é que vai, e criar pacotes, grupos, vamos supor grupos ao invés de fazer essa derivação a caso. Não ajudaria a evitar que isso ocorresse? aí vai cair no que a gente tá falando... vai ficar bem fechadinho. Não vai ter essa coisa de mover demais, de misturar demais? Não seria algo que conseguisse ficar mais controlado? Porque se assim for, a abordagem vai ficar praticamente perfeita, né? vai ser o verdadeiro sonho de consumo de programador. Isso pode [inaudível] criar manutenção no futuro. Você vai direto no ponto. Mas isso ai é uma suposição, tá!? Eu não tenho essa visão ainda, da abordagem, pra poder dizer se é ou se não é, mas se fosse, seria perfeito. Porque a dependência você já deixa tudo definido na seleção. Ou seja, você selecionou isso aqui, tudo que vem dali já tá pre-estabelecido, já tá selecionado, já tá...

[Researcher] isso aí já está lá definido no feature model, nas restrições do feature model já é automaticamente definido. É isso. Alguém quer dar mais algum comentário? Quem ficou aí mais tímido, as meninas ali... que ficaram mais...

Então tá bom. Obrigado aí pela colaboração de vocês!

This volume has been typeset in LATEXwith the UFBAThesis class (<www.dcc.ufba.br/~flach/ufbathesis>). For details about this document, click here.