PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

Conditional compilation is often used to implement variability in configurable systems. This technique relies on #ifdefs to delimit feature code. Previous studies have shown that #ifdefs may hinder code comprehensibility. However, those studies did not explicitly take feature dependencies into account. Feature dependency occurs when different features refer to the same program element, such as a variable. Comprehensibility may be even more affected in the presence of feature dependency, as the developer must reason about different scenarios affecting the same variable. Our goal is to understand how feature dependency affects the comprehensibility of configurable system source code. We conducted four complementary empirical studies. In Study 1, forty-six developers responded to an online experiment. They executed tasks in which they had to analyze programs containing #ifdefs with and without feature dependency. However, feature dependencies may be of different types depending on the scope of the shared variable. In Study 1, we were not concerned with different types of dependency. Thus, in Study 2, we carried out an experiment in which 30 developers debugged programs with different types of feature dependency. Each program included a different type of feature dependency: global, intraprocedural, or interprocedural. We used an eye-tracking device to record developers' gaze movements while they debugged programs. However, feature dependencies do not differ from each other only in terms of types. They can also present differences in terms of number of dependent variables and degree of variability, among others. To address those characteristics, we complemented Study 1 and 2 by means of Studies 3 and 4. In Study 3, we executed a controlled experiment with 12 participants who analyzed programs with different numbers of dependent variables and number of uses of dependent variables. In Study 4, we carried out an experiment in which 12 developers analyzed programs with different degrees of variability. Our results show that: (i) analyzing programs containing #ifdefs and feature dependency required more time than programs containing #ifdefs but without feature dependency, (ii) debugging programs with #ifdefs and global or interprocedural dependency required more time and higher visual effort than programs with intraprocedural dependency, (iii) the higher the number of dependent variables the more difficult it was to understand programs with feature dependency, and (iv) the degree of variability did not affect the comprehensibility of programs with feature dependency. In summary, our studies showed that #ifdefs affected comprehensibility of configurable systems in different degrees depending on the presence or not of feature dependency, on the type of feature dependency, and on the number of dependent variables.

DSC | 041 | 2023

Comprehensibility of Source Code with Feature Dependency in Configurable Systems

Djan Almeida Santos

UFBA

Comprehensibility of Source Code with Feature Dependency in Configurable Systems

Djan Almeida Santos

Tese de Doutorado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

Março | 2023

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

COMPREHENSIBILITY OF SOURCE CODE WITH FEATURE DEPENDENCY IN CONFIGURABLE SYSTEMS

Djan Almeida Santos

PHD THESIS

Salvador
28 de março de 2023

DJAN ALMEIDA SANTOS

COMPREHENSIBILITY OF SOURCE CODE WITH FEATURE DEPENDENCY IN CONFIGURABLE SYSTEMS

Esta tese de doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia como parte do cumprimento parcial dos requisitos para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Dr. Cláudio Nogueira Sant'Anna
Co-orientador: Dr. Márcio de Medeiros Ribeiro

Salvador
28 de março de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

S237 Santos, Djan Almeida

Comprehensibility of source code with feature dependency in configurable systems / Djan Almeida Santos. – Salvador, 2023.

114p.: f.

Orientador: Prof. Dr. Cláudio Nogueira Sant'Anna

Coorientador: Prof. Dr. Márcio de Medeiros Ribeiro

Tese (Doutorado) – Universidade Federal da Bahia. Instituto de Computação, 2023.

1. Systems - Configurable . 2. Computing. 3. Feature dependency. I. Sant'Anna, Cláudio Nogueira. II. Ribeiro, Márcio de Medeiros. III. Universidade Federal da Bahia. IV. Título.

CDU 004.45

Djan Almeida Santos

Comprehensibility of Source Code with Feature Dependency in Configurable Systems

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, 28 de março de 2023

[illegible]

Prof. Dr. Cláudio Nogueira Sant'Anna (Orientador - UFBA)

Rohit Gheyi

Prof. Dr. Rohit Gheyi (UFCG)

[illegible]

Prof. Dr. Flávio Mota Medeiros (IFAL)

[illegible]

Prof. Dr. Eduardo Santana de Almeida (UFBA)

[illegible]

Prof. Dr. Ivan do Carmo Machado (UFBA)

ACKNOWLEDGEMENTS

Consagre ao Senhor tudo o que você faz, e os seus planos serão bem-sucedidos. Provérbios 16:3. Primeiramente agradeço ao meu Senhor, meu Deus, por ter me permitido alcançar esse objetivo. Por ter me dado forças e por ter me ajudado a nunca pensar em desistir.

Gostaria de agradecer ao meu orientador, o professor Cláudio Sant'Anna. Cláudio muito obrigado por toda confiança, suporte, atenção, ensinamentos, e paciência dedicados a mim durante esses anos. A forma como você conduziu a sua orientação foi determinante para que eu chegasse até aqui, e espero poder repassar teus ensinamentos a meus orientados na minha carreira acadêmica. Gostaria de agradecer também ao meu coorientador, o professor Márcio Ribeiro. Sempre fui um grande admirador dos seus trabalhos. Márcio muito obrigado por todas as nossas conversas, ensinamentos, e por estar sempre disponível. A praticidade com que você conduziu nossos estudos foram primordiais para que eles fossem exitosos. Muito obrigado de verdade! É um orgulho tê-los como meus orientadores.

Gostaria de agradecer aos colegas do IFBA, *campus* Vitória da Conquista, vocês foram essenciais desde o primeiro momento, quando me permitiram afastar da sala de aula, até os últimos, quando participaram de todos os meus estudos. A vocês a minha gratidão e espero retribuir com o mesmo carinho e preocupação que dispensaram a mim.

Gostaria de agradecer aos professores, Cláudio, Manoel, Eduardo, Rafael, temos excelentes professores na UFBA. Foi prazeroso e desafiador os nossos momentos de aulas e projetos. Também gostaria de citar dois professores da época do Mestrado, o professor Paulo Bonomo, que sempre me socorreu nos meus problemas estatísticos, e o professor Ronaldo Thibes, um dos grandes incentivadores da minha carreira científica. Também gostaria de agradecer aos meus colegas da UFBA. Muitos estiveram presentes durante a minha passagem pela UFBA, agradeço a todos por isso. Mas poucos compartilharam os momentos de almoços, de viagens, de conselhos, e de risadas. Muito obrigado Luis Paulo, Dósea, Crescêncio, Cláudio, Pablo, Tiago Motta, Leandro e Jonatas.

Gostaria de agradecer aos meus amigos Danilo e Maria. Sem eles, minha passagem por Salvador teria sido muito custosa e solitária. Muito obrigado meus amigos por toda receptividade e por me acolher tão bem.

Por último, e não menos importante, gostaria de agradecer a minha esposa Jéssica. Ninguém mais que nossa família sente os efeitos de conviver com um doutorando. E ela sempre compreensiva e dedicada a família. Aos meus filhos, Martim, Rael e Dimi, que sempre souberam dosar suas energias no momento que papai estava em reunião ou estudando. Nossos filhos são nossas alegrias. Aos meus pais, Vavá e Maurisa, meu irmão Ialy, por estarem sempre ao meu lado e me darem todo incentivo e confiança. Amo todos vocês.

ABSTRACT

Conditional compilation is often used to implement variability in configurable systems. This technique relies on #ifdefs to delimit feature code. Previous studies have shown that #ifdefs may hinder code comprehensibility. However, those studies did not explicitly take feature dependencies into account. Feature dependency occurs when different features refer to the same program element, such as a variable. Comprehensibility may be even more affected in the presence of feature dependency, as the developer must reason about different scenarios affecting the same variable. Our goal is to understand how feature dependency affects the comprehensibility of configurable system source code. We conducted four complementary empirical studies. In Study 1, forty-six developers responded to an online experiment. They executed tasks in which they had to analyze programs containing #ifdefs with and without feature dependency. However, feature dependencies may be of different types depending on the scope of the shared variable. In Study 1, we were not concerned with different types of dependency. Thus, in Study 2, we carried out an experiment in which 30 developers debugged programs with different types of feature dependency. Each program included a different type of feature dependency: global, intraprocedural, or interprocedural. We used an eye-tracking device to record developers' gaze movements while they debugged programs. However, feature dependencies do not differ from each other only in terms of types. They can also present differences in terms of number of dependent variables and degree of variability, among others. To address those characteristics, we complemented Study 1 and 2 by means of Studies 3 and 4. In Study 3, we executed a controlled experiment with 12 participants who analyzed programs with different numbers of dependent variables and number of uses of dependent variables. In Study 4, we carried out an experiment in which 12 developers analyzed programs with different degrees of variability. Our results show that: (i) analyzing programs containing #ifdefs and feature dependency required more time than programs containing #ifdefs but without feature dependency, (ii) debugging programs with #ifdefs and global or interprocedural dependency required more time and higher visual effort than programs with intraprocedural dependency, (iii) the higher the number of dependent variables the more difficult it was to understand programs with feature dependency, and (iv) the degree of variability did not affect the comprehensibility of programs with feature dependency. In summary, our studies showed that #ifdefs affected comprehensibility of configurable systems in different degrees depending on the presence or not of feature dependency, on the type of feature dependency, and on the number of dependent variables.

Keywords: Configurable systems, comprehensibility, feature dependency.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTING OF SOUCE CODE

Chapter 1

INTRODUCTION

1.1 GENERAL CONTEXT

Software development companies search for ways to increase productivity and reduce production costs. Configurable systems have become a way (CLEMENTS; NORTHROP, 2002; SCHMID; RUMMLER, 2012; CAVALCANTE et al., 2012), as they support the creation of different systems with adapted configurations, promoting the systematic reuse of software components and assets (CLEMENTS; NORTHROP, 2003). Configurable systems are made up of several features that may be enabled or disabled allowing variability (GARVIN; COHEN, 2011). The features that differ the configurations of a configurable system are called variability (APEL; BEYER, 2011).

There are large industrial product lines (CLEMENTS; NORTHROP, 2002; BERGER et al., 2014) and open-source systems, like the Linux kernel, that are examples of configurable systems (ABAL; BRABRAND; WASOWSKI, 2014; ABAL et al., 2018). Developing configurable systems requires developing codes that promote variability, in addition to traditional implementations of features, structures, processes, interfaces, among others. One of the techniques most used to allow variability is conditional compilation. By means of preprocessor directives, like #ifdef, this technique allows developers to include or exclude code fragments that will or will not be compiled (ERNST; BADROS; NOTKIN, 2002; LIEBIG et al., 2010; GARVIN; COHEN, 2011). While programming, developers use #ifdefs to delimit code fragments related to optional or alternative features. Then, only features explicitly enabled are compiled.

On the Listing 1.1, we have an example of using conditional compilation in code snippets limited by #ifdefs. Between lines 8 and 11 we have a piece of code of feature A and between lines 13 and 15 a piece of code of feature B. Each piece of code delimited by #ifdefs is part of the scope of each feature and it will only be compiled if the features are enabled. For a feature to be enabled, it must be previously defined and enabled in a system configuration file or in the source code header itself. To define features we use the command `#define`. Lines 4 and 5 on Listing 1.1 define and enable features A and B. Therefore, this example illustrates a source code snippet where the source code of features A and B will be compiled.

Listing 1.1 Code example with feature implementation using #ifdef

```c
#include <stdio.h>
#include <conio.h>
...
#define A
#define B
...
int main(){
#ifdef A
    int x;
    x = 1;
#endif
...
#ifdef B
    x++;
#endif
}
```

(BANIASSAD; MURPHY, 1998; RIBEIRO et al., 2010; MEDEIROS; RIBEIRO; GHEYI, 2013a; RIBEIRO; BORBA; KÄSTNER, 2014; MEDEIROS et al., 2015; RODRIGUES et al., 2016). Spencer and Collyer (SPENCER; COLLYER, 1992) presented, in an initial study, some activities that are impaired by #ifdef during a maintenance task and highlighted localizing feature dependencies as one of those activities. Medeiros *et al.* (MEDEIROS et al., 2015; MEDEIROS et al., 2017a) focused their studies on evaluating the number of possibilities for applying refactorings in configurable systems. They concluded that the use of #ifdef generates several problems that harm refactorings, including the fact that, #ifdefs tend to generate a high number of dependencies between features. They also mentioned the importance of disciplining the use of #ifdef to facilitate the task of maintenance (MALAQUIAS et al., 2017).

In Listing 1.1 we have an example of feature dependency. The variable `x` is defined inside feature A and has its value incremented inside feature B. In this case, feature B depends on feature A, because of variable `x`, which is, thus, called dependent variable. Maintainability problems involving feature dependencies happen when a developer changes the behavior of a dependent variable in a feature and does not consider all the effects of this change on other dependent features. In Listing 1.1, a developer could, for example, change the type or name of the variable `x` and this change would reflect on feature B and all other features dependent on feature A.

Previous studies have stated that the use of code with #ifdef is prone to introducing defects in source code and increasing the complexity of configurable systems (LIEBIG et al., 2010; GARVIN; COHEN, 2011). For example, on in Listing 1.1 the developer needs to mentally simulate the execution of the source code in Listing 1.1 considering four possible configuration scenarios: (i) only feature A enabled, (ii) only feature B enabled, (iii) both features enabled and (iv) both features disabled. These possibilities for various configuration combinations make configurable systems complex and difficult to understand. Comprehensibility is hinder because the multiple scenarios obfuscate the source code, making maintenance defect-prone and costly (CAFEO et al., 2012). As a consequence, the so-called variability bugs are likely to be introduced.

A variability bug is a defect that occurs only in some scenarios of a configurable system, but not in all, when features are enabled or not (ERNST; BADROS; NOTKIN, 2002; MEDEIROS; RIBEIRO; GHEYI, 2013a; ABAL; BRABRAND; WASOWSKI, 2014; MEDEIROS et al., 2015; BRAZ et al., 2016). Listing 1.1 shows an example of a variability bug involving feature dependency. If a developer chooses a configuration with feature A disabled and feature B enabled, we will have a variability bug, as the piece of code where variable `x` is defined (feature A) will not be compiled and the piece of code of feature B uses `x` variable on line 7. In this case, the compiler will not warn the programmer that there is an undefined variable bug, since the compiler still does not know which features will be compiled. The developer will only realize this when compiling a configuration in which this defect appears. Some studies show that variability defects occur frequently (ERNST; BADROS; NOTKIN, 2002; GARVIN; COHEN, 2011; MEDEIROS; RIBEIRO; GHEYI, 2013a; ABAL; BRABRAND; WASOWSKI, 2014; MEDEIROS et al., 2015; BRAZ et al., 2016; ABAL et al., 2018).

Listing 1.2 Source code of GLib

```
1 GInetAddress *g_inet_address_new_from_string (...) {
2 #ifdef G_OS_WIN32
3     struct sockaddr_storage sa;
4     ...
5     gint len;
6 #else  /* NOT G_OS_WIN32 */
7     ...
8 #endif
9     (void) g_inet_address_get_type ();
10    ...
11 }
```

Listing 1.2 displays a code snippet from *GLib*[1], a real configurable system available in the Git repository[2]. *GLib* is a general purpose library for applications written in C. Line 1 of Listing 1.2 presents the `g_inet_address_new_from_string` function that parses a string containing an IP address. Inside this function, there is a call to `g_inet_address_get_type()` (line 9). To ensure that the compiler did not optimize the return value of this function, the developer had to modify this code. For this, She or he added a variable to the source code to receive the return value of the function. Listing 1.3 shows these modifications. The developer added a `type` variable on line 5 in blue and on line 10 in yellow she or he assigns the return value of the function to the `type` variable.

Observing Listing 1.3 we noticed that these changes generated a variability defect. The defect happens because the `type` variable (line 5) was defined inside a #ifdef block and, therefore, is only accessible when the `G_OS_WIN32` feature is enabled. If we do not enable `G_OS_WIN32`, that is, in a non-Windows system, we will get an "undefined variable"

[1]https://developer.gnome.org/glib/
[2]https://git.gnome.org/browse/glib/

variability bug because the variable `type` will be used without having been defined and we will not be able to compile the code. Note that this case is simple, but it happens frequently in real configurable systems because of the different features of the system (ABAL; BRABRAND; WASOWSKI, 2014). This type of bug, as well as the one in the Listing 1.1 are not accused by compilers or identified during code development, sometimes they are only identified at compile time or runtime if the feature containing the defect is enabled in that configuration.

Listing 1.3 Modified source code of GLib

```
 1 GInetAddress *g_inet_address_new_from_string (...) {
 2 #ifdef G_OS_WIN32
 3     struct sockaddr_storage sa;
 4     ...
 5     volatile GType type;
 6     gint len;
 7 #else /* NOT G_OS_WIN32 */
 8     ...
 9 #endif
10     type = g_inet_address_get_type ();
11     ...
12 }
```

Listing 1.4 Fixed variability bug in *GLib*

```
 1 GInetAddress *g_inet_address_new_from_string (...) {
 2     volatile GType type;
 3 #ifdef G_OS_WIN32
 4     struct sockaddr_storage sa;
 5     ...
 6     gint len;
 7 #else /* NOT G_OS_WIN32 */
 8     ...
 9 #endif
10     type = g_inet_address_get_type ();
11     ...
12 }
```

Listing 1.4 shows a code modification to fix the variability bug. The developer moved the definition of the variable `type` to a mandatory part of the code, moving it from line 5 to line 2. The result of this modification is the code in the Listing 1.4. This modification guarantees the presence of the code that defines the variable `type` (in blue) ceasing this dependency and correcting the problem.

Many researchers suggest that scenarios like *GLib*, which need modification in source code with #ifdefs, increase problems and maintenance effort, as they harm the comprehensibility by forcing the developer to think about different scenarios affecting the same variable (CATALDO et al., 2009; RIBEIRO et al., 2010; RIBEIRO et al., 2012; QUEIROZ et al., 2012; SCHULZE et al., 2013; RIBEIRO; BORBA; KÄSTNER, 2014; RODRIGUES et al., 2016; CAFEO et al., 2016; MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017; MALAQUIAS et al., 2017; OLIVEIRA; CAFEO; HORA, 2019).

1.2 PROBLEM STATEMENT

This section provides an overview of the research problems that motivate thesis.

Problem 1: Feature dependency can affect the comprehensibility of configurable systems.

Configurable systems usually include a high number of features implemented with #ifdefs. Thus, it is likely that two or more features share program elements and obfuscate the source code, causing difficulty for understanding it.

The developer when debugging codes with #ifdef needs to reason about different possible scenarios to understand a configurable system. Schulze *et al.* (SCHULZE et al., 2013) showed that #ifdefs make program debugging more difficult and time-consuming. Melo *et al.* (MELO et al., 2017) compared source code snippets with and without #ifdefs and confirmed that #ifdefs increased debugging time and developer visual effort. However, these studies do not explicitly consider feature dependencies when investigating the comprehensibility of source code with #ifdef. The effort to understand source code that contains feature dependencies can be even higher than code without dependencies, as developers need to reason about different scenarios that affect the same variable.

Problem 2: Influence of feature dependency types on source code comprehensibility of configurable systems.

Feature dependencies are common in practice (RIBEIRO; BORBA; KÄSTNER, 2014). Source code with feature dependency have different characteristics that depend on the definition scope of the shared variable. Rodrigues *et al.* defined three types of features dependencies: global, intraprocedural, and interprocedural (RODRIGUES et al., 2016). Rodrigues *et al.* (RODRIGUES et al., 2016) stated that interprocedural dependencies occur more frequently and that they can be more difficult to detect and correct. However, it is necessary to understand more clearly and with other perspectives how #ifdefs affect comprehensibility when their use causes different types of feature dependency.

Problem 3: Influence of the number of dependent variables on the comprehensibility of source code of configurable systems.

#ifdefs affect comprehensibility when their use implies different types of feature dependencies (SANTOS; SANT'ANNA, 2019). However, feature dependencies do not differ from each other only in terms of types, they can also present differences in terms of the number of dependent variables, the number of uses of each of these variables, variability, functions, data flow, control-flow, among others.

The dependency between two features may include just one dependent variable or more than one. It is reasonable to suspect that a high number of dependent variables and their uses makes the analysis of variability scenarios more complex. In this sense, it is also important investigating whether the comprehensibility of configurable systems varies not only based on the types of feature dependencies but also according to the number of dependent variables present in the source code.

Problem 4: Influence of degree of variability on the comprehensibility of

source code of configurable systems.

A study by Melo *et al.* (MELO; BRABRAND; WASOWSKI, 2016) indicates that the time to find defects in configurable systems increases linearly with the increase of variability. Their study also reveals that increasing the variability does not affect the effectiveness of finding defects. Despite an important contribution, their study did not control the number of dependent variables either the number of uses of dependent variables.

It is necessary to understand more clearly how degrees of variability affect program comprehension controlling factors that influence in the comprehensibility of source code of configurable systems like feature expressions, feature constant and dependent variables.

1.3 MAIN GOAL AND RESEARCH QUESTIONS

This work addresses the following overall research question:

How does feature dependency affect the comprehensibility of configurable systems source code?

In this context, our general objective is to investigate the influence of feature dependency on the comprehensibility of source code of configurable systems implemented with #ifdefs. Besides the presence of feature dependency, we investigate some characteristics of feature dependencies we suppose that can impact source code comprehensibility, namely: (i) types of dependency, (ii) number of dependent variables, and (iii) number of feature expressions and feature constants involved in the dependencies. For this, we undertake empirical analyses that take into account the cognitive and behavioral factors of developers, supported by eye tracker and heart rate monitor (smartwatch).

We expected that our findings with this work can inspire, as future work, the creation of tools to support programmers for the challenges of reasoning about dependencies between features, contributing to more effective debugging and, ultimately, decreasing defects in configurable systems.

RQ1: *Does feature dependency affect the comprehensibility of configurable systems?*

Our first research question deals with problem 1 (Section 1.2). We investigated whether or not code containing feature dependency affected the comprehensibility of configurable systems.

For that we carried out our Study 1, an online experiment with forty-six developers. A online experiment is a research method for collecting information that describes, compares or explains knowledge, attitudes, and behaviors (PFLEEGER; KITCHENHAM, 2001; KOHAVI et al., 2009; BAKSHY; ECKLES; BERNSTEIN, 2014). Participants performed tasks in which they had to analyze programs containing #ifdef with and without feature dependency. We quantified comprehensibility by means of the time and number of tentative participants took to answer each task correctly.

RQ2: *How do different types of feature dependency affect source code comprehensibility of configurable systems?*

This research question refers to problem 2. Therefore, we investigated how different types of feature dependencies affect the comprehensibility of configurable systems.

We performed Study 2, a controlled experiment with 30 participants who debugged programs with different types of feature dependencies while their eye movements were recorded. We quantified comprehensibility using metrics based on participants' visual effort, time spent on each task, and number of tasks performed correctly.

RQ3: *How do different numbers of dependent variables affect the comprehensibility of configurable system source code?*

This research question answered Problem 3. Thus, we analyzed how the comprehensibility of configurable systems were affected by feature dependencies with different characteristics in terms of number of dependent variables.

For that, we executed Study 3, a controlled experiment with 12 participants who analyzed programs trying to specify the output. We quantified comprehensibility using metrics based on time and tentative to answer tasks correctly, participants visual effort and participants' heart rate.

RQ4: *How do degrees of variability affect the comprehensibility of configurable system?*

This research question answered Problem 4. Thus, we investigated how the comprehensibility of configurable systems were affected by degrees of variability taking into account dependent variables, feature expressions and feature constant.

For that, we executed Study 4, a controlled experiment with 12 participants who analyzed programs trying to specify the output. We also quantified comprehensibility using metrics based on time and tentative to answer tasks correctly, participants visual effort and participants' heart rate.

1.4 CONTRIBUTIONS

The contributions of this thesis are:

- Feature dependency state-of-the-art since we contributed to updating the knowledge of feature dependency in configurable systems;

- Understanding how feature dependencies affect comprehensibility in terms of the time and attempt to specify the output in configurable systems (Chapter 3).

- Understanding how different types of feature dependency affect the comprehensibility in terms of the time, attempt, and visual effort of bug-finding in configurable systems (Chapter 4).

- Understanding how different numbers of dependent variables affect the comprehensibility in terms of the time, tentative, visual effort, and heart rate variability of specify the output in configurable systems (Chapter 5).

- Understanding how different degrees of variability affect comprehensibility in terms of the time, tentative, visual effort, and heart rate variability of specify the output in configurable systems (Chapter 6).
- Providing the first studies of feature dependency using smartwatch (Chapter 5 and Chapter 6).
- All material from the empirical studies was made publicly available on the web so that the studies can be replicated or extended in further investigations;

1.5 PUBLICATIONS

We published the papers listed below in chronological order.

1. SANTOS, D.; SANTANNA, C. Influência da Dependência entre features na Compreensibilidade do Código Fonte de Sistemas Altamente Configuráveis In: Anais do Workshop de Teses e Dissertações do CBSOFT. São Carlos - SP - Brazil, 2018. p. 62-70.

2. SANTOS, D.; SANTANNA, C. How does feature dependency affect configurable system comprehensibility?. IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, Montreal - Quebec - Canadá, pp. 1929. doi:10.1109/ICPC.2019.00016.

We submit for publication the journal:

3. SANTOS, D.; SANTANNA, C; RIBEIRO, M. Comprehensibility of feature dependency in configurable system using #ifdef: Two empirical studies. Submitted to Journal of Systems and Software.

1.6 CHAPTER MAP

We divided this document as follows:

- **Chapter 2** provides an overview of the background of this research. In particular, we present the notion of conditional compilation, feature dependency and comprehensibility evaluation using eye tracking and smartwatch.
- **Chapter 3** aims to answer RQ1 showing the results of an online experiment conducted with software developers that investigated if feature dependency affect comprehensibility of configurable systems.
- **Chapter 4** aims to answer RQ2 showing the results of a controlled experiment, with which we analyzed how different types of feature dependency affect the comprehensibility of configurable systems.
- **Chapter 5** aims to answer RQ3 showing the results of a controlled experiment, with which we investigated how dependent variables affect the comprehensibility of configurable systems.
- **Chapter 6** summarizes our work by describing what we have done in the context of this research. In addition, we point out perspectives on future research directions.

Chapter 2

BACKGROUND

2.1 CONFIGURABLE SYSTEMS

Configurable systems consist of features (functionalities) that can be enabled or disabled allowing a great variability in the instantiation of different configurations of a system (GARVIN; COHEN, 2011). Configurable systems create configurations by combining a reusable set of features to instantiate products to meet the requirements of industry segments and extend portability across different hardware platforms. Configurable systems include industrial product lines (CLEMENTS; NORTHROP, 2002; POHL; BÖCKLE; LINDEN, 2005; BERGER et al., 2014) and open-source systems. In some cases, such as in the Linux kernel, thousands of feature configuration options are used to control the compilation process (BERGER et al., 2013b).

Software variability supports the development of configurable systems. Variability is the concept that allows deriving software systems (program variants) from a common source code by setting configuration options as enabled or disabled.(MELO et al., 2017). In other words, variability means that the developer can control whether to include or exclude certain features from a program variant. Configuration options range from small pieces of code to entire modules (APEL et al., 2013).

It can be said that the main purpose of developing configurable systems is to manage variability, support the automated generation of products and facilitate the reuse and adaptation of systems to different contexts (APEL et al., 2013). To implement configurable systems, various technologies can be used: object-oriented patterns, aspect-oriented programming, domain-specific languages and code generation, plug-in mechanisms, and so on. Among these, C preprocessor (cpp) conditional compilation directives are one of the oldest, simplest, and most popular mechanisms in use (ERNST; BADROS; NOTKIN, 2002; KÄSTNER; APEL; KUHLEMANN, 2008; LIEBIG et al., 2010).

2.2 CONDITIONAL COMPILATION DIRECTIVES

Programmers often use conditional compilation directives to implement variability (LE; WALKINGSHAW; ERWIG, 2011). In the C language, pre-processing directives can be recognized by the # symbol in the first column of the line in which they occur, and instruct the pre-processor whether a given piece of code should be compiled or not, according to the evaluation of a conditional expression. Every decision structure must be terminated by #endif (FOROUZAN; GILBERG, 2000).

In Table 2.1, adapted from Forouzan and Gilbert, we show the set of commands for conditional compilation that can be used for implementing variabilities (FOROUZAN; GILBERG, 2000).

Table 2.1 Command set used in conditional compilation directives

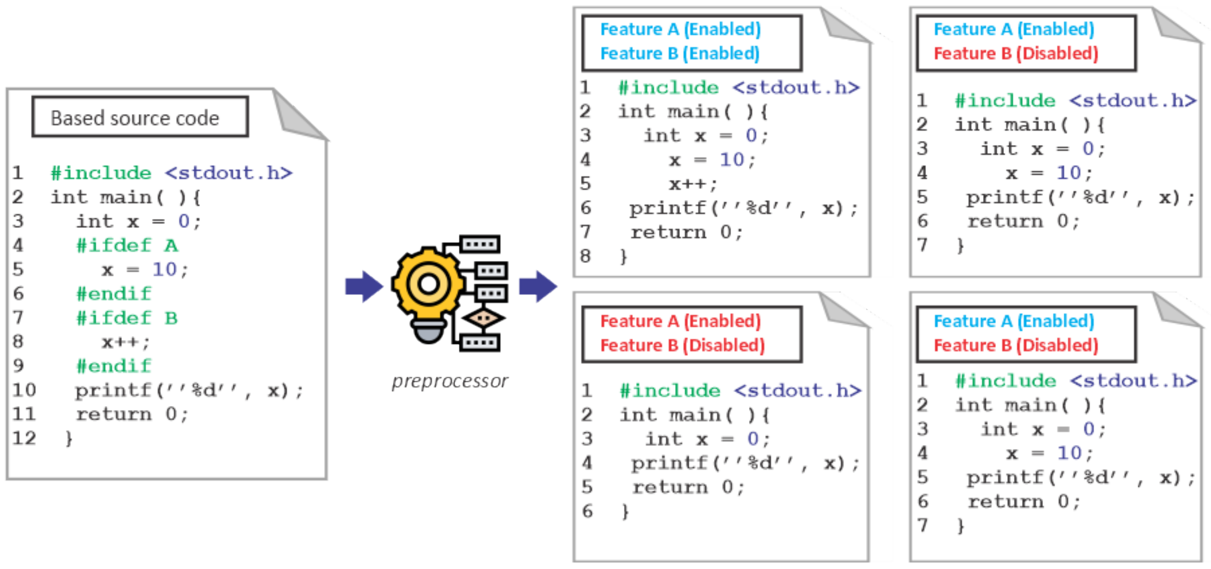
Command	**Meaning**
`#if` *expression*	Enables the inclusion of the code, if the expression is true.
`#endif`	Delimits the end of conditional compilation.
`#else`	Specifies the alternative code in a decision of two optional features.
`#elif`	Specifies the alternate code in a decision of several features.
`#ifdef` feature	Abbreviation of command `#if` associated with the expression `defined`.
`#ifndef` feature	Abbreviation of command `#if` associated with the expression `!defined`.

The #if directive controls compilation of parts of a source file. If the expression we write (after the #if) has a non-zero value, the group of lines immediately following the #if directive will be compiled. However, on configurable systems, the most common way to define features is using the #ifdef command which is the #if command including the abbreviation of expression `defined`. All conditional compilation directives such as #if and #ifdef must match a closing #endif directive before the end of the file. Otherwise, an error message will be generated.

The preprocessor will include code from features, evaluating each #ifdef or #elif until it finds an expression with a feature enabled. If all occurrences of #ifdef are disabled features or if no #elif appears, the preprocessor will include the code associated with the #else clause. The #if, #elif, #else, and #endif directives can nest within the text parts of other #if directives. Each nested #else, #elif, or #endif directive belongs to the closest previous #ifdef directive.

To exemplify how configurable systems are implemented using conditional compilation directives, we present Figure 2.1 adapted from Oliveira *et al.* (OLIVEIRA; CAFEO; HORA, 2019). On the left side we have the base code of the configurable system without going through the preprocessor. We have two features (A and B) delimited by #ifdef and #endif commands. Between lines 4 and 6 of the base code we have the code for feature A and between lines 7 and 9 the code for feature B. Pieces of code delimited by #ifdefs are part of the scope of each feature and will only be present in the final build of the product if features are enabled during pre-processing. This process is illustrated on the right side of Figure 2.1, in the four possible configurations after pre-processing the base code. The pre-processing process of this base code resulted in four programs with different source codes.

Figure 2.1 presents four configurations which we can obtain from the base code. Configuration 1 presents the possibility that features A and B are enabled, in this case when executing such code, the output value is number 11. In configuration 2, feature A is enabled and feature B disabled, so the code referring to *feature* B was not included in this configuration, and the system output is value 10. In configuration 3, where feature A and B are disabled, the output value is 0. Configuration 4, where only feature B is enabled, the system output is the value 1.



```
#include <stdout.h>
int main( ){
   int x = 0;
   #ifdef A
      x = 10;
   #endif
   #ifdef B
      x++;
   #endif
   printf(''%d'', x);
   return 0;
 }
```

```
#include <stdout.h>
int main( ){
   int x = 0;
      x = 10;
      x++;
   printf(''%d'', x);
   return 0;
 }
```

```
#include <stdout.h>
int main( ){
   int x = 0;
      x = 10;
   printf(''%d'', x);
   return 0;
 }
```

```
#include <stdout.h>
int main( ){
   int x = 0;
   printf(''%d'', x);
   return 0;
 }
```

```
#include <stdout.h>
int main( ){
   int x = 0;
      x = 10;
   printf(''%d'', x);
   return 0;
 }
```

Figure 2.1 Implementation of a configurable system with all multiple scenarios

Using conditional compilation for variability. In this case, variability means that specific features may or may not compile (ERNST; BADROS; NOTKIN, 2002; LIEBIG et al., 2010). The base code can originate different configurations, according to the amount of features present in it.

2.3 FEATURE DEPENDENCY

In Section 2.2, we discuss conditional compilation and see how this technique allows for variability in configurable systems. Despite its ease of use and the advantages of allowing variability, the use of conditional compilation can imply in feature dependencies.

Feature dependencies happen whenever a developer defines a variable in a feature and uses the same variable in another feature i.e. when two or more features use the same variable (BANIASSAD; MURPHY, 1998; RIBEIRO et al., 2010; RIBEIRO; BORBA; KÄSTNER, 2014; RODRIGUES et al., 2016).

For example, in Listing 2.1, the variable `final_grade` is defined in the feature `CALCULA-TE_GRADE`. Furthermore, `final_grade` is used on features `APPLY_PENALTY` (line 10) and `RESULT` (lines 16 and 18). Therefore, there is features dependency between: (i) features `CALCULATE_GRADE` and `APPLY_PENALTY`, (ii) features `CALCULATE_GRADE` and `RESULT` and (iii) features `APPLY_PENALTY` and `RESULT`. In this example, the variable `final_grade` is called *dependent variable* because it is the code element that causes the dependency.

In Listing 2.1, the developer needs to consider the following scenarios to understand the behavior of the variable `final_grade`: (i) both features `APPLY_PENALTY` and `RESULT` enabled; (ii) both features `APPLY_PENALTY` and `RESULT` disabled; (iii) `APPLY_PENALTY` enabled and `RESULT` disabled, and (iv) vice versa. The developer does not need to simulate scenarios with and without the feature `CALCULATE_GRADE`, because it encompasses all the code in the file. Mental simulation of all these scenarios can increase the comprehension effort.

Listing 2.1 Example of global dependency

```
 1 #ifdef CALCULATE_GRADE
 2 float final_grade = 0;
 3 #ifdef APPLY_PENALTY
 4 const float penalty = 1.5;
 5 int exceeded_time = 0;
 6 #endif
 7 int main() {
 8     printf ( "Enter the student grade:" );
 9     scanf ( "%f", &final_grade );
10     #ifdef APPLY_PENALTY
11     printf ( "Enter the exceeded time in minutes:" );
12     scanf ( "%f", &exceeded_time );
13     final_grade = final_grade - (exceeded_time * penalty);
14     #endif
15     #ifdef RESULT
16     if (final_grade >= 0 && final_grade < 5)
17         printf ("disapproved");
18     else if(final_grade >=5 && final_grade <7)
19         printf ("approved to MS.C.");
20     else
21         printf ("approved to Ph.D.");
22     #else
23         printf ("Final Grade = %f", final_grade);
24     #endif
25 ...
26 #endif
```

Based on the scope of definition and use of the dependent variable, Rodrigues *et al.* (RODRIGUES et al., 2016) defined three types of dependency between features: global, intraprocedural and interprocedural.

A *Global Dependency* occurs when different features refer to the same global variable. Listing 2.1 is an example of a global dependency. The variable `final_grade` (line 2) is a global variable defined in `CALCULATE_GRADE` and used by features `RESULT` and `APPLY_PENALTY`. As previously explained `final_grade` is a dependent variable on this example.

An *Intraprocedural Dependency* occurs when different features within a function refer to the same local variable. Listing 2.2 illustrates an intraprocedural dependency. In this case, the dependent variable is `days_delay`, which is defined inside the function `return_book()`. Still inside the function `return_book()`, the variable `days_delay` is used by features `FINE_IN_CASH` (line 11) and `PUNISHMENT_IN_DAYS` (line 13).

Listing 2.2 Example of intraprocedural dependency

```
 1 void return_book( ) {
 2 #ifdef FINE_IN_CASH
 3 const float fine_rate_day = 1.5, fine_amount = 0;
 4 #endif
 5 #ifdef PUNISHMENT_IN_DAYS
 6 const int days_of_punishment=1, days_punished = 0;
 7 #endif
 8 int days_delay = 2;
 9 if(days_delay>0){
10 #ifdef FINE_IN_CASH
11         fine_amount = days_delay * fine_rate_day;
12 #ifdef PUNISHMENT_IN_DAYS
13         days_punished= days_delay * days_of_punishment;
14 #endif
15 #endif
16 }
17 ....
18 int main() {
19         return_book( );
20 ...
```

Listing 2.3 Example of interprocedural dependency

```
 1 float calc_points(float total_points){
 2 ...
 3 #ifdef CONSIDER_SPECIALIZATION
 4         bool isSpecialist = true;
 5 #endif
 6 #ifdef CONSIDER_SPECIALIZATION
 7         if(isSpecialist)
 8                 total_points += 1;
 9 #endif
10 ...
11 return total_points;
12 }
13 int main() {
14   float total_points = 0;
15   total_points = calc_points(total_points);
16   printf("Total points = %.2f", total_points);
17 ...
```

An *Interprocedural Dependency* occurs when a variable is defined or used in a function and the content of that variable is passed as an argument to another function. In the first function, the variable is manipulated by a feature and, in the second function, the argument is used by another feature. Listing 2.3 exemplifies an interprocedural dependency. The variable `total_points` is defined in the function `main()` (line 14). Its content is passed to function `calc_points()`, where the corresponding parameter is used by feature `CONSIDER_SPECIALIZATION` (line 10). Despite not being inside an #ifdef, the `main()` function is also considered part of a feature, in this case, a mandatory feature. A feature is called mandatory when its presence is mandatory in all system configurations (CZARNECKI; HELSEN; EISENECKER, 2005). The other classifications for features are: (i) optional when a feature may or may not be necessary; (ii) inclusive alternative, when one or more features must be chosen; (iii) exclusive alternative, when only one of the features is needed; (iv) mutually inclusive alternative when two or more features are always needed together (CZARNECKI; HELSEN; EISENECKER, 2005).

2.4 VARIABILITY BUGS

Listing 2.4 Example of variability bug

```c
#ifdef CALCULATE_GRADE
float final_grade = 0;
#ifdef APPLY_PENALTY
const float penalty = 1.5;
int exceeded_time = 0;
#endif
int main() {
    printf ( "Enter the student grade:" );
    scanf ( "%f", &final_grade );
    #ifdef APPLY_PENALTY
    printf ( "Enter the exceeded time in minutes:" );
    scanf ( "%f", &exceeded_time );
    final_grade = final_grade - (exceeded_time * penalty);
    #endif
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
...
#endif
```

Features in a configurable system interact with and influence the functionality of other features. When these interactions induce errors that manifest themselves in certain configurations but not in others, or that manifest themselves differently in different configurations, we call it variability bug (ABAL; BRABRAND; WASOWSKI, 2014). As the number of configurations is exponential in the number of features, it is not always trivial to analyze each configuration separately. As a result, variability bugs occur frequently (ABAL; BRABRAND; WASOWSKI, 2014; ABAL et al., 2018). We use the term bug to refer to faults or defects in the program. A bug is an incorrect instruction in software, introduced into code as a result of human error. Bugs lead software to fail, such as a pointer being null when it should not be (IEEE, 1990).

Listing 2.4 shows an example of a variability bug involving dependency between features. In this code snippet we have two features. Feature `RESULT` (lines 15 to 24) and feature `APPLY_PENALTY` (lines 3 to 6 and 10 to 14). Feature `RESULT` prints the result obtained by a student in a language proficiency test. In this case, the result depends on the value of the variable `final_grade`. Feature `APPLY_PENALTY` imposes a penalty on the student if she or he exceeds the duration of the test. In this case, the student's result (variable `final-grade`) is decremented. The dependency between features is established when both use the same variable (`final_grade`) to calculate the student's grade. A change in the value of the variable in one of the features may compromise the behavior of the other feature.

Still in the Listing 2.4, we can see that the feature `APPLY_PENALTY` gives rise to two programs with different configurations. One configuration with feature `APPLY_PENALTY` enabled and another with feature `APPLY_PENALTY` disabled. Feature `APPLY_PENALTY` is optional, it may or may not be enabled when instantiating a product. A variability bug occurs when feature `APPLY_PENALTY` is enabled and the value of time exceeded by the student (variable `exceeded_time`) is large enough for the variable `final_grade` to have a negative value. In this case the program will fail because feature `RESULT` will print a wrong result. It is important to point out that variability bugs occur only in specific configurations and not in others.

2.5 COMPREHENSIBILITY OF PROGRAMS

Program comprehension is the activity of understanding how a software system or part of it works. Previous studies have found that program comprehension is an important cognitive process in software development, because developers spend most of their time trying to understand source code (MAYRHAUSER; VANS; HOWE, 1997; TIARKS, 2011). According to Singer *et al.* (SINGER et al., 2010), program comprehension occurs primarily before code changes, because developers need to explore source code and other artifacts to identify and understand the relevant code subset for the intended change. The strategies followed to understand software may vary between developers, depending on their personality, experience, skills, task at hand or used technology (MAALEJ et al., 2014). Program comprehension is a knowledge-intensive activity in which developers consume and produce a significant amount of knowledge about the software in question. Comprehension can be seen as the product of the development and coordination of various skills, such as reading (including word recognition), reading fluency, syntactic processing and knowledge of the meanings of words (RAYNER et al., 2006). It is difficult to identify which elements contribute to a high or low comprehension rate. Thus, identifying the factors that contribute to comprehension continues to challenge researchers.

Research on program comprehension began more than 50 years (SACKMAN; ERIKSON; GRANT, 1966). During that time, numerous studies were conducted to investigate how developers understand source code. In the past, to measure program comprehensibility, researchers used different techniques, such as: *think-aloud* protocols, memorization and comprehension tasks.

Think-aloud protocols can be seen as protocols when people speak loudly what they are thinking, or introspection. This technique has been used to observe cognitive processes where participants verbalize their thoughts, which are usually recorded in audio or video (SHAFT; VESSEY, 1995). The *think-aloud* protocol is a common technique, even today, for observing the comprehension process. Memorization tasks were another way to measure comprehensibility. This technique tested whether developers remembered a piece of source code they had seen earlier. They had to memorize and then reproduce the learned code. Although it seems strange today, since to understand a code the developer does not need to memorize and reproduce it, this was one of the most used techniques in the first studies on comprehensibility (SAMMET, 1983). Comprehension tasks were a technique known as "fill in the blanks". Developers were only able to fill in the blanks in code if they first understood it (SOLOWAY; EHRLICH, 1984). As we will see below, most of these approaches to measuring comprehensibility have evolved to include biometric equipment to capture human factors.

Software systems are everywhere these days, and there are hundreds of programming languages, design patterns, IDEs and other tools and techniques that are designed to support program comprehension. However, it is difficult to understand the cognitive processes of developers when they work with these tools. Thus, one of the challenges of software engineering is to measure comprehensibility considering the human factor.

Traditionally, comprehensibility studies mainly use metrics based on the programmer's ability and time to perform a task (HOFMEISTER et al., 2017; MELO et al., 2017). In addition to these conventional techniques, researchers are also embracing new measurement techniques to gain new perspectives on program comprehension. Recent works indicate that evaluating the comprehensibility of programs is a challenge for software engineering since it is necessary to consider cognitive aspects of the developers, which can only be measured by means of biometric equipment (SIEGMUND, 2016; SHARIF; FALCONE; MALETIC, 2012).

2.6 BIOMETRIC EQUIPMENTS

2.6.1 Eye Tracking device

Some studies (RAYNER, 1998; RAYNER et al., 2006; RAYNER, 2009) indicate that monitoring eye movements during reading can provide valuable information about moment-to-moment comprehension processes. In this sense, the inclusion of eye tracking and other biometric equipment (such as heart rate monitor, blood flow meter, magnetic resonance image) in scientific research involving the capture of individual's cognitive aspects is increasing (SIEGMUND, 2016; SHARIF; FALCONE; MALETIC, 2012). An eye tracker lets you capture data on where participants are looking. Figure 2.2 show three main components of eye movements during reading: saccades, fixations, and regressions. Studies indicate that these components are related to comprehensibility (RAYNER et al., 2006; JR; STAUB; RAYNER, 2007; RAYNER, 2009; HOFMEISTER et al., 2017).

While it usually appears that our eyes are gliding smoothly across the page of text as

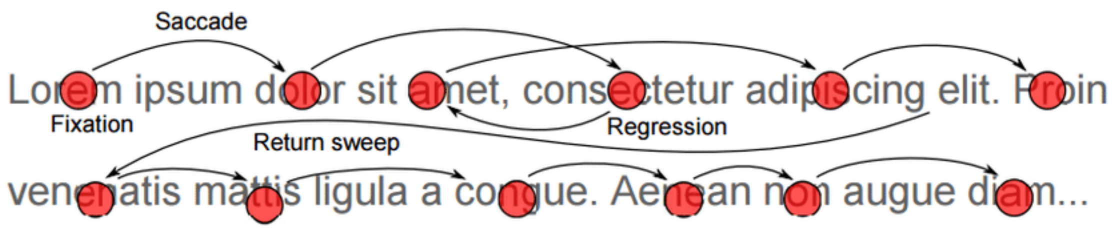

Saccade
Fixation
Return sweep
Regression
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin
venenatis mattis ligula a congue. Aenean non augue diam...

Figure 2.2 Eye movement during reading (JR; STAUB; RAYNER, 2007).

we read, in reality, they make a series of rapid movements (saccades) separated by pauses (fixations) that usually last approximately 200-250 ms (JR; STAUB; RAYNER, 2007; RAYNER et al., 2006). An eye fixation is a type of movement in which the gaze stops on an object of interest to obtain information (RAYNER et al., 2006; HOFMEISTER et al., 2017). Saccade movements are very quick voluntary movements between fixations. Among the saccade movements, the regressions are performed in the opposite direction to the reading direction (RAYNER, 2009). Return-sweeps are an essential eye movement that takes the readers' eyes from the end of one line of text to the start of the next. While return-sweeps are common during normal reading, the eye-movement literature is dominated by single-line reading studies where no return-sweeps are needed (JR; STAUB; RAYNER, 2007; RAYNER et al., 2006).

When readers encounter words that are more difficult to identify (eg, low-frequency words) or syntactically complex sentences, fixations become longer (RAYNER et al., 2006). It is generally assumed that as the text gets more difficult, readers make longer and more fixations, shorter saccades, and more regressions (RAYNER, 1998; RAYNER et al., 2006; JR; STAUB; RAYNER, 2007; RAYNER, 2009).

When the empirical questions focus on text or source code comprehensibility, some measures can be calculated, for example, the time of the first fixation, the time of all fixations in a region, the number of fixations in a region since the first entry into a region, the number of regressions, the amplitude of the regressions, however, the most useful measure in data analysis may vary with the specific study (SHARIF; FALCONE; MALETIC, 2012; SHARIF et al., 2013).

2.6.2 Smartwatch

Changes in the cognitive load influence the way the Autonomic Nervous System regulates the cardiovascular system and causes detectable variations of the heart rate known as heart rate variability (WALTER; PORGES, 1976; FRITZ et al., 2014; HIJAZI et al., 2021). One of the most important non-invasive markers used to access the regulation mechanisms of the Autonomic Nervous System over the cardiovascular system is heart rate variability, which is based on the evaluation of changes of time periods between consecutive cardiac cycles. (NAKAGAWA et al., 2014; COUCEIRO et al., 2019).

Heart rate variability (HRV) refers to the variation in time between successive heartbeats, also known as beat-to-beat intervals. The idea of using HRV to detect changes in the cognitive load is not new (WALTER; PORGES, 1976; TARVAINEN; RANTA-AHO; KARJALAINEN, 2002). However, the within-subject measurements of HRV are still uncertain because each subject exhibits distinct heart rate rhythms. To mitigate this risk, it is needed to capture a baseline of each subject and normalize the data with it.

Previous research has shown that certain biometric measures, such as heart rate variability can be linked to task difficulty or difficulty in comprehending small code snippets (FRITZ et al., 2014; MÜLLER; FRITZ, 2015; HIJAZI et al., 2021).

Stress level is the result of analysis of heart rate variability. Where the smartwatch device analyzes heart rate variability to determine your overall stress. The stress level range is from 1 to 100, where 1 is a very low-stress state and 100 is a very high-stress state (GARMIN, 2017).

2.7 RELATED WORK

In this section, we discuss related work. Studies related to feature dependencies are presented in subsection 2.7.1, while studies related to empirical studies about configurable systems are discussed in subsections 2.7.2 and 2.7.3.

2.7.1 Feature Dependencies

A variety of studies focused on feature dependencies. Liebig *et al.* defined structural metrics that measure the complexity of feature dependency (LIEBIG et al., 2010). They introduce, for example, several metrics that reflects the occurrences of #ifdef, number lines of feature code and the average nesting depth of #ifdefs. They applied the metrics in forty configurable systems. The study pointed out that code snippets with feature dependency imply a propensity for errors and a decrease in the abstraction of the feature code, which may be linked to low understanding. Rodrigues *et al.* classified the types of feature dependency (RODRIGUES et al., 2016). They also established a set of metrics that measure the occurrence of each type of dependency. Ribeiro *et al.* showed that feature dependency occurs in 65% of the methods of the systems they studied (RIBEIRO et al., 2012). Thus, feature dependency often occurs in practice.

It seems reasonable that more lines of feature code or the more feature expressions in a program, more difficult it will be to understand. However, software measurements cannot fully capture the complex process of understanding source code (SIEGMUND, 2016). Thus, we should not rely solely on measures based on code properties to quantify program comprehension. It is necessary to evolve to a scenario that includes the human factor in the process. In this context, we included the human factor in our studies.

2.7.2 Online experiment about variability in Configurable Systems

Some online experiments were conducted with developers to evaluate different aspects of variability. Berger *et al.* conducted an online experiment with industrial practitioners to investigate what they perceived as benefits and challenges of variability modeling (BERGER et al., 2013a). Their results show that industrial product line developers indicate a much wider applicability of variability modeling, exceeding simple configuration modeling with variability management, requirements specification, design and architecture planning. Villela *et al.* conducted an online experiment with practitioners from organizations developing systems with much variability in order to identify the variability management approaches currently used by companies (VILLELA et al., 2014). Muniz *et al.* evaluated the perception of developers in variability bugs on configurable systems with #ifdefs, and the tools and strategies used for developers to identify and remove them. Their results show that developers care about variability bugs, they may not detect some variability bugs reported in the bug trackers, and do not use proper tools to deal with them. Some variability bugs occur due to two feature interactions (MUNIZ et al., 2018).

Due to some limitations for online experiment executions, for example, a limited set of pre-defined questions, and inaccurate or incomplete answers, it is complicated for researchers to conduct online experiments on program comprehension. Our challenge was to conduct an online experiment where participants answered tasks while analyzing programs. online experiments can reach a large number of people from different countries in a short amount of time.

2.7.3 Experiments about Configurable Systems

Previous studies performed controlled experiments with developers to evaluate different aspects of configurable system source code. Schulze *et al.* observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor annotations (SCHULZE et al., 2013). Santos *et al.* carried out a quasi-experiment focused on understanding how variability affects bug-finding tasks using feature-oriented programming and conditional compilation. Their results show no significant statistical differences regarding the evaluated measures (correctness, understanding, or response time) in the tasks (SANTOS et al., 2019). In addition, some controlled experiments used biometric devices to evaluate humans' behavior while debugging programs. Siegmund *et al.* used images captured from a functional magnetic resonance imaging (fMRI) to identify patterns of brain activation for small comprehension tasks (SIEGMUND et al., 2014). However, due to the high cost of these types of equipment, researchers focuses more on using eye trackers and smartwatch to measure aspects of human cognition and attention.

Couceiro *et al.* show that mental effort of programmers in code understanding tasks can be monitored through heart rate variability (COUCEIRO et al., 2019), Müller *et al.* carried out an experiment to identify code quality concerns while a developer is making a change to the code (MüLLER; FRITZ, 2016). In the same way, Hijazi *et al.* used a smartwatch device to identify regions of source code that cause developerss comprehension difficulty to provide real-time comprehension support (HIJAZI et al., 2021). We carried out the first studies on configurable systems using heart rate metrics.

Kevic *et al.* used eye tracing device to identify the navigational behavior of the developer when performing a source code change activity (KEVIC et al., 2015). Costa *et al.* evaluated whether disciplined #ifdef annotations correlate with improvements in code comprehension and visual effort using an eye tracker (COSTA et al., 2021a). Melo *et* *al.* used an eye-tracking device to evaluate the comprehensibility of configurable systems (MELO et al., 2017). However, they compared programs with and without #ifdef of only two domains. Moreover, they did not analyzed their data taking feature dependency into account. This is the main difference from our work, which explicitly analyzed in details how feature dependency affect the comprehensibility of configurable systems and how feature dependency affect the comprehensibility of source codes containing #ifdefs.

Medeiros *et al.* performed an empirical study to evaluate a technique of detecting configuration-related weaknesses in configurable systems (MEDEIROS et al., 2020). Fenske *et al.* showed that functions with #ifdefs generally changed more frequently and more profoundly than other functions (FENSKE; SCHULZE; SAAKE, 2017a).

2.7.4 Variability Bugs

A group of researchers investigated the types of errors and bugs in source code of configurable systems. Medeiros *et al.* found and identified syntax errors in releases and commits of configurable systems (MEDEIROS; RIBEIRO; GHEYI, 2013b). In another study, Medeiros *et al.* performed an empirical study with 15 systems and identified some types of errors that developers have made in source code containing #ifdefs (MEDEIROS et al., 2015). Abal *et al.* performed a qualitative study about 42 bugs collected from bug-fixing commits of the Linux kernel repository, a large configurable system. They provided insights into the nature and occurrence of what they call as variability bugs, i.e. bugs caused by the use of #ifdefs (ABAL; BRABRAND; WASOWSKI, 2014).

2.7.5 #Ifdefs in undisciplined ways

A variety of studies have focused on investigating problems when developers use the #ifdefs in undisciplined ways. Malaquias *et al.* analyzed the importance of disciplined use of #ifdefs to facilitate the maintenance of configurable systems (MALAQUIAS et al., 2017). Medeiros *et al.* proposed a catalog of refactorings to convert undisciplined #ifdef usages into disciplined ones (MEDEIROS et al., 2017b). Da Costa *et al.* conducted a controlled experiment with the use of eye tracker to compare comprehensibility of programs with disciplined and undisciplined use of #ifdefs. (COSTA et al., 2021b).

Chapter 3

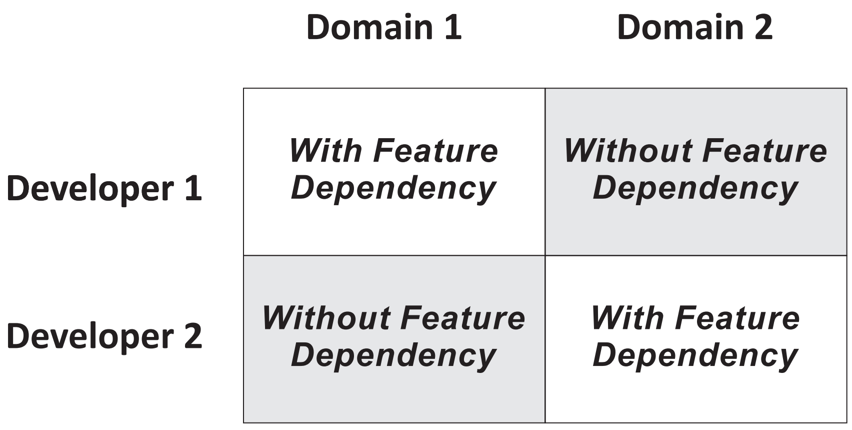
STUDY 1 - AN ONLINE EXPERIMENT ON HOW FEATURE DEPENDENCY AFFECTS PROGRAM COMPREHENSIBILITY

In this chapter, we describe the online experiment that aims at evaluating the impact of feature dependency on the comprehensibility of configurable systems. Configurable systems usually include a high number of features implemented with #ifdefs. But not all #ifdefs cause feature dependency. A feature dependency occurs in a configurable system when source code snippets of different features share code elements, like a variable. Several studies criticize #ifdef because their use obfuscates the source code, causing difficulty for understanding it. In this context, the following research question guided our study:
– ***How does feature dependency affect the comprehensibility of configurable system?***

3.1 DESIGN

As the COVID-19 pandemic was still ongoing in many parts of the world we performed an online experiment with 46 developers, who analyzed programs trying to specify the output. They analyzed different programs with and without feature dependency. We compared the comprehension effort they spent to analyze each program. We quantified comprehension effort based on two perspective: (i) the time the developers took to analyze each program and (ii) the number of attempts until the developers provide the correct answer. More details about these metrics are given in Section 3.6.

In order to avoid the learning effect, we used programs in two different domains. One domain was the simulation of control of products for sale (Domain 1). The other domain was the simulation of vaccination management for covid 19 (Domain 2). Each domain was implemented in two versions: with and without feature dependency. So we implemented four programs. Each participant realized tasks in both domains, yet some participants answered tasks about Domain 1 (Product control program) with feature dependency and



	Domain 1	Domain 2
Developer 1	*With Feature Dependency*	*Without Feature Dependency*
Developer 2	*Without Feature Dependency*	*With Feature Dependency*

Figure 3.1 Latin Square (2x2).

the others answered tasks about the same Domain 1 (Product control program), but without feature dependency.

We designed our study 1 as a standard Latin Square. The 2x2 Latin Square we adopted can be explained by means of Figure 3.1. Its columns are labeled with Domain 1 and Domain 2. Its rows are labeled with Developer 1 and Developer 2. The four squares in the center contains four programs with two treatments: programs with and without feature dependency. Therefore, according to his or her row, each developer performed tasks by analyzing two different programs in two different domains, one with feature dependency and the other without.

The Latin Square design controls one factor and its variations, ensuring that no row or column contains the same treatment twice. Our factor is the presence of feature dependency. Each line of our Latin Square has two programs in two different domains arranged in different orders to avoid learning effects. We developed a web system for the participants to answer the tasks. The system randomizes the programs for each participant, balancing, therefore, the Latin Square with the same number of data points for all programs. As we had 46 participants, we obtained 92 data points, 23 data points for each of the four programs.

3.2 PARTICIPANTS

In total, we counted with four undergraduates, three postgraduate students, six master students, seven doctoral students, nine have master degree, nine have doctoral degree, and eight professors. Twenty-one of the participants are developers from industry. Regarding the experience with programming languages, thirty participants reported having an experience with C for over than ten years. In addition, seven reported having it for five to ten years, five for two to five years. Regarding their #ifdef background knowledge, six participants are researchers working on topics related to #ifdef, thirteen frequently work with source code containing #ifdef, seventeen worked with source code containing #ifdef a few times, four have never worked with source code containing #ifdef but were able to understand the logic during the online experiment.

3.3 PROGRAMS

Feature dependency are common in configurable systems. More than half of functions with preprocessor directives have intraprocedural dependencies, while over a quarter of all functions have interprocedural dependencies (RODRIGUES et al., 2016). However, we could not use real system source code in our study 1 due to the following restrictions:
Domain. Our programs had to be on domains which our participants would be able to easily understand it. So, we avoided programs from configurable systems in complex domains, such as Linux.
Language. In order to simplify the understanding and to widen the audience of potential participants, we had versions of the programs with identifier names in English and versions in Portuguese, so that the participants could choose the language they were more familiar with.
Small programs. To avoid fatigue on the participants.

Keeping those restrictions in mind, we have created small programs, based on real systems, in two popular domain and written in C. We had variable names, feature names, function names and messages in English or in Portuguese for each program. The four programs are similar to each other in terms of number of lines of code (LOC) (LANZA; MARINESCU, 2007), number of features (APEL et al., 2013) and McCabe cyclomatic complexity (CC) (MCCABE, 1976). Following, we describe each program that we implemented.

Program 1 - Product control program with feature dependency. Listing 3.1 shows a code snippet of Program 1, which implements features of Domain 1 with feature dependency. It is about products that can be discard or returned. If the product is adulterated it will be discarded and if the product is expired it will be returned. It has three features: `CONTROL_EXPIRATION`, `CONTROL_STORE` and `CONTROL_RETURN`. `CONTROL_EXPIRATION` sorts products by expiration date. `CONTROL_STORE` checks if a product is adulterated and `CONTROL_RETURN` checks if a product can be returned.

Listing 3.1 shows the version of Domain 1 with feature dependency. Listing 3.1 has the following dependent variables: (i) *numberOfProduct*, defined in line 4 (mandatory feature) and used on feature `CONTROL_RETURN` in line 40. It is a global dependency. (ii) *numberOfDicardedProducts*, defined in line 26 (mandatory feature) and used on feature `CONTROL_RETURN` in line 41. This variable was defined and used within of discardProduct() function. It is a intraprocedural dependency. (iii) *p* defined in line 21 (mandatory feature) within of discardProduct() function. It is also parameter on feature `CONTROL_EXPIRATION` in line 23. This dependency is an interprodecural dependency.

Program 2 - Product control program without feature dependency. Listing 3.2 shows a code snippet of Program 2, which implements features of Domain 1 without feature dependency. This program implements the same product control domain of Program 1. The difference that exists between both versions is the use of dependent variables. It is about dispatching products based on their purchase date. The program has three features: `CONTROL_WEIGHT`, `ORDER_DELIVERY_QUEUE` and `CONTROL_FRAGILE_PRODUCT`. The feature `CONTROL_WEIGHT` verifies if the product is heavier than the limit. The feature `ORDER_DELIVERY_QUEUE` sorts the products by purchase date and the feature `CONTROL_FRA-`

GILE_PRODUCT checks if the product is fragile.

Listing 3.1 Code snippet of control product program with feature dependency

```
...
struct Product {
        char productName[50];
        int numberOfProduct = 0;
        #ifdef CONTROL_EXPIRATION
        char expiry_date[9];
        #endif
        #ifdef CONTROL_STORE
        bool adulterated;
        #endif
        #ifdef CONTROL_RETURN
        bool can_return;
        #endif
};
#ifdef CONTROL_EXPIRATION
bool checkExpiration(char expiry_date[9]) {
...}
void orderByExpiration(Product p[4]) {
...}
#endif
void discardProduct(Product p[4]) {
    #ifdef CONTROL_EXPIRATION
    orderByExpiration(p);
    #endif
    int i, n = 4;
        int numberOfDiscardedProducts = 0;
        int numberOfReturnedProducts = 0;
    for ( i = 0; i < n; i ++ ) {
        printf ("\n Product name: %s " , p[i].productName);
        printf ("\n Number of product: %d " , p[i].numberOfProduct);
        #ifdef CONTROL_STORE
        if ( p[i].adulterated ) {
        #endif
            numberOfDiscardedProducts += p[i].numberOfProduct;
            #ifdef CONTROL_EXPIRATION
            if ( checkExpiration(p[i].expiry_date) ) {
            #endif
                #ifdef CONTROL_RETURN
                if ( p[i].can_return ) {
                    numberOfReturnedProducts += p[i].numberOfProduct;
                    numberOfDiscardedProducts -= p[i].numberOfProduct
                        ;
                    printf ("\n Product returned");
                } else
                #endif
                    printf ("\n Product discarded");
            #ifdef CONTROL_EXPIRATION
            }
            #endif
        #ifdef CONTROL_STORE
            }
        #endif
        }
        printf ("\n Number of Discarded Products: %d " ,
           numberOfDiscardedProducts);
        printf ("\n Number of returned products: %d " ,
           numberOfReturnedProducts);
}
int main(){
        Product p[4] ;
        ...
        discardProduct(p);
        return 0;
}
```

Listing 3.2 Code snippet of control product program without feature dependency

```
...
struct Product {
        char productName[50];
        char purchaseDate[9];
        bool dispatched;
        #ifdef CONTROL_WEIGHT
        float weight;
        #endif
        #ifdef CONTROL_FRAGILE_PRODUCT
        bool fragileProduct;
        #endif
} p[4];
...
#ifdef ORDER_DELIVERY_QUEUE
void orderDelivery() {
...}
#endif
void dispatchedProduct() {
    #ifdef ORDER_DELIVERY_QUEUE
    orderDelivery();
    #endif
    #ifdef CONTROL_WEIGHT
    float limitWeightDelivery = 300;
    #endif
        #ifdef CONTROL_FRAGILE_PRODUCT
        int fragileProductTotal = 0;
        #endif
    int i, n = 4;
    for ( i = 0; i < n; i ++ ) {
        if ( !p[i].dispatched ) {
            #ifdef CONTROL_FRAGILE_PRODUCT
            if ( !p[i].fragileProduct ) {
            #endif
                #ifdef CONTROL_WEIGHT
35                 if ( p[i].weight < limitWeightDelivery ) {
36                 #endif
37                     printf("\n Product name: %s", p[i].productName);
38                     printf("\n Product purchase date: %s", p[i].
                           purchaseDate);
39                     #ifdef CONTROL_WEIGHT
40                     printf("\n Weight: %.2f", p[i].weight);
41                     #endif
42                     #ifdef CONTROL_FRAGILE_PRODUCT
43                     if ( !p[i].fragileProduct ) {
44                         printf("\n fragile product: No");
45                     }
46                     #endif
47                     p[i].dispatched = true;
48                     break;
49                 #ifdef CONTROL_WEIGHT
50                 } else
51                     printf("\n Overweight delivery product! The
                           customer must come to pick the product!");
52                 #endif
53             #ifdef CONTROL_FRAGILE_PRODUCT
54             } else {
55                 printf("\n Fragile product! The customer must come to
                       pick the product!");
56                 fragileProductTotal++ ;
57                 }
58             #endif
59         }
60         if ( i == n - 1 )
61             printf("\n There are no products to be dispatched!!!");
62     }
63 }
64 int main(){
65     ...
66     dispatchedProduct();
67     return 0;
68 }
```

Listing 3.2 shows the version of Domain 1 without feature dependency. Program 2 is similar to Program 1 in terms of number of lines of code (LOC), number of features and McCabe cyclomatic complexity (CC). The global variable *weight* is defined in line 7 and used in line 35. The intraprocedural variable *limitWeightDelivery* is defined in line 23 and used in line 35.

Program 3 - Vaccine control program with feature dependency. Listing 3.3 shows a code snippet of Program 3, which implements Domain 2 with feature dependency. It is about vaccines available for use. If a vaccine is expired or is reserved for use in second dose application it will be unavailable for use. It has three features: `CONTROL_EXPIRATION`, `CONTROL_SECOND_DOSE` and `CONTROL_TEMPERATURE`. The feature `CONTROL_EXPIRATION` sorts vaccines by expiration date. `CONTROL_SECOND_DOSE` checks if a vaccine is reserved for use in second dose application and the feature `CONTROL_TEMPERATU-`

RE sorts vaccine by the lowest storage temperature.

Listing 3.3 Code snippet of control vaccine program with feature dependency

```
...
struct Vaccine {
    char vaccineBrand[50];
    int numberOfAvailableVaccines;
    #ifdef CONTROL_SECOND_DOSE
    int numberOfRequireDoses;
    #endif
    #ifdef CONTROL_EXPIRATION
    char expiry_date[9];
    #endif
    #ifdef CONTROL_TEMPERATURE
    int temperatureOfStorage;
    #endif
};
#ifdef CONTROL_TEMPERATURE
void orderTemperatureQueue ( Vaccine v[4] ) {
...}
#endif
#ifdef CONTROL_EXPIRATION
bool checkExpiration ( char expiry_date[9] ) {
...}
#endif
void provideVaccineBatch ( Vaccine v[4] ) {
    #ifdef CONTROL_TEMPERATURE
    orderTemperatureQueue ( v );
    #endif
    int i, n = 4;
        int generalNumberOfVaccinesForUse = 0;
        int generalNumberOfVaccinesBlocked = 0;
    for ( i = 0; i < n; i ++ ) {
        #ifdef CONTROL_EXPIRATION
        if ( !checkExpiration( v[i].expiry_date ) ) {
        #endif
            #ifdef CONTROL_SECOND_DOSE
            if ( v[i].numberOfAvailableVaccines > v[i].
                numberOfRequireDoses ) {
            #endif
                printf ( "\n Vaccine's Brand: %s " , v[i].
                    vaccineBrand );
                #ifdef CONTROL_SECOND_DOSE
                v[i].numberOfAvailableVaccines = v[i].
                    numberOfAvailableVaccines - v[i].
                    numberOfRequireDoses;
                #endif
                generalNumberOfVaccinesForUse += v[i].
                    numberOfAvailableVaccines;
                v[i].numberOfAvailableVaccines = 0;
                #ifdef CONTROL_SECOND_DOSE
                v[i].numberOfAvailableVaccines = v[i].
```

```
                         numberOfRequireDoses;
45                   printf ( "\n Number of available vaccines: %d " , v[i
                         ].numberOfAvailableVaccines );
46                   printf (" \n Number of vaccines reservedr to second
                         dose: %d doses" , v[i].numberOfRequireDoses );
47                   #endif
48               #ifdef CONTROL_SECOND_DOSE
49               } else {
50                   printf( "\n Number of vaccines below the limit for
                         second dose." );
51                   generalNumberOfVaccinesBlocked += v[i].
                         numberOfAvailableVaccines;
52                                 }
53               #endif
54           #ifdef CONTROL_EXPIRATION
55           } else {
56               printf(" \n Expired vaccine. It will not be allowed." );
57               generalNumberOfVaccinesBlocked += v[i].
                     numberOfAvailableVaccines;
58           }
59           #endif
60       }
61           printf (" \n Number of vaccine to use: %d " ,
                 generalNumberOfVaccinesForUse );
62 }
63 int main(){
64       Vaccine v[4];
65       ...
66           provideVaccineBatch(v);
67           return 0;
68 }
```

Listing 3.3 shows the version of Domain 2 with feature dependency. The program has the following dependent variables:(i) *numberOfAvailableVaccines* defined in line 4 (mandatory feature) and used on feature CONTROL_SECOND_DOSE in line 35, 39 and 44. It is a global dependency. (ii) *generalNumberOfVaccinesBlocked* defined in line 29 (mandatory feature) and used on feature CONTROL_SECOND_DOSE in line 51. The variable were defined and used within of provideVaccineBatch() function. It is a intraprocedural dependency. (iii) *v* defined in line 23 (mandatory feature) within of provideVaccineBatch() function and is parameter on feature CONTROL_EXPIRATION in line 25. This dependency is an interprodecural dependency.

Program 4 - Vaccine control program without feature dependency. Listing 3.4 shows a code snippet of the program four implemented without feature dependency. This program implements the same Domain 2. It is about how can be vaccinated. The program has three features: CONTROL_AGE, ORDER_PREGNANT and CONTROL_ORDER_QUEUE. CONTROL_AGE verifies if the patient age is greater than the limit. ORDER_ORDER_QUEUE sorts the patients by age and CONTROL_PREGNANT checks if the patient is pregnant.

Listing 3.2 shows the version of Domain 2 without feature dependency. Program 4 is similar to program 3 in terms of number of lines of code (LOC), number of features and McCabe cyclomatic complexity (CC). The global variable *age* is defined in line 5 and used in line 39. The intraprocedural variable *vaccinateAgeLimit* is defined in line 23 and used in line 32.

Listing 3.4 Code snippet of control vaccine program without feature dependency

```
1  ...
2  struct Patient {
3          char name[40];
4          #ifdef CONTROL_AGE
5          int age;
6          #endif
7          char gender;
8          #ifdef CONTROL_PREGNANT
9          bool pregnant;
10         #endif
11         bool vaccinated;
12 } p;
13 ...
14 #ifdef ORDER_QUEUE
15 void orderPatientQueue ( ) {
16 ...}
17 #endif
18 void vaccinate ( ) {
19     #ifdef ORDER_QUEUE
20     orderPatientQueue();
21     #endif
22     #ifdef CONTROL_AGE
23     int vaccinateAgeLimit = 40;
24     #endif
25     #ifdef CONTROL_PREGNANT
26     int pregnantTotal = 0;
27     #endif
28     int i, n = 4;
29     for ( i = 0; i < n; i ++ ) {
30         if ( !p[i].vaccinated ) {
31             #ifdef CONTROL_AGE
32             if ( p[i].age > vaccinateAgeLimit ) {
33             #endif
34                 #ifdef CONTROL_PREGNANT
35                 if ( !p[i].pregnant ) {
36                 #endif
37                     printf( "patient's name: %s \n", p[i].name );
38                     #ifdef CONTROL_AGE
39                     printf( "Age: %d \n", p[i].age );
40                     #endif
41                     printf( "Gender: %c \n", p[i].gender );
42                     #ifdef CONTROL_PREGNANT
43                     if ( tolower( p[i].gender ) == 'f' && p[i].
                           pregnant )
44                         printf( "Pregnant: Yes \n" );
45                     #endif
```

```
                    p[i].vaccinated = true;
                    break;
                #ifdef CONTROL_PREGNANT
                } else {
                    printf( "Pregnant pacients cannot be vaccinated!
                       \n" );
                    pregnantTotal++;
                }
                #endif
            #ifdef CONTROL_AGE
            } else
                printf( "Insufficient age to vaccinate: %d \n", p[i].
                    name );
            #endif
        }
    }
}
int main(){
    ...
        vaccinate();
        return 0;
}
```

3.4 PILOT STUDIES

We carried out two pilot studies to adjust the online experiment procedure and artifacts. In the first one, Brazilians, which are Portuguese speakers, answered the online experiment using the Portuguese version of the website. The first run was pretty close to the final of the online experiment. The only difference was that initially, we showed array values in the task statement only. We performed the first pilot study with four programming students. Participants requested the array values to be also showed in the source code so that they were able to check them easier. Thus we adjusted that for the second pilot study.

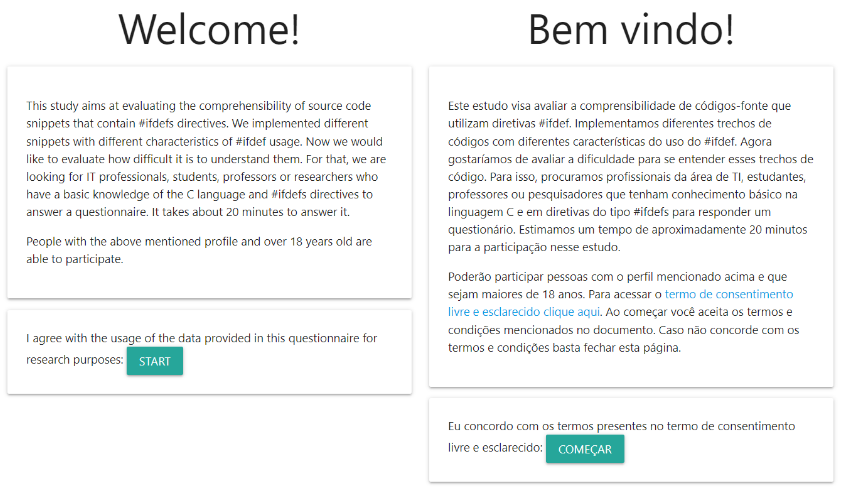
The participants of the second pilot study were Brazilians who have English as their second language. They answered the online experiment using the English version of the website. We have added an English version to increase the audience attendance. We performed the second pilot study with two programming professors. We did not consider the data collected in the pilot studies for our analysis.

3.5 PROCEDURE

After the pilot studies, we sent the online experiment invitation to two different groups. The first group comprised developers from our relationship group: (i) friends, (ii) professors, (iii) research group, and (iv) postgraduate students. We sent them an email message with a personalized salutation, an explanation about this study, and the link for the online experiment. Ten days later, we sent them a reminder. The second group consisted of developers recruited via social media: we published the online experiment on facebook[1], twitter[2], C programming language community in reddit[3], software engineering research groups and student groups in whatsapp[4] and asked developers to answer it. We left the online experiment open until we received no more responses for two weeks. This online experiment was open from August 22 to October 30, 2021.

The participants responded to the online experiment by means of a Web system we developed. Figure 3.2 shows the home page of the Web system. The home page presents the online experiment contents in English and in Portuguese. The participants choose the language she or he prefers. In the first page of the online experiment, we explained the goal of the online experiment and our research goals. We also informed the participants about the time estimated to finish the online experiment, and then, asked the respondents for their informed consent. Informed consent is a process of consenting to participation in research. Finally, the participants were requested to choose whether they wanted the online experiment in English or in Portuguese.

In the instructions page, the participant was informed about how to procedure to each task. Each participant had to understand and realize the mental execution of two programs. Each participant also had to execute three tasks concerning each program. The participants were not allowed to proceed to the next task until they correctly answered the task in progress. We explain that the number of attempts would be counted, so we recommended them to avoid random answers. Furthermore, the time to answer each one of the tasks would be recorded, so we recommended the participants to avoid losing focus during the task. For each new task, the web system would reset the timer.


Welcome!

This study aims at evaluating the comprehensibility of source code snippets that contain #ifdefs directives. We implemented different snippets with different characteristics of #ifdef usage. Now we would like to evaluate how difficult it is to understand them. For that, we are looking for IT professionals, students, professors or researchers who have a basic knowledge of the C language and #ifdefs directives to answer a questionnaire. It takes about 20 minutes to answer it.

People with the above mentioned profile and over 18 years old are able to participate.

I agree with the usage of the data provided in this questionnaire for research purposes: START

Bem vindo!

Este estudo visa avaliar a comprensibilidade de códigos-fonte que utilizam diretivas #ifdef. Implementamos diferentes trechos de códigos com diferentes características do uso do #ifdef. Agora gostaríamos de avaliar a dificuldade para se entender esses trechos de código. Para isso, procuramos profissionais da área de TI, estudantes, professores ou pesquisadores que tenham conhecimento básico na linguagem C e em diretivas do tipo #ifdefs para responder um questionário. Estimamos um tempo de aproximadamente 20 minutos para a participação nesse estudo.

Poderão participar pessoas com o perfil mencionado acima e que sejam maiores de 18 anos. Para acessar o termo de consentimento livre e esclarecido clique aqui. Ao começar você aceita os termos e condições mencionados no documento. Caso não concorde com os termos e condições basta fechar esta página.

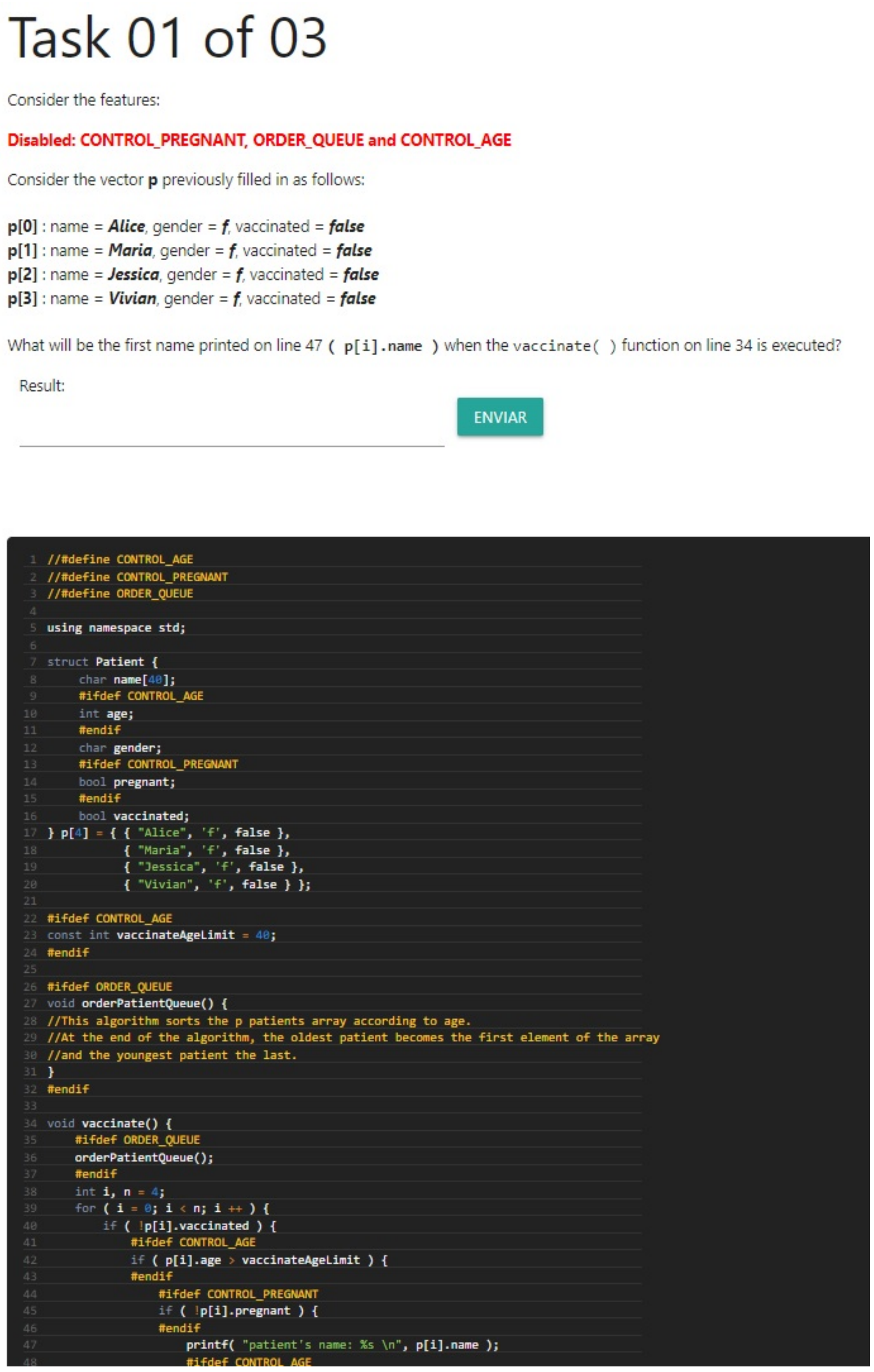
Eu concordo com os termos presentes no termo de consentimento livre e esclarecido: COMEÇAR

Figure 3.2 Home page of the online experiment.

For each task, participants had to read the question carefully and mentally execute

[1]http://www.facebook.com
[2]http://www.twitter.com
[3]http://www.reddit.com
[4]http://www.whatsapp.com

the code according to the proposed scenarios. Each scenario involves a configuration of enabled and disabled features.



Task 01 of 03

Consider the features:

Disabled: CONTROL_PREGNANT, ORDER_QUEUE and CONTROL_AGE

Consider the vector **p** previously filled in as follows:

p[0] : name = ***Alice***, gender = ***f***, vaccinated = ***false***
p[1] : name = ***Maria***, gender = ***f***, vaccinated = ***false***
p[2] : name = ***Jessica***, gender = ***f***, vaccinated = ***false***
p[3] : name = ***Vivian***, gender = ***f***, vaccinated = ***false***

What will be the first name printed on line 47 (`p[i].name`) when the `vaccinate( )` function on line 34 is executed?

Result:

```
 1 //#define CONTROL_AGE
 2 //#define CONTROL_PREGNANT
 3 //#define ORDER_QUEUE
 4
 5 using namespace std;
 6
 7 struct Patient {
 8      char name[40];
 9      #ifdef CONTROL_AGE
10      int age;
11      #endif
12      char gender;
13      #ifdef CONTROL_PREGNANT
14      bool pregnant;
15      #endif
16      bool vaccinated;
17 } p[4] = { { "Alice", 'f', false },
18            { "Maria", 'f', false },
19            { "Jessica", 'f', false },
20            { "Vivian", 'f', false } };
21
22 #ifdef CONTROL_AGE
23 const int vaccinateAgeLimit = 40;
24 #endif
25
26 #ifdef ORDER_QUEUE
27 void orderPatientQueue() {
28 //This algorithm sorts the p patients array according to age.
29 //At the end of the algorithm, the oldest patient becomes the first element of the array
30 //and the youngest patient the last.
31 }
32 #endif
33
34 void vaccinate() {
35     #ifdef ORDER_QUEUE
36     orderPatientQueue();
37     #endif
38     int i, n = 4;
39     for ( i = 0; i < n; i ++ ) {
40         if ( !p[i].vaccinated ) {
41             #ifdef CONTROL_AGE
42             if ( p[i].age > vaccinateAgeLimit ) {
43             #endif
44                 #ifdef CONTROL_PREGNANT
45                 if ( !p[i].pregnant ) {
46                 #endif
47                     printf( "patient's name: %s \n", p[i].name );
48                     #ifdef CONTROL_AGE
```

Figure 3.3 Example of how the Web system shows a task to the online experiment participants.

Different scenarios were chosen to force the participant to mentally simulate different configurations involving features and dependent variables. To ensure same difficulty level in all set of tasks, we defined the same three feature configuration scenarios for all four programs:

- Task 01: all features disabled.
- Task 02: all features enabled.
- Task 03: one feature disabled and two features enabled.

For example, the first task of each program (with and without feature dependency) should be answered considering all features disabled.

Figure 3.3 shows how the Web system shows the first task about Program 4. First, it informs the scenario of features the participant should consider:

Task 1: Consider the features:

Disabled: `CONTROL_PREGNANT`, `ORDER_QUEUE` and `CONTROL_AGE`

Then, it informs the data the participants should consider when analysing the program:

Consider the vector `p` previously filled in as follows:

`p[0]` : name = Alice, gender = f, vaccinated = false
`p[1]` : name = Maria, gender = f, vaccinated = false
`p[2]` : name = Jessica, gender = f, vaccinated = false
`p[3]` : name = Vivian, gender = f, vaccinated = false

Then, the system asks the question for the participant to answer:

"What will be the first name printed on line 37 (`p[i].name`) when the `vaccinate()` function on line 34 is executed?"

After the question, there is a field and a button for the participant to enter his or her answer. To answer the question, the participant has to mentally simulate the execution of the program to figure out the value of the variable, according to the configuration of the feature.

Finally, the system presents the entire source code of the Program (Figure 3.3 only shows part of the program due to space restrictions). Note that the first three lines of the program set the feature configuration scenario for that task. In the source code shown in Figure 3.3, the first three lines are commented, meaning that the three features are disabled, as defined in the description of the task.

After answering the three tasks of one of the programs in Domain 1, participants were redirected to one of the two programs in Domain 2, for them to answering another three tasks. The system randomized it for each participant based on Latin Square design described in Section 3.1.

We have presented each program to the participants as shown in Figure 3.3. We colored the source code to simulate the use of an IDE. Participants were instructed not to use tools or IDEs. We also hide parts of the code to avoid participants copying the source code and compiling it in an IDE.

Table 3.1 Mean time and number of attempts needed for the all participant until giving the correct answer for 3 tasks for each program

Program	Mean time for 3 tasks	Total number of attempts for 3 tasks
without dependency - Domain 1	135.84	106
without dependency - Domain 2	133.61	94
without dependency - all programs	134.78	200
with dependency - Domain 1	191.25	116
with dependency - Domain 2	228.21	86
with dependency - all programs	201.03	202

Finally, when a participant finished all the tasks, the online experiment asked her or him some questions to obtain qualitative feedback on how she or he performed the tasks. We have also asked participants about their profiles.

3.6 RESULT

We have measured comprehensibility according to: (i) time the participant took to finish the task with the correct answer, (ii) number of attempts needed for the participant until giving the correct answer. We ran ANOVA tests for hypothesis about time. We used p-value < 0.05 as the probability about rejecting null hypothesis. For the number of attempts, we used inferential statistics. We ran our tests with the support of R[5]. All artifacts used in our study 1 are available at our website[6] and our research share website.[7].

Ninety three participants started to respond the online experiment, but only 53 answered all the questions. In total, seven participants took too long to answer the questions and were considered as outliers by R analysis. Thus, we did not take their data into account. Therefore, our analysis was based on data from 46 participants.

3.6.1 Time participant took to finish tasks with correct answer

We measured the time that each participant spent (in seconds) to answer each task. Our null hypothesis about this metric is:

H_0*time*: There is no significant difference in the time participants took to finish the tasks with the correct answer when comparing programs with and without feature dependency.

We had 46 participants organized according our Latin square design. Consequently, we had 23 observations for each program on a specific domain, which summed up as a total of 92 observations. Shapiro test confirmed that the data about time to find bugs was normally distributed.

[5]http://www.r-project.org
[6]http://www.djansantos.com.br/projects/survey
[7]https://doi.org/10.5281/zenodo.7982409

Rows in Table 3.1 shows the mean time (column mean time) spent by participants for programs on each domain and for all programs with and without feature dependency. For example, they spent a mean time of 133.61 seconds to answer the questions without feature dependency in Domain 2. For programs with feature dependency in Domain 2, the mean time was 228.21 seconds. Our data revealed that there was a significant difference in time for the developers to analyze different types of source code. Considering all programs p-value = 5.9837-05. We, thus, reject our null hypothesis (H_0*time*).

Result 1: Programs containing feature dependencies required more time for the participant to answer the tasks correctly than programs without feature dependency.

3.6.2 Number of attempts needed until correct answer

The number of attempts to answer the tasks refers to how difficult it was for the participant to answer the tasks correctly. We counted all attempts each participant made until correctly answering each task. Our null hypothesis about this metric is:

H_0*attempt*: There is no significant difference in number of attempts needed for correctly answering tasks related to programs with feature dependency and programs without it.

Table 3.1 shows the number of attempts needed the participants for programs on each domain and for all programs with and without feature dependency. (column number of attempts). For example, participants made 106 attempts to give the correct answer for tasks of programs without feature dependency on Domain 1. For tasks about programs with feature dependency on Domain 1, the number of attempts of the all participants was 116 times.

Comparing programs with and without feature dependency, our data reveal that 23 participants made 200 attempts to finish tasks in programs without feature dependency. For programs with feature dependency, participants made 202 attempts. The χ^2 test (Pearson's Chi-squared test) (CAMILLI; HOPKINS, 1978) revealed no significant difference between programs with and without feature dependency. The value χ^2 is less than 5.53, and the p-values are greater than 0.35. Based on this, we cannot reject our null hypothesis H_0*attempt*.

Result 2: There was no significant difference on the number of attempts needed for the participants until giving the correct answer when comparing programs with and without feature dependency.

3.6.3 Discussion

Here we answer our research question *How does feature dependency affect the comprehensibility of configurable system?*

Feature dependency affects comprehensibility in programs with #ifdef.

Result 1 shows that programs with feature dependency demanded more time the participant took to finish the task with the correct answer. Developers spent more time when compared with programs without feature dependency. We hypothesize that this would occur because feature dependency forces developers to worry more with dependent variables and makes it difficult to simulate different configurations of enabled/disabled features.

The use of #ifdef hinders comprehensibility while debugging dependent variables. Multiple researches also indicate that #ifdef hinder comprehensibility (SPENCER; COLLYER, 1992),(LE; WALKINGSHAW; ERWIG, 2011). However, our results indicate that feature dependency increases the problem.

Feature dependency did not affect the number of attempts needed for the participant until giving the correct answer.

Result 2 revealed that the number of correct specifications on the output was not affected by features dependency. This means that feature dependency may increase time to specify the output, but do not decrease developers ability to specify the output. This result confirms previous studies that showed that most participants correctly performed tasks in programs with #ifdef (MELO et al., 2017; SANTOS; SANT'ANNA, 2019).

3.7 THREATS TO VALIDITY

3.7.1 Internal validity

Programming language. We wrote our programs in C, because conditional compilation directives in C are native and are one of the most popular mechanisms in use. The knowledge in C could influence our results. To minimize that, we only admitted participants with previous experience on C.

Participants' experience. We only had two sets of tasks. Some participants were experts in #ifdefs, and they were randomly distributed according to our Latin square. We distributed participants into the Latin square sets. The order of tasks was randomized according to our Latin square design. The web system distributed the tasks according to the order in which participants accessed the site. So, we controlled confounding factors via the Latin square design and randomization.

Code access. The programs were displayed in a web page. All participants accessed the programs through the browser of their choice. Some pieces of source codes have been omitted to inhibit participants from copying the source code into an IDE. If any of them tried to do that, it would certainly have taken a considerable amount of time leading it to be considerad as an outlier.

3.7.2 External validity

Real systems and features. Due to our limitations we have made the use of small programs. But, our programs were inspired on real configurable systems. For this reason, our results may hold to other programs. However, for programs over 70 lines of code and more than 3 features, there may be additional effects that we have not observed.

Meaningful variables names. The maintenance and comprehensibility of source code can be hindered by choosing variables with non-meaningful names. Choosing meaningful variable names is also important for domain comprehension. To minimize this threat, we select relevant and meaningful variable names.

Mental simulation of scenarios. In practice, programmers encounter codes containing many features, and they don't always test or compile all possible configurations. Our results are limited to programs with a few features where the developer mentally simulates all possible configurations. Additionally, we utilize functional features, which are easier to analyze than architectural features, for example.

Lab settings. Our results are also limited to the environment we have adopted. A more realistic environment, with IDEs and source code with multiple files, would be ideal. However, this design would not be attractive for many participants, since it would require more time for execution.

Volunteer bias. We recruit participants both by personal invitations and via social media to avoid volunteers different from the target population. Furthermore, we ensured anonymity and confidentiality of volunteers in order to try to increase participation, and thus, decrease volunteer bias.

3.7.3 Construct validity

Comprehensibility measurement. Measuring comprehensibility is not trivial because it involves human factors. Therefore, it is always a threat to construct validity. To minimize this threat, we quantified comprehensibility by means of time, frequently used in previous studies (MELO et al., 2017).

Misinterpret task Respondents can misinterpret the tasks. To reduce this risk we included an instruction page with details of how execute the tasks.

Chapter

4

STUDY 2 - AN EXPERIMENT ON HOW FEATURE DEPENDENCY TYPES AFFECTS PROGRAM COMPREHENSIBILITY

In this chapter, we present the experiment that aims at evaluating the impact of different types of feature dependency on the comprehensibility of configurable systems. If a specific type of dependency makes it more difficult to understand the source code, configurable system developers should take care when either maintaining or testing code fragments with such type of dependency. In this context, the following research question guided our study:
-- *How do different types of feature dependency affect the comprehensibility of configurable system source code?*

4.1 DESIGN

To answer our research question, we carried out a controlled experiment with 30 developers, who analyzed programs trying to find bugs. They analyzed different programs each with a different type of feature dependency (global, intraprocedural and interprocedural). We compared the comprehension effort they spent to analyze each program. We quantified comprehension effort from different perspectives: (i) time to analyze each program, (ii) number of correctly found bugs, and (iii) visual effort. We quantified visual effort by means of different metrics collected by the use of an eye-tracking device. We give more details about these metrics in Section 4.5.

We also aimed to confirm whether the differences on comprehension effort were due to the different feature dependency types (implemented with #ifdefs) or were only due to differences related to variable scope, such as the use of global variables or local variables, regardless of the use of #ifdefs. To achieve this, we also asked the developers to analyze programs without #ifdefs, but equivalent to the ones with #ifdefs. For each program with #ifdef, we have an equivalent one without #ifdef. They are equivalent mainly in two aspects. First, in the programs without #ifdef, we replaced the #ifdefs with regular *if* statements. Second, the program without #ifdef follow the same structure in terms of variable use of its equivalent with #ifdef. For instance, a program with intraprocedural dependency has an equivalent with the same function and local variable, but with *ifs* as replacements of #ifdefs. In Section 4.3, when describing the bugs we used in our experiment, we give an example of two equivalent programs with and without #ifdefs.

In order to avoid learning effect, we selected six different variability bugs to compose each program: (i) null pointer dereference, (ii) assertion error, (iii) variable overlap, (iv) nested feature, (v) undefined variable, and (vi) uninitialized variable. Related literature has reported these bugs as recurring in configurable systems (ABAL; BRABRAND; WASOWSKI, 2014; MEDEIROS et al., 2015; BRAZ et al., 2016). Section 4.3 describes each of them. Also to avoid learning effect, we had programs on six different domains, each domain for a variability bug. For example, we implemented the null pointer dereference variability bug in a program on the "sales authorization message" domain.

In summary, for each variability bug, we implemented six similar programs on the same domain, each one with the following characteristics: (i) global dependency with #ifdef (GI), (ii) equivalent to global dependency without #ifdef (GW), (iii) intraprocedural dependency with #ifdef (IAI), (iv) equivalent to intraprocedural dependency without #ifdef (IAW), (v) interprocedural dependency with #ifdef (IEI) and (vi) equivalent to interprocedural dependency without #ifdef (IEW). Therefore, we implemented 36 programs.

Table 4.1 Latin square design

	Variability Bugs					
Group	Null pointer	Assertion error	Logic error	Nested feature	Undefined variable	Uninitialized variable
G1	GI	GW	IAI	IAW	IEI	IEW
G2	GW	IAI	IAW	IEI	IEW	GI
G3	IAI	IAW	IEI	IEW	GI	GW
G4	IAW	IEI	IEW	GI	GW	IAI
G5	IEI	IEW	GI	GW	IAI	IAW
G6	IEW	GI	GW	IAI	IAW	IEI

We designed our experiment as a standard Latin square, which is a common solution for this kind of experiment (BOX; HUNTER; HUNTER, 2005; BAILEY, 2008; MELO et al., 2017). We can explain the 6x6 Latin square we adopted by means of Table 4.1. In its columns, we have the variability bugs. The lines represent the groups of developers. We divided the developers in six groups (G1 to G6). The acronyms in the cells represent the program characteristics, as listed in the previous paragraph. For instance, developers in group G1 analyzed the program with *global dependency with #ifdef (GI)* that has the *null pointer dereference bug* (first column of G1 line). Developers in group G2 also analyzed the program with *global dependency with #ifdef (GI)*, but the one with the *uninitialized variable bug (VI)* (three last columns of G2 line).

The Latin square design controls one factor and its variations, ensuring that no row or column contains the same treatment twice. Our factor was the program characteristic (type of dependency + with/without #ifdef). Each line of our Latin square has six programs with different characteristics arranged in different orders. However, there might still be learning effects due to repetition of variability bugs. We avoided this by distributing the variability bugs along our Latin square columns. Each column has a different type of variability bug (without repetition). Therefore, according to his/her designated group, each developer debugged six different programs, each with a different variability bug (and its respective domain) and each with a different type of feature dependency (with and without #ifdef). The result is the same number of data points for all debugging tasks. As we had 30 participants, we obtained 180 data points, five data points for each of the 36 programs.

4.2 PARTICIPANTS

We counted on 30 participants to run our experiment: six undergraduate students, six MSc students, six PhD students, six professors, and six developers from industry. We put them in six groups with five participants each. Randomly, we formed each group with one participant of each category, i.e., one undergraduate student, one Msc student and so forth.

We selected the students from three universities located at three different cities of Brazil and the developers from two companies located at two different cities of Brazil. No compensation was provided for the participants.

Out of these 30 participants, one participant's eye tracking data were discarded due to poor quality. This was due to technical issues with the eye tracker. Eighteen participants have normal vision and 12 have vision corrected by glasses. Seven participants are females and 23 are males. All of them have similar experience in C programming language. Sixteen participants declared themselves as expert developers and only three claimed that had not worked at the industry yet.

4.3 VARIABILITY BUGS

When a fault or error happens in a configurable system due to variability implementation, it is called as variability bug (ABAL; BRABRAND; WASOWSKI, 2014). Previous studies related variability bugs to feature dependencies (ABAL; BRABRAND; WASOWSKI, 2014; RIBEIRO; BORBA; KÄSTNER, 2014; MEDEIROS et al., 2015; BRAZ et al., 2016). We implemented the programs used in our experiment inspired in concrete variability bugs, which occurred in real large-scale configurable systems (ABAL; BRABRAND; WASOWSKI, 2014; MELO et al., 2017; ABAL et al., 2018): Linux (ABAL; BRABRAND; WASOWSKI, 2014; ABAL et al., 2018), BusyBox (MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017), BestLap (RIBEIRO; BORBA; KÄSTNER, 2014), GLib (RODRIGUES et al., 2016) and Libxml (RODRIGUES et al., 2016). However, we could not use source code of large-scale systems in our study due to the following restrictions:

Domain. Our programs should be on domains which participants could easily understand. We avoided programs (like the ones from Linux) which could affect comprehensibility.

Native language. To facilitate the understanding and to widen the audience of potential participants, the programs should be written in the participants' native language (Portuguese).

Small programs. The programs should fit on a 39-line display window so that the eye-tracking device would record all gaze movements of participants.

Having these restrictions in mind, we took an example of each selected variability bug from a real configurable system or from previous studies. Then, we reproduced that bug in a small program, on a popular domain, written in C with variable and feature names in the participants' native language.

The 36 programs are similar in terms of number of lines of code (LOC) (LANZA; MARINESCU, 2007), number of features (NOFC) (LIEBIG et al., 2010) and McCabe cyclomatic complexity (CC) (MCCABE, 1976). In the following, we describe each variability bug we implemented.

Listing 4.1 Code snippet of the null pointer dereference bug with #ifdef

```c
 1 char *p = NULL;
 2 char message[25];
 3 ...
 4 int main() {
 5 char msg[25];
 6 ...
 7 #ifdef CUSTOMIZE_MESSAGE
 8     strcpy(message,"Store XYZ - ");
 9     p = message;
10 #endif
11 if(*p =="")
12     strcpy(message, msg);
13 ...
```

Null pointer deference. This bug happens when a program attempts to read a value from a null pointer. Listing 4.1 shows a code snippet of the program we wrote with a null pointer dereference bug. It is about "printing a sales authorization message". It has two features: `CUSTOMIZE_MESSAGE` and the feature `SETUP_COMMUNICATION`. `CUSTOMIZE_MESSAGE` personalizes the message with the name of the store and `SETUP_COMMUNICATION` checks communication errors. An exception occurs when `CUSTOMIZE_MESSAGE` is disabled. It happens because `p` is updated within `CUSTOMIZE_MESSAGE`. The expression `if(*p =="")` (line 12) causes a null-pointer exception because `p` has null value and cannot be compared with empty. We wrote this program taking as example a bug found in BusyBox (MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017), an open source system that provides essential Unix tools.

Listing 4.2 shows the version of the program with null pointer dereference bug now without #ifdefs. We rewrote the program showed in Listing 4.1 by replacing #ifdefs with `if` clauses. The bug remains the same in Listing 4.2, as line 12 executes the expression `if(*p =="")`, which causes a null-pointer exception. However, in this case, the bug is caused by a variable whose value is false, and not by disabling a feature. It is important to recall that Listing 4.1 shows the program with global dependency with #ifdef and Listing 4.2 shows its equivalent without #ifdef. Besides them, we also had in our experiment other four programs with the null pointer dereference bug, which implement intraprocedural and interprocedural dependencies and its equivalent without #ifdefs. For the bugs we describe next, we only show one of the programs, the one with #ifdef.

In some of the programs, there are #ifdefs surrounding variable definitions (for instance, Listing 2.2, lines 2 to 4), or #ifdefs surrounding `else` clauses (for instance, Listing 4.3, lines 11 to 13). In this cases, we just deleted the #ifdefs and did not replace them with `if` clauses. We did this because, in fact, converting those #ifdefs into `ifs` does not make sense.

Listing 4.2 Code snippet of the null pointer dereference bug without #ifdef

```c
char *p = NULL;
char message[25];
bool customize_message = false;
int main() {
char msg[25];
...
if (customize_message){
    strcpy(message,"Store XYZ - ");
    p = message;
}
if(*p =="")
    strcpy(message, msg);
...
```

Assertion-Error. An Assertion-Error occurs when something that should never happen happens. This bug was found in BestLap, which is a commercial highly configurable race game, investigated by previous research (RIBEIRO et al., 2012; RIBEIRO; BORBA; KÄSTNER, 2014; MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017). The car racing game calculates score of players and adds a penalty when their cars crash. As the score should not be negative, the assertion error occurs when the score stores negative values. Based on BestLap, we wrote a program that "calculates scores in language proficiency tests". We apply penalties in case the exam time is exceeded. Listing 2.1 (Section 2.3) shows a code snippet of one of the programs with this bug. It has two features: `APPLY_PENALTY` and `RESULT`. `APPLY_PENALTY` applies penalties in case the exam time is exceeded, and `RESULT` prints the final result. The bug occurs when both `APPLY_PENALTY` and `RESULT` are enabled and `final_grade` is negative. In this case, instead of generating an error, the program prints a wrong message (line 17).

Logic error: Logic error occurs when a program produces unintended or undesired output. Logic errors are often the most general errors and hardest to identify (HRISTOVA et al., 2003). Listing 4.3 shows an excerpt of our program that "calculates the value of payments if the customer decides to purchase in 3 installments". The customer can use credit card or checks to split a sale. If he or she chooses checks, the program adds an interest of 5% to the purchase value. If the sale is paid on a credit card, the program does not add any interest. The customer also may choose not to split the purchase. The `CREDIT_CARD` and `CHECK` features calculate instalment values without and with interest, respectively. Note that both features enable the program to split the purchase in three installments, but the purchase will be split whether the value of `use_card` (line 5) or `use_checks` (line 9) is true. The logic error occurs when `CREDIT_CARD` and `CHECK` are enabled. When the two features are enabled, the clause `else` (line 13) is associated with the scope of clause `if` of `use_checks` only. Thus, if `use_card` (line 5) is true and `use_checks` (line 9) is false, the value of `instalments` is overlapped.

Listing 4.3 Code snippet of the logic error

```
 1 int main() {
 2 ...
 3 #ifdef CREDIT_CARD
 4 if (use_card)
 5     instalments = purchase/3;
 6 #endif
 7 #ifdef CHECK
 8 if (use_checks)
 9     instalments=(purchase + (purchase * 0.05))/3;
10 #endif
11 #ifdef CREDIT_CARD || CHECK
12 else
13 #endif
14 instalments = purchase;
15 ...
```

Listing 4.4 Code snippet of the undefined variable bug

```
 1 ...
 2 float calculateRegistrationFee() {
 3 #ifdef STUDENT_DISCOUNT
 4     bool apply_discount_student = true;
 5 #endif
 6 ...
 7 #ifdef STUDENT_DISCOUNT || MEMBERSHIP_ASSOCIATION
 8     float discount=0;
 9 #endif
10 #ifdef STUDENT_DISCOUNT
11     if (apply_discount_student)
12         discount += 0.1;
13 #endif
14 ...
15 registration = registration - (registration * discount);
16 ...
```

Nested features. Previous studies (LIEBIG et al., 2010) reported that nesting features make the source code prone to errors (SINGH; GIBBS; COADY, 2007; LIEBIG et al., 2010). Listing 2.2 (Section 2.3) shows an excerpt of our program where nested features cause a variability bug. The program processes the "return of books to a library". If a book is returned after the due date, the system applies either a fine or a penalty in days during which the user will be unable to make new loans. `FINE_IN_CASH` calculates the fine (line 11) and `PUNISHMENT_IN_DAYS` calculates the penalty in days. Note that

`PUNISHMENT_IN_DAYS` (line 12) is inside `FINE_IN_CASH` (line 10). The bug happens when `FINE_IN_CASH` is disabled, because doing this also disables `PUNISHMENT_IN_DAYS`.

Undefined variable. This bug happens when a variable is not previously declared but it is accessed later on. To write our program we took as example a bug found in Libxml[1], a configurable system for parsing XML files (RODRIGUES et al., 2016). Listing 4.4 presents a code snippet of our program with this variability bug. It "calculates the registration fee of an event". The program applies discounts for students or attendees with membership association. The program has two features called `STUDENT_DISCOUNT` and `MEMBERSHIP_ASSOCIATION`. `STUDENT_DISCOUNT` applies a discount for students (lines 3, 7 and 10). `MEMBERSHIP_ASSOCIATION` applies a discount for attendees with a membership association (line 7). The bug happens because the `discount` variable is defined only if either `STUDENT_DISCOUNT` or `MEMBERSHIP_ASSOCIATION` is enabled (line 8). If both features are disabled, `discount` is undefined and the program runs into an undefined variable error, because `discount` is used ahead (in line 15).

Uninitialized variable. This bug happens when a variable its value is declared but is not set before its use. Abal *et al.* show this is a frequent bug in Linux kernel (ABAL; BRABRAND; WASOWSKI, 2014; ABAL et al., 2018). Our program with this bug "calculates points based on a students' curriculum" (Listing 2.3 (Section 2.3)). The number of points of a student increases with the number of courses he or she accomplished. When `CONSIDER_SPECIALIZATION` is enabled, `total_points` receives one extra point (line 8), if `isSpecialist` is true. The variability bug occurs because `total_points` is not initialized before (line 14).

4.4 EXPERIMENT PROCEDURES

Before executing the actual experiment, we carried out three pilot studies. In the first one, in addition to find bugs, we asked participants to fix them. However, they took so long to do that. Moreover, we had problems to record their gaze movements as our eye-tracking device does not work well on non-static screens. So, we changed our experiment procedures for the second and third pilot studies: the participants only had to find bugs analyzing a small program showed in a static image, which they could not manipulate. The second and third pilot studies were mainly useful for us to correct problems with the programs. We performed the pilot studies with four PhD students. We did not consider their results in our analysis.

Before starting the experiment tasks, we briefly trained every developer on conditional compilation, variability, features and system configuration. Then, we calibrated the eye-tracking device and performed a small warm-up task. All the participants signed a consent form.

The participants debugged the programs as we planned in our Latin square design. When the participant indicated he or she finished analyzing a program, we registered the time. The time was paused at the moment the participant reported to have found the error. We did not consider the time the participant took to respond in our analyses. The time was paused and recorded in the tool that records the gaze data. Finally, to check

[1]http://xmlsoft.org/

whether he or she correctly found the bug, we asked each developer: "how would you fix the bug you found?". For the purpose of analyzing the results, we considered only the times of the participants who correctly answered the task.

We presented each program to the participants as static images displayed on a single screen. Participants did not have access to tools, IDEs or browsers. For each participant, we recorded `x` and `y` coordinates (fixations) via an eye tracker.

We performed each experiment trial individually. All participants used the same personal computer to avoid unintended effects from different software and hardware environments. The computer has the following configuration: a 64-bit windows 10 home single language with Intel core i5. The screen resolution was set to 1920 by 1080 pixels into a 15 inch LCD screen. All experiment trials were conducted in similar classrooms. We recorded all of the eye tracking data using the open-source tool OGAMA (VOSSKÜHLER et al., 2008). We used the Tobii EyeX Device.

4.5 EXPERIMENTAL RESULTS

In this section, we test our hypotheses and discuss results. We measured comprehensibility according to: (i) time to find a bug, and (ii) number of correctly found bugs. We also measured developers' visual effort: (i) number of fixations, and (ii) gaze transitions. We also analyzed attention maps generated by means of the eye tracker.

We ran ANOVA tests for hypothesis testing. We used p-value < 0.05 as the probability about rejecting null hypotheses. The only exception, was the *number of found bugs* variable. ANOVA does not apply for it as it holds binary values. Thus, we used inferential statistics to evaluate it. We ran our tests with the support of R[2]. All artifacts used in our experiment are available at our website[3] and our research share website.[4] In the following, we present the results regarding each metric.

4.5.1 Time to find bugs

Table 4.2 Time results

	Variability Bugs											
	Null pointer		Assertion error		Logic error		Nested feature		Undefined variable		Uninitialized variable	
Group	Char.	time	Char.	time	Char.	time	Char.	time	Char.	time	Char.	time
G1	GI	338	GW	287	IAI	292	IAW	216	IEI	252	IEW	203
G2	GW	418	IAI	366	IAW	258	IEI	267	IEW	212	GI	182
G3	IAI	315	IAW	262	IEI	224	IEW	281	GI	123	GW	101
G4	IAW	383	IEI	339	IEW	224	GI	412	GW	362	IAI	234
G5	IEI	414	IEW	225	GI	194	GW	206	IAI	324	IAW	148
G6	IEW	461	GI	353	GW	390	IAI	287	IAW	288	IEI	168

We measured the time (in seconds) each participant took to analyze each program,

[2]http://www.r-project.org/
[3]http://www.djansantos.com.br/projects/FeatureDependency/
[4]https://doi.org/10.5281/zenodo.7982409

Table 4.3 Mean time and found bugs for each types of dependency

Dependencies	**GI**	**GW**	**IAI**	**IAW**	**IEI**	**IEW**
Mean time	323.10	228.70	255.10	235.00	322.50	287.20
#found bugs	21	24	23	24	17	23

similarly as Sharif et al. (SHARIF et al., 2013) did. Our null hypothesis about this metric is:

H_0t: There is no significant difference in the time to find bugs when comparing programs with different types of feature dependency.

Rows in Table 4.2 show the mean time (column time) spent by each group of participants (G1 to G6) for each characteristic of program and for each variability bug. Each group was compose by 5 participants. Consequently, we had 5 observations for each group, 30 observations for each row, which summed up as a total of 180 observations. Shapiro test confirmed that the data about time to find bugs was normally distributed.

Table 4.3 shows the mean time spent by all participants for each type of dependency. For example, they spent a mean time of 323.10 seconds to analyze programs of global dependency with #ifdefs (GI). For global dependency without #ifdefs (GW), the mean time was 228.70 seconds. Our data revealed that there was a significant difference in time for the developers to analyze different types of dependency (p-value = 5.613e-05). We, thus, reject our null hypothesis (H_0t).

We, then, used Tukey HSD (honestly significant difference) test to find means of time that are significantly different. Tukey HSD test compares all possible pairs of means of time. First, we compared the mean time related to programs with #ifdefs (GI, IAI and IEI) (Table 4.3). The difference of time between GI and IEI is negligible (p-value = 1.00). In contrast, when comparing IAI with GI and IEI, the difference is significant (p-value = 0.045 and 0.049, respectively). This leads to our first result.

Result 1: Global dependencies and interprocedural dependencies required more time for finding bugs than intraprocedural dependency.

We also compared the time spent for analyzing the three types of programs without #ifdefs (GW, IAW and IEW) (Table 4.3). According Tukey HSD test, the time to analyze each of them is not significantly different (all p-values are larger than 0.126). This, somehow, reinforces that the differences stated in Result 1 are due to the use of #ifdefs.

Curiously, our data also revealed that the mean time is not significantly different when we compare programs with intraprocedural dependency with and without #ifdefs (IAI vs. IAW) (p-value = 0.961). This leads to our second result, which, regarding a specific context, contradicts previous study that says that #ifdefs increase debugging time (MELO et al., 2017).

Result 2: The use of #ifdef did not increase bug detection time in programs with intraprocedural dependency characteristic.

We also compared programs with interprocedural characteristic with and without #ifdefs (IEI vs. IEW). Similarly to intraprocedural programs, our results showed negligible difference in time for bug detection (p-value = 0.657).

We also compared global dependency with and without #ifdefs (GI vs. GW) (Table 4.3). In this case, the difference in bug detection time is significant (p-value = 0.001). Based on this, we state our third result.

Result 3: The use of #ifdefs increases bug detection time for programs with global dependency characteristic.

4.5.2 Number of correctly found bugs

Table 4.4 Hits results

	Variability Bugs											
	Null pointer		Assertion error		Logic error		Nested feature		Undefined variable		Uninitialized variable	
Group	Char.	hits	Char.	hits	Char.	hits	Char.	hits	Char.	hits	Char.	hits
G1	GI	3	GW	3	IAI	5	IAW	4	IEI	3	IEW	3
G2	GW	3	IAI	5	IAW	5	IEI	1	IEW	4	GI	5
G3	IAI	1	IAW	5	IEI	3	IEW	5	GI	5	GW	5
G4	IAW	1	IEI	4	IEW	4	GI	2	GW	5	IAI	5
G5	IEI	1	IEW	4	GI	2	GW	5	IAI	3	IAW	4
G6	IEW	3	GI	4	GW	3	IAI	4	IAW	5	IEI	5

The number of correctly found bugs metric refers to whether a participant answered each task correctly. It is about correctness (SHARIF et al., 2013). If a participant finds a bug correctly, he or she scores one for that program. For this, we registered the answers provided by participants. Our null hypothesis about this metric is:

H_0a: There is no significant difference in number of correctly found bugs when comparing programs with different types of feature dependency.

Table 4.4 shows the number of bugs correctly found (column hits) by each group of participants (G1 to G6) for each characteristic of program and for each variability bug. Table 4.3 shows the number of correctly found bugs by all participants for each type of dependency.

Comparing programs with #ifdefs, from a total of 30 tasks, 21 participants answered correctly the tasks about global dependency (GI), 23 answered correctly for intraprocedural dependency (IAI) and 17 for interprocedural dependency (IEI). Interprocedural dependency, therefore, seems to make bug detection more difficult. However, the χ^2 test (Pearson's Chi-squared test) (CAMILLI; HOPKINS, 1978) revealed no significant difference between the three types of feature dependency. The value χ^2 is less than 5.53, and the p-values are greater than 0.35. Based on this, we cannot reject our null hypothesis H_0a.

Result 4: There was no significant difference on the number of correctly found bugs for different types of feature dependency.

4.5.3 Visual effort

Visual effort is directly linked to the cognitive effort (RAYNER, 2009; SIEGMUND et al., 2014; SHARIF et al., 2013). A set of eye-tracking measures representing visual effort are derived from eye gaze data. A fixation is a type of eye movement in which the eye stops on some object of interest to obtain information. Saccades are very fast voluntary movements between fixings. Regression is a saccade performed in the opposite direction to the reading direction (RAYNER, 2009).

The number of fixations is thought to be negatively correlated with search efficiency (GOLDBERG; KOTVAL, 1998). Also, the proportion of time looking at a particular display element could reflect the importance of that element (KOLERS; DUCHNICKY; FERGUSON, 1981; JACOB; KARN, 2003). The literature sets a duration of 60 microseconds as the minimum threshold for having a fixation. Also, it sets a space of seven to nine letters for characterizing saccades (RAYNER et al., 2006). We followed these thresholds and discarded anything below them (KOLERS; DUCHNICKY; FERGUSON, 1981; RAYNER, 1998; DUCHOWSKI, 2017).

Number of fixations

The number of fixations increases when a text is difficult to comprehend (RAYNER et al., 2006). We counted the number of fixations per program. Our null hypothesis about this metric is:

H_0f: There is no significant difference in number of fixations when developers try to find bugs in programs with different types of feature dependency.

Considering programs with #ifdef, our data revealed that the number of fixations were significantly different between programs with different types of feature dependency (p-value = 5.613e-05). Analyzing programs with global dependency (GI) required from all participants a mean of 1011.93 fixations. For intraprocedural dependency (IAI), it required a mean of 741.56 fixations, and for interprocedural dependency (IEI), the mean was 1016.67 fixations.

Tukey HSD showed that the difference between global dependency (GI) and interprocedural dependency (IEI) is negligible (p-value = 0.99). In contrast, the number of fixations in programs with global and interprocedural dependencies is significantly higher than the number of fixations in programs with intraprocedural dependency (IAI) (p-value = 0.008 and 0.006 respectively).

Result 5: Developers made more fixations to understand programs with global and interprocedural dependencies than programs with intraprocedural dependencies.

In addition, when considering programs without #ifdefs (GW vs. IAW vs. IEW), our data shows that there is no significant difference among them in terms of number of fixation (all p values are larger than 0.9127183). This somehow reinforces that the differences stated in Result 5 are due to the use of #ifdefs and not due to the characteristic of the programs in terms of use of variables and functions.

Our study also revealed that the number of fixations is not significantly different when comparing data obtained for programs with intraprocedural characteristic with and without #ifdefs (IAI vs. IAW) (p-value = 0.86). This leads to the following result, which reinforces Result 2.

Result 6: The use of #ifdefs did not increase the number of fixation when developers try to find bugs in programs with intraprocedural dependency characteristic.

Finally, our results indicate that, regarding global and interprocedural characteristics, programs with #ifdefs required more number of fixations than the ones without #ifdefs (p-value = 2.1e-06 and 1.93e-04, respectively).

Result 7: The use of #ifdefs increased the number of fixations in programs with global and interprocedural characteristics.

Gaze transitions

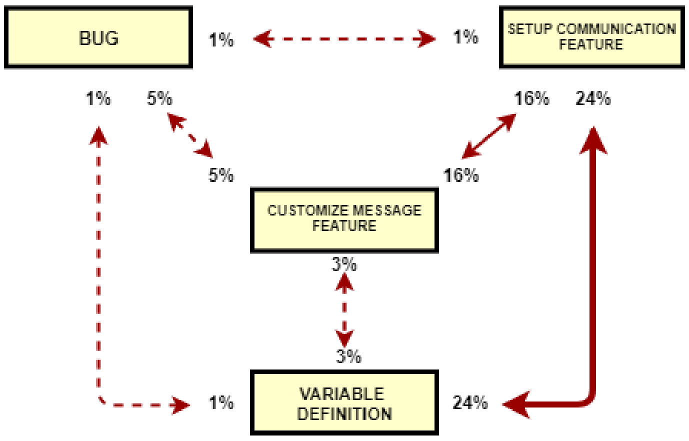
Gaze transitions (a.k.a. saccades) are rapid eye movements from one place to another separated by pauses (RAYNER et al., 2006). A larger number of saccades in both directions indicates difficulty associated with understanding (RAYNER, 1998; RAYNER et al., 2006; RAYNER, 2009).

We compute gaze transitions based in areas of interest (AOI) of each program. An AOI is a region of interest in a study. We defined the AOIs with OGAMA's AOI editor. Our AOIs comprised regions of source code that showed variable definitions, feature code and bug regions. For example, in Listing 4.1, we define variables in lines 1 and 2. Thus, we defined this region as the AOI "variable definition". Lines 7 to 10 comprise source code of `CUSTOMIZE_MESSAGE` feature. We defined this AOI as "customize message feature". The same happens with the `SETUP_COMMUNICATION` feature (not shown in Listing 4.1). We defined its source code as AOI "setup communication feature". Finally, the AOI "Bug" is limited by lines 11 and 13, because it is where the bug occurs. The definitions of AOIs for each of our programs are detailed in our website.

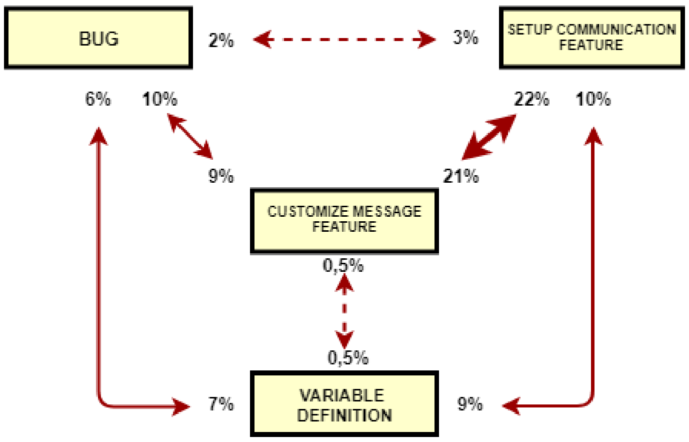
Figure 4.1 shows our gaze transitions diagrams related to programs with the "null pointer dereferenced" bug. Arrows indicate the percentage of gaze transactions between different AOIs of the program in both directions. For example, Figure 4.1a depicts that 5% of all gaze transitions performed by participants to find the "null pointer dereferenced" bug happened from the AOI "bug" to the AOI "customize message feature". A dashed arrow indicates that the sum of gaze transactions in both directions corresponds to 10% or less of the total of gaze transactions. A bold arrow indicates that the sum of gaze transactions corresponds to 40% or more of the total of gaze transactions. A regular line indicates intermediate values.

Figure 4.1a shows the diagram for the program with intraprocedural dependency with #ifdef (IAI), Figure 4.1b shows the diagram for the program with global dependency with #ifdef (GI) and Figure 4.1c shows the diagram for the program with interprocedural dependency with #ifdef (IEI). The three are related to the "null point dereferenced" bug. We took this bug as an example for what similarly happens for other cases.

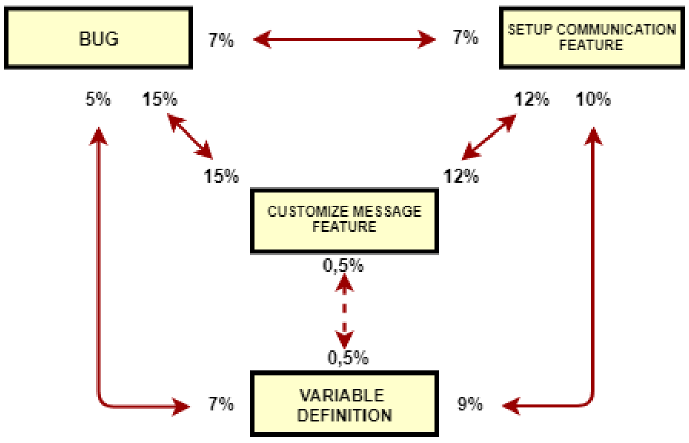
The gaze transitions diagram reveals that, for programs with global and intraprocedural dependencies, participants concentrated the largest number of saccades between few AOIs. Figure 4.1a shows a concentration of about 48% (24% + 24%) of gaze transitions between the AOIs "variable definition" and "setup communication feature". Figure 4.1b shows a concentration of about 43% (21% + 22%) of gaze transitions between AOIs



(a) intraprocedural



(b) global



(c) interprocedural

Figure 4.1 Gaze transitions between AOIs for different types of feature dependency in programs with the null pointer dereferenced bug.

"customize message feature" and "setup communication feature". On the other hand, Figure 4.1c reveals that that participants, when debugging programs with interprocedural dependency, need to navigate over all source code, and, as a consequence, the gaze transitions are more distributed between all AOIs. Note in Figure 4.1c that values do not exceed 30%. Although we only show here the diagrams for the "null pointer dereferenced" bug, we observed similar results for most of the other bugs.

Result 8: Interprocedural dependency seems to force the developer to perform more gaze transitions over different parts of the source code.

Attention map

An attention map is a histogram, also known as heat map, that displays an aggregation of fixations. An attention map shows how attention is distributed among program parts (ŠPAKOV; MINIOTAS, 2007). It uses colors to represent the fixation time in each location on the screen. Three examples are shown in Figure 4.2. The lowest value in the attention map (short fixation time) is shown with the green color and the highest value in red (long fixation time), with a smooth transition between these extremes.

Figure 4.2 shows the aggregated attention maps for the "assertion error" variability bug for the three types of feature dependency. We generated the aggregated attention maps using OGAMA. We superimposed all individual attention maps from each participant. Each attention map of the "assertion error" variability bug (Figure 4.2) is, thus, composed by the overlapping of five individual attention maps.

The red regions indicate where most of participants' attention was directed to. Comparing the red regions of the three attention maps in Fig 4.2, we observe that the attention distribution is similar for programs with global and intraprocedural dependencies. In these cases, participants focused most of their attention in the source code of the `APPLY_PENALTY` feature. The bug occurs inside `RESULTS`, when `APPLY_PENALTY` is enable. Thus, this area requires more attention from participants. In addition. in both programs, `APPLY_PENALTY` is near `RESULTS`.

On the other hand, in the attention map regarding interprocedural dependency (Figure 4.2c), there are two distinct red regions. One region encapsulates the source code of `APPLY_PENALTY` and the other red region is about `RESULTS`. In this case, however, the two regions are far from each other. Participants need, thus, to focus on each region separately. This leads to the following result.

Result 9: Interprocedural dependencies appear to increase fixation time in more distinct areas of the source code.

In the right side of Figure 4.2, we also show gaze transition diagrams related to each attention map. These diagrams depict the first three minutes of an "assertion error variability" debugging task performed by a participant. The diagram shows the y-coordinate that the participant was looking at as a function of time. The top of the diagram corresponds to the first line of the program and the bottom corresponds to the last line. We can observe in these diagrams that the initial scan the participant performed on the program is relatively similar for programs with global and intraprocedural dependencies. Normally, developers perform a preliminary reading of the source code in a very fast pace and then start reviewing the source code afterward. Concerning the source codes with #ifdefs, participant appears to prolong the preliminary reading in programs with global and intraprocedural dependencies. Similar behavior was reported in previous studies (UWANO et al., 2006; MELO et al., 2017).

In contrast, Figure 4.2c shows that, in the program with interprocedural dependency, the participant performed a very fast preliminary reading and, then, continued with the mental simulation of the `main()` function. This reinforces our hypothesis that participants navigated through more distinct areas of the code in programs with interprocedural dependency.

4.5.4 Discussion

Here we answer our research question *How different types of feature dependency affect the comprehensibility of configurable systems?* by discussing different aspects of our findings.

#ifdefs affect comprehensibility in different degrees according with the type of feature dependency.

Our results 1, 3, 5 and 7 show that programs with global or interprocedural dependencies demand more comprehension effort for finding bugs. Developers spent more time and more number of fixations, when compared with programs with intraprocedural dependency. We hypothesize that this occurs because with both global and interprocedural dependencies, the point of definition of the dependent variable is far from its usage. Our results also show that the use of #ifdef hinders comprehensibility while debugging. Multiple researches also indicate that #ifdef might amplify maintenance problems (ERNST; BADROS; NOTKIN, 2002; GARVIN; COHEN, 2011; RIBEIRO et al., 2012; MEDEIROS et al., 2015; FENSKE; SCHULZE; SAAKE, 2017b). However, our results indicate that this only happened for programs with global and interprocedural dependencies. Again, we hypothesize that this happens because the long distance between the dependent variable definition and its usages makes it difficult to simulate different configurations of enabled/disabled features.

Intraprocedural dependencies had no influence on program comprehensibility.

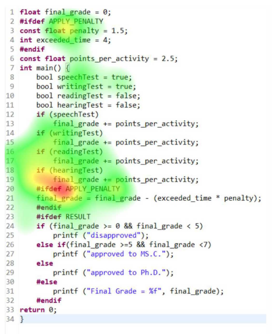
Our Results 2 and 6 indicate that #ifdefs may not affect the comprehensibility of programs with intraprocedural dependency. We hypothesize that this happens because the point of definition of dependent variable is closer from its usage. This result contradicts a previous study (MELO et al., 2017). Their study showed that #ifdef increases debugging time. However, it is important to say that, in their study, they did consider programs including different types of feature dependencies.

Interprocedural dependencies required more visual effort.

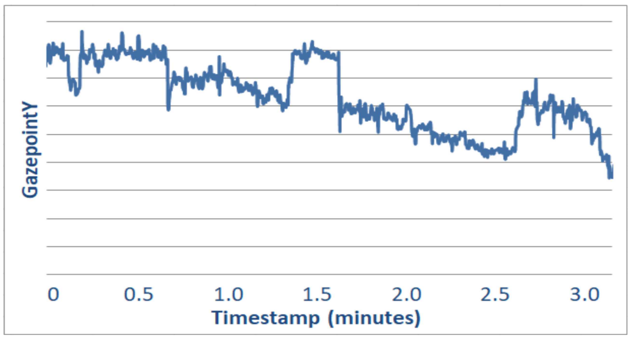
Results 8 and 9 show that the interprocedural dependency may increase visual effort. To find a bug, participants needed to perform more gaze transitions and focused more time on distinct parts of the source code.

Feature dependency did not affect the number of found bugs.

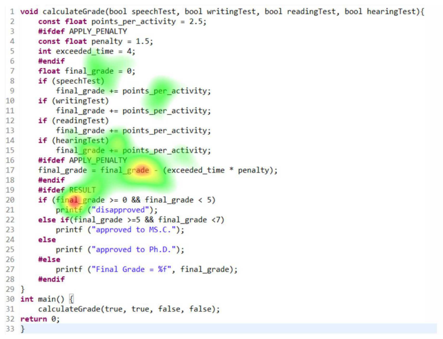
Result 4 revealed that the number of correctly found bugs was not affected by features dependency. This means that feature dependency may increase time and visual effort to find bugs, but do not decrease developers' ability to find bugs. This result confirms previous studies that showed that most participants correctly identify bugs in programs



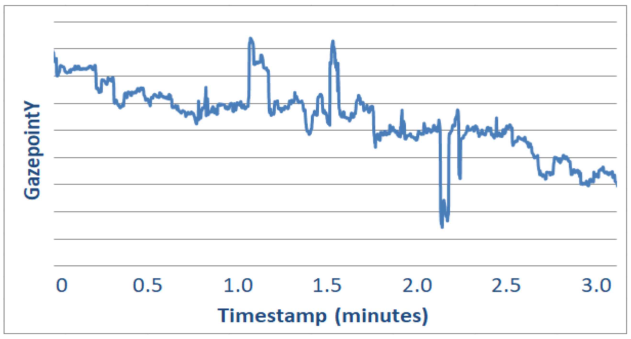
```c
float final_grade = 0;
#ifdef APPLY_PENALTY
const float penalty = 1.5;
int exceeded_time = 4;
#endif
const float points_per_activity = 2.5;
int main() {
    bool speechTest = true;
    bool writingTest = true;
    bool readingTest = false;
    bool hearingTest = false;
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    #ifdef APPLY_PENALTY
    final_grade = final_grade - (exceeded_time * penalty);
    #endif
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
return 0;
}
```



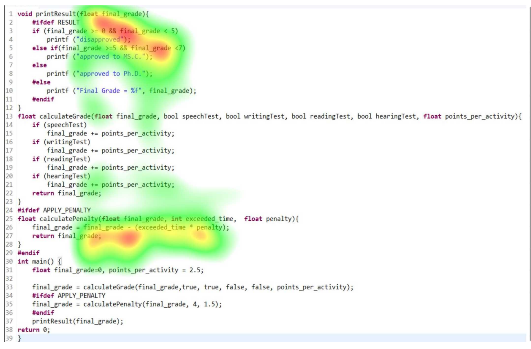
(a) Global



```c
void calculateGrade(bool speechTest, bool writingTest, bool readingTest, bool hearingTest){
    const float points_per_activity = 2.5;
    #ifdef APPLY_PENALTY
    const float penalty = 1.5;
    int exceeded_time = 4;
    #endif
    float final_grade = 0;
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    #ifdef APPLY_PENALTY
    final_grade = final_grade - (exceeded_time * penalty);
    #endif
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
}
int main() {
    calculateGrade(true, true, false, false);
return 0;
}
```

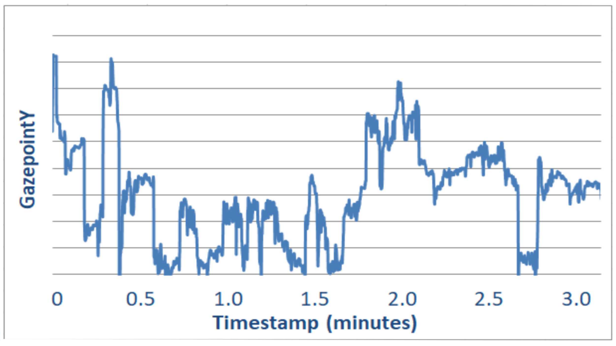


(b) Intraprocedural



```c
void printResult(float final_grade){
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
}
float calculateGrade(float final_grade, bool speechTest, bool writingTest, bool readingTest, bool hearingTest, float points_per_activity){
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    return final_grade;
}
#ifdef APPLY_PENALTY
float calculatePenalty(float final_grade, int exceeded_time,  float penalty){
    final_grade = final_grade - (exceeded_time * penalty);
    return final_grade;
}
#endif
int main() {
    float final_grade=0, points_per_activity = 2.5;

    final_grade = calculateGrade(final_grade,true, true, false, false, points_per_activity);
    #ifdef APPLY_PENALTY
    final_grade = calculatePenalty(final_grade, 4, 1.5);
    #endif
    printResult(final_grade);
return 0;
}
```



(c) Interprocedural

Figure 4.2 Heat map and gaze transition diagram with initial scan

with #ifdef (MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017). However, programs with interprocedural dependency had fewer hits than other types of feature dependency.

Other findings

We also compared data in terms of variability bugs. We observed that the "null pointer dereference" bug was always difficult to find. Participants spent more time and found a lower number "null pointer dereference" bugs in both with and without #ifdef versions of the programs. Only 40% of the participants found this bug. In contrast, the "uninitialized variable" bug was always easy to find. Only 3 developers of 30 did not find this bug. These results may indicate that these bugs are not induced by variability, which contradicts previous studies (ABAL; BRABRAND; WASOWSKI, 2014; ABAL et al., 2018). It is important to recall that these differences did not impact the analysis about feature dependencies as all participants analyzed programs with all types of bugs.

4.6 THREATS TO VALIDITY

4.6.1 Internal validity

Programming language. We wrote our programs in C, because conditional compilation directives in C are native and are one of the most popular mechanisms in use. Beside, most of the studies that report variability bugs are also in C. The knowledge in C could influence our results. To minimize that, we only admitted participants with previous experience on C.

Participants' experience. We selected participants according their level of experience and distributed them into the Latin square groups. For example, each of the six PhD students went to a different group (G1 - G6) (Table 4.2). The order of participation in the experiment determined the group. For example, the first PhD student went to group one (G1) and so on. So, we controlled confounding factors via the Latin square design and randomization. No participant was an expert in #ifdef, thus the distribution of participants in the Latin square was balanced.

Lab settings. All experiment trials were done in similar classrooms and with our supervision. We observed temperature and brightness conditions. In the moment of the execution of experiment, the classroom had only the participant and the authors.

4.6.2 External validity

Real bugs and features. Due to limitations we used small programs. But, our programs were inspired on concrete variability bugs found in real configurable systems. For this reason, our results may hold to other programs. However, for programs over 39 lines of code and more than two features, there may be additional effects that we have not observed.

Meaningful variables names. The maintenance and comprehensibility of source code can be hindered by choosing variables with non-meaningful names. Choosing meaningful variable names is also important for domain comprehension. To minimize this threat, we select relevant and meaningful variable names.

Mental simulation of scenarios. In practice, programmers encounter codes containing many features, and they don't always test or compile all possible configurations. Our results are limited to programs with a few features where the developer mentally simulates all possible configurations. Additionally, we utilize functional features, which are easier to analyze than architectural features, for example.

Lab settings. Our results are also limited to the environment we adopted. A more realistic environment, with IDEs and source code with multiple files, would be ideal. However, this design would not be attractive for many participants, since it would require more time for execution. In addition, we have the limitation that the source code should fit on the screen due to the eye tracking device.

4.6.3 Construct validity

Comprehensibility measurement. Measuring comprehensibility is not trivial because it involves human factors. Therefore, it is always a threat to construct validity. To minimize this threat, we quantified comprehensibility by means of different metrics, all of them already used in previous studies.

Chapter
5

STUDY 3 - AN EXPERIMENT ON HOW DEPENDENT VARIABLES AFFECT PROGRAM COMPREHENSIBILITY

This chapter describes an experiment that aims at evaluating how dependent variables impact the comprehensibility of configurable systems. Dependent variable is the name given to every variable that is defined within a feature and used within another feature. A dependent variable is what imposes a dependency relationship between features. In this sense, the variation in the amount of variable definition and variable usage can affect the comprehensibility of configurable systems.

This study aims to answer our third research question:
-- *How do different numbers of dependent variables affect the comprehensibility of configurable system source code?*

5.1 DESIGN

To answer our research question, we carried out a controlled experiment with 12 developers who analyzed programs trying to specify the output. Developers analyzed four similar programs in terms of lines of code, cyclomatic complexity and number of feature dependency. The differences between the programs were the domain, and the number of definitions of dependent variables.

We compared the comprehension effort the developers spent to analyze each program. We quantified comprehension effort from different perspectives: (i) time to analyze each program, (ii) number of attempts until the developers provide the correct answer, (iii) visual effort and, (iv) heart-related biometrics. We quantified visual effort by means of different metrics collected by the use of an eye-tracking device and we quantified heart-related biometrics collected by the use of a smartwatch. We give more details about these metrics in Section 5.8

In order to avoid learning effect, we selected two different domains. Domain 1 (sale of property) and Domain 2 (grade calculation) each one with 3 features and 7 feature dependencies. We implemented two programs for each domain, one program with 2

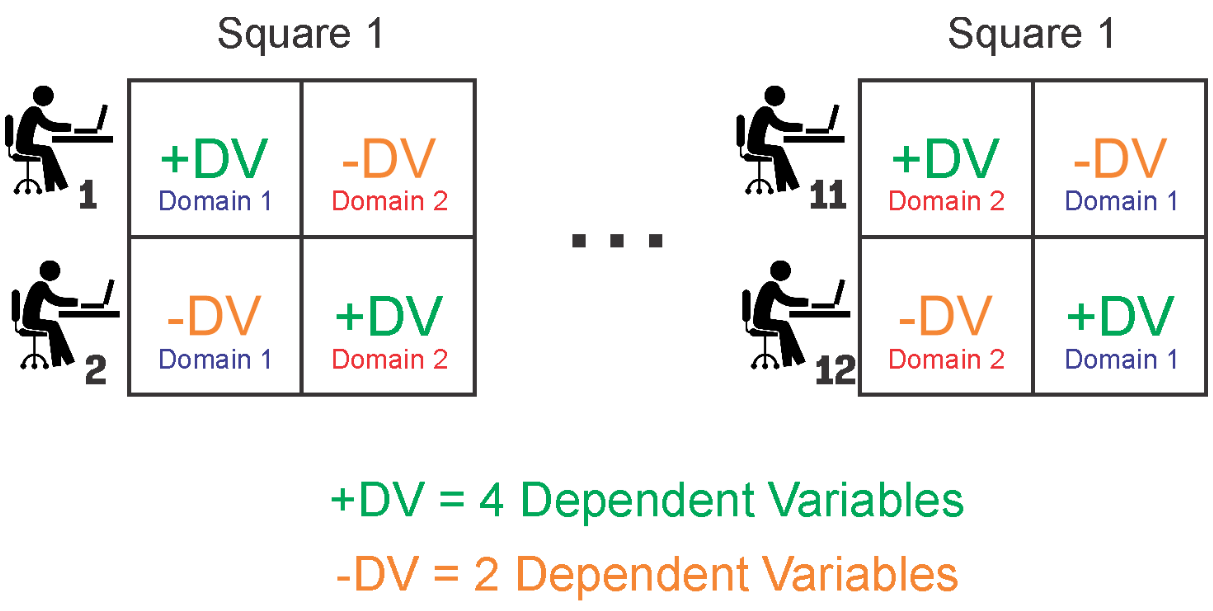


Figure 5.1 Latin Square design (2x2).

dependent variables and the other with 4 dependent variables. We used programs with a maximum of 4 dependent variables due to display limitations of the code on the screen. In summary, for each domain, we implemented two programs with different numbers of dependent variables.

We designed our experiment as a standard Latin Square. Our previous studies discuss Latin Square design (Chapter 3 and Chapter 4). Figure 5.1 explains our 2x2 Latin Square. In its columns, we have the treatment, in this case, the number of dependent variables. The lines represent developers. The acronyms in the cells represent the program characteristics, +DV represents a program with 4 dependent variables, and -DV represents a program with 2 dependent variables. Developer 1 firstly analyzed the program with 4 dependent variables, and, afterwards, the program with 2 dependent variables. Developer 2 also analyzed programs with 4 and 2 dependent variables but in reverse order.

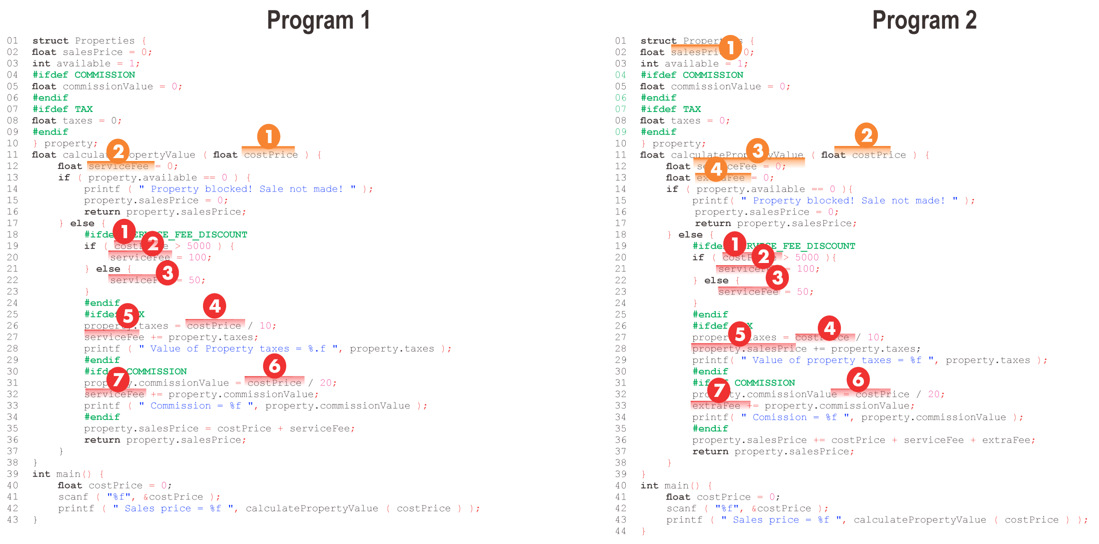
To avoid learning effects due repetition of domains we distribute the domains along our Latin square columns. Each column has a different domain. Therefore, each developer analyzed two different programs, each one with different numbers of dependent variables and different domains.

5.2 PARTICIPANTS

We counted on 12 participants to run our experiment: six professors, and six developers from industry. We selected professors from one university in Brazil and developers from three companies in Brazil. No compensation was provided for the participants.

No data were discarded due to poor quality. Eight participants have normal vision and four have vision corrected by glasses. Four participants are females. All of them have experience in C programming language. Nine participants declared themselves as expert developers and all of them knew the syntax of #ifdef.

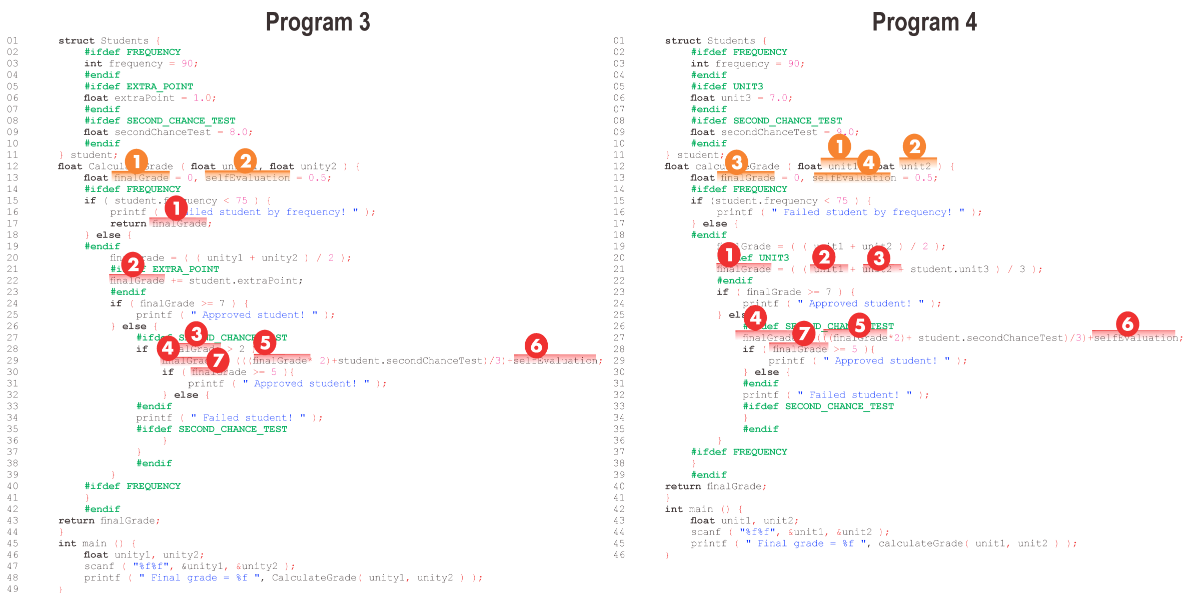
5.3 PROGRAMS



```
struct Properties {
float salesPrice = 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    if ( property.available == 0 ) {
        printf ( " Property blocked! Sale not made! " );
        property.salesPrice = 0;
        return property.salesPrice;
    } else {
        #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ) {
            serviceFee = 100;
        } else {
            serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        serviceFee += property.taxes;
        printf ( " Value of Property taxes = %.f ", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        serviceFee += property.commissionValue;
        printf ( " Commission = %f ", property.commissionValue );
        #endif
        property.salesPrice = costPrice + serviceFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

```
struct Properties {
float salesPrice = 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    float extraFee = 0;
    if ( property.available == 0 ){
        printf( " Property blocked! Sale not made! " );
         property.salesPrice = 0;
         return property.salesPrice;
    } else {
        #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ){
            serviceFee = 100;
        } else {
            serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        property.salesPrice += property.taxes;
        printf( " Value of property taxes = %f ", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        extraFee += property.commissionValue;
        printf( " Comission = %f ", property.commissionValue );
        #endif
        property.salesPrice += costPrice + serviceFee + extraFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

Figure 5.2 Program 1 and Program 2 highlighting variable definitions (orange) and variable usages (red).



```
    struct Students {
        #ifdef FREQUENCY
        int frequency = 90;
        #endif
        #ifdef EXTRA_POINT
        float extraPoint = 1.0;
        #endif
        #ifdef SECOND_CHANCE_TEST
        float secondChanceTest = 8.0;
        #endif
    } student;
    float CalculateGrade ( float unity1, float unity2 ) {
        float finalGrade = 0, selfEvaluation = 0.5;
        #ifdef FREQUENCY
        if ( student.frequency < 75 ) {
            printf ( " Failed student by frequency! " );
            return finalGrade;
        } else {
        #endif
            finalGrade = ( ( unity1 + unity2 ) / 2 );
            #ifdef EXTRA_POINT
            finalGrade += student.extraPoint;
            #endif
            if ( finalGrade >= 7 ) {
                printf ( " Approved student! " );
            } else {
                #ifdef SECOND_CHANCE_TEST
                if ( finalGrade > 2 ) {
                    finalGrade = (((finalGrade* 2)+student.secondChanceTest)/3)+selfEvaluation;
                    if ( finalGrade >= 5 ){
                        printf ( " Approved student! " );
                    } else {
                #endif
                printf ( " Failed student! " );
                #ifdef SECOND_CHANCE_TEST
                    }
                }
                #endif
            }
        #ifdef FREQUENCY
        }
        #endif
    return finalGrade;
    }
    int main () {
        float unity1, unity2;
        scanf ( "%f%f", &unity1, &unity2 );
        printf ( " Final grade = %f ", CalculateGrade( unity1, unity2 ) );
    }
```

```
    struct Students {
        #ifdef FREQUENCY
        int frequency = 90;
        #endif
        #ifdef UNIT3
        float unit3 = 7.0;
        #endif
        #ifdef SECOND_CHANCE_TEST
        float secondChanceTest = 9.0;
        #endif
    } student;
    float calculateGrade ( float unit1, float unit2 ) {
        float finalGrade = 0, selfEvaluation = 0.5;
        #ifdef FREQUENCY
        if (student.frequency < 75 ) {
            printf ( " Failed student by frequency! " );
        } else {
        #endif
            finalGrade = ( ( unit1 + unit2 ) / 2 );
            #ifdef UNIT3
            finalGrade = ( ( unit1 + unit2 + student.unit3 ) / 3 );
            #endif
            if ( finalGrade >= 7 ) {
                printf ( " Approved student! " );
            } else {
                #ifdef SECOND_CHANCE_TEST
                finalGrade = (((finalGrade*2)+ student.secondChanceTest)/3)+selfEvaluation;
                if ( finalGrade >= 5 ){
                    printf ( " Approved student! " );
                } else {
                #endif
                printf ( " Failed student! " );
                #ifdef SECOND_CHANCE_TEST
                }
                #endif
            }
        #ifdef FREQUENCY
        }
        #endif
    return finalGrade;
    }
    int main () {
        float unit1, unit2;
        scanf ( "%f%f", &unit1, &unit2 );
        printf ( " Final grade = %f ", calculateGrade( unit1, unit2 ) );
    }
```

Figure 5.3 Program 3 and Program 4 highlighting variable definitions (orange) and variable usages (red).

We implemented the programs used in our experiment inspired by our previous studies (Chapter 3 and Chapter 4) and common programming tasks. We avoided pieces of code form real programs (like the ones from Linux) because their complexity could affect comprehensibility. Furthermore, to facilitate understanding and to widen the audience of potential participants, our programs were written in the participants' native language (Portuguese).

Listing 5.1 Program 1: Sale of property domain with 2 dependent variable

```
struct Properties {
    float salesPrice = 0;
    int available = 1;
    #ifdef COMMISSION
    float commissionValue = 0;
    #endif
    #ifdef TAX
    float taxes = 0;
    #endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    if ( property.available == 0 ) {
        printf ( "Property blocked! Sale not made!" );
        property.salesPrice = 0;
        return property.salesPrice;
    } else {
    #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ) {
                serviceFee = 100;
        } else {
                serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        serviceFee += property.taxes;
        printf ( "Value of Property taxes = %.f", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        serviceFee += property.commissionValue;
        printf ( "Commission = %f", property.commissionValue );
        #endif
        property.salesPrice = costPrice + serviceFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( "Sales Price = %f", calculatePropertyValue ( costPrice )
        );
43 }
```

We used an eye-tracking device on a 32-inch screen to record all gaze movements of participants. Our programs should fit on a 45-line display window so that the eye-tracking device could record all gaze movements of participants. The participants were seated about 60 to 65 cm from the stimuli. A Courier New font of size 14 pt with text color black and background color white was used for presenting the source code. To make the code easier to understand, we used the default line spacing size and we highlighted ifdefs in green, symbols and numbers in red, and output texts in blue color. This color scheme is the default in most Integrated Development Environments (IDEs), such as Eclipse[1], CodeBlocks[2], DevC++[3], and Microsoft Visual Studio[4]. The variables had meaningful names and were defined using the CamelCase naming convention (MCCONNELL, 2004).

The four programs are similar in terms of number of lines of code (LOC) (LANZA; MARINESCU, 2007), number of features (NOFC) (LIEBIG et al., 2010) and McCabe cyclomatic complexity (CC) (MCCABE, 1976). In the following, we describe each program.

Program 1: sale of property with 2 dependent variables. Listing 5.1 shows Program 1 source code. It has three features: `COMMISSION`, `TAX` and `SERVICE_FEE_DISCOUNT` feature. `COMMISSION` calculates the value of commission of a property sale. `TAX` calculates the value of Government taxes and the feature `SERVICE_FEE_DISCOUNT` calculates the service fee.

This program has 2 dependent variables: `costPrice` and `serviceFee`. The variable `costPrice` defined in a mandatory feature in line 11 and `serviceFee` is also defined in a mandatory feature but in line 12. Variable `costPrice` had 3 uses that generate dependencies: (i) in line 19 it is used within `SERVICE_FEE_DISCOUNT` feature, (ii) in line 26 it used within `TAX` feature and (iii) in line 31 it is used within `COMMISSION` feature.

The variable `serviceFee` had 4 uses that generate dependencies: (i) in line 20 within `SERVICE_FEE_DISCOUNT` feature, (ii) in line 22 also within `SERVICE_FEE_DISCOUNT` feature, (iii) in line 32 within `COMMISSION` feature and (iv) in line 32 within of `COMMISSION` feature.

Program 2: sale of property with 4 dependent variables. Listing 5.2 shows Program 2. We rewrote the program shown in Listing 5.1 by including 2 dependent variables: `salesPrice`, defined in line 2 and `extraFee`, defined in line 13.

The variable `salesPrice` is defined within a mandatory feature. It causes a dependency because it is used within `TAX` feature in line 28. The variable `extraFee` is also defined within a mandatory feature. It causes a dependency because it is used within `COMMISSION` feature in line 33.

Figure 5.2 shows Program 1 and Program 2 side by side highlighting dependent variables: variable definition points in orange, and variable usage points in red.

Program 3: grade calculation with 2 dependent variables.. Listing 5.3 shows

[1]https://www.eclipse.org

[2]https://www.codeblocks.org/

[3]https://sourceforge.net/projects/orwelldevcpp

[4]https://visualstudio.microsoft.com

Program 3. It has three features: `FREQUENCY`, `EXTRA_POINT` and `SECOND_CHANCE_TEST` feature. `FREQUENCY` verifies the percentage of student attendance. `EXTRA_POINT` adds extra points to students due to some extra class tasks and `SECOND_CHANCE_TEST` gives the student the chance for an extra test.

Listing 5.2 Program 2: Sale of property domain with 4 dependent variables

```c
struct Properties {
    float salesPrice = 0;
    int available = 1;
    #ifdef COMMISSION
    float commissionValue = 0;
    #endif
    #ifdef TAX
    float taxes = 0;
    #endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    float extraFee = 0;
    if ( property.available == 0 ){
        printf( "Property blocked! Sale not made!" );
        property.salesPrice = 0;
        return property.salesPrice;
    } else {
        #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ){
            serviceFee = 100;
        } else {
            serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        property.salesPrice += property.taxes;
        printf( "Value of property taxes = %f", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        extraFee += property.commissionValue;
        printf( "Comission = %f", property.commissionValue );
        #endif
        property.salesPrice += costPrice + serviceFee + extraFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( "Sales price = %f",calculatePropertyValue (costPrice ));
}
```

Listing 5.3 Program 3: Grade calculation domain with 2 dependent variables

```c
struct Students {
    #ifdef FREQUENCY
    int frequency = 90;
    #endif
    #ifdef EXTRA_POINT
    float extraPoint = 1.0;
    #endif
    #ifdef SECOND_CHANCE_TEST
    float secondChanceTest = 8.0;
    #endif
} student;
float CalculateGrade ( float unity1, float unity2 ) {
    float finalGrade = 0, selfEvaluation = 0.5;
    #ifdef FREQUENCY
    if ( student.frequency < 75 ) {
        printf ( "Failed student by frequency!" );
        return finalGrade;
    } else {
        #endif
        finalGrade = ( ( unity1 + unity2 ) / 2 );
        #ifdef EXTRA_POINT
        finalGrade += student.extraPoint;
        #endif
        if ( finalGrade >= 7 ) {
           printf ( "Approved student!" );
        } else {
            #ifdef SECOND_CHANCE_TEST
            if ( finalGrade > 2 ){
                finalGrade = ( ( ( finalGrade * 2 ) + student.
                   secondChanceTest ) / 3 ) + selfEvaluation;
                if ( finalGrade >= 5 ){
                printf ( "Approved student!" );
                } else {
            #endif
                printf ( "Failed student!" );
                #ifdef SECOND_CHANCE_TEST
                }
            }
            #endif
        }
    #ifdef FREQUENCY
    }
    #endif
    return finalGrade;
}
int main () {
    float unity1, unity2;
    scanf ( "%f%f", &unity1, &unity2 );
    printf ( "Final grade = %f", CalculateGrade(unity1, unity2));
}
```

Listing 5.4 Program 4: Grade calculation domain with 4 dependent variables

```
1  struct Students {
2      #ifdef FREQUENCY
3      int frequency = 90;
4      #endif
5      #ifdef UNIT3
6      float unit3 = 7.0;
7      #endif
8      #ifdef SECOND_CHANCE_TEST
9      float secondChanceTest = 9.0;
10     #endif
11 } student;
12 float calculateGrade ( float unit1, float unit2 ) {
13     float finalGrade = 0, selfEvaluation = 0.5;
14     #ifdef FREQUENCY
15     if (student.frequency < 75 ) {
16         printf ( "Failed student by frequency!" );
17     } else {
18         #endif
19         finalGrade = ( ( unit1 + unit2 ) / 2 );
20         #ifdef UNIT3
21         finalGrade = ( ( unit1 + unit2 + student.unit3 ) / 3 );
22         #endif
23         if ( finalGrade >= 7 ) {
24             printf ( "Approved student!" );
25         } else {
26             #ifdef SECOND_CHANCE_TEST
27             finalGrade = ( ( ( finalGrade * 2 ) + student.
                   secondChanceTest ) / 3 ) + selfEvaluation;
28             if ( finalGrade >= 5 ){
29                 printf ( "Approved student!" );
30             } else {
31             #endif
32                 printf ( "Failed student!" );
33             #ifdef SECOND_CHANCE_TEST
34             }
35             #endif
36         }
37     #ifdef FREQUENCY
38     }
39     #endif
40     return finalGrade;
41 }
42 int main () {
43     float unit1, unit2;
44     scanf ( "%f%f", &unit1, &unit2 );
45     printf ( "Final grade = %f", calculateGrade( unit1, unit2 ));
46 }
```

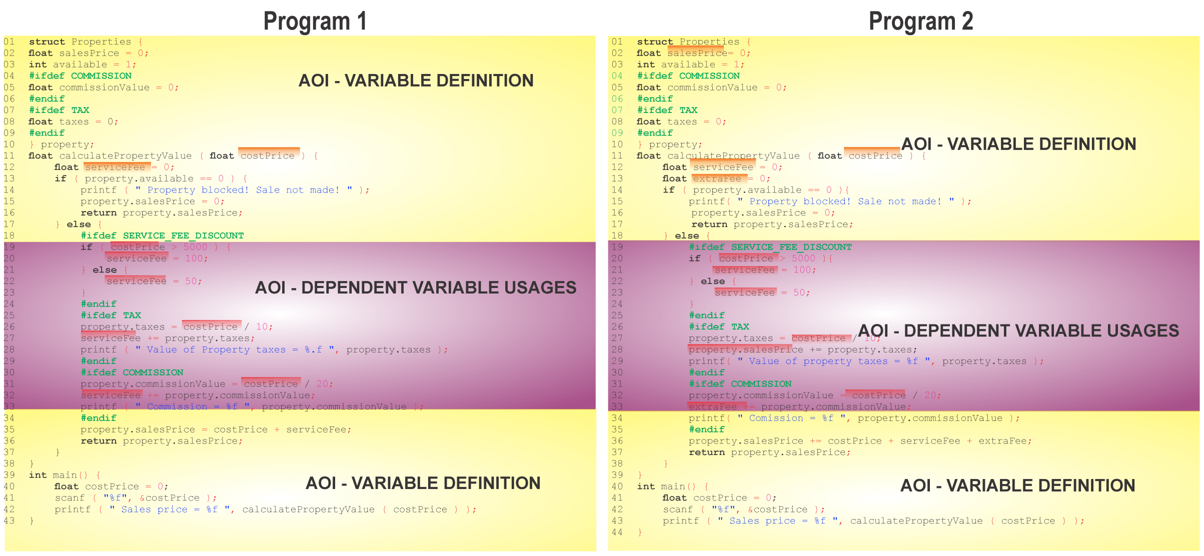
This program has 2 dependent variables: `finalGrade`, defined in a mandatory feature in line 13, and `selfEvaluation`, also defined in a mandatory feature in line 13. Variable `finalGrade` has 5 uses that generate dependencies: (i) in line 17 it is used within `FREQUENCY` feature, (ii) in line 22 it is used within `EXTRA_POINT` feature, the other uses occur within `SECOND_CHANCE_TEST` feature, (iii) in line 28, (iv) in line 29, and (v) in line 30. The variable `selfEvaluation` has 1 use that generates dependencies in line 29 within `SECOND_CHANCE_TEST` feature.

Program 4: grade calculation with 4 dependent variables. Listing 5.4 shows Program 4. We rewrote the program showed in Listing 5.3. We replaced the feature `EXTRA_POINT` with `UNIT3` and we included 2 dependencies in 2 variables: `unity1` and `unity2`. variables `unity1` and `unity1` are defined in line 12 within a mandatory feature. They are used in line 21 within `UNIT3` feature.

Figure 5.3 shows Program 3 and Program 4 side by side highlighting dependent variables. Variable definition points are in orange and variable usage points are in red.

5.4 AREAS OF INTEREST (AOI)

An AOI is a region of interest of a displayed stimulus in an eye-tracker-based study. In those type of studies, we can extract eye movement metrics based on AOIs.(KRAJBICH; ARMEL; RANGEL, 2010). We defined our AOIs with OGAMA AOI editor (VOSSKÜHLER et al., 2008). Our AOIs comprised regions of source code that contained variable definitions and variable usages.

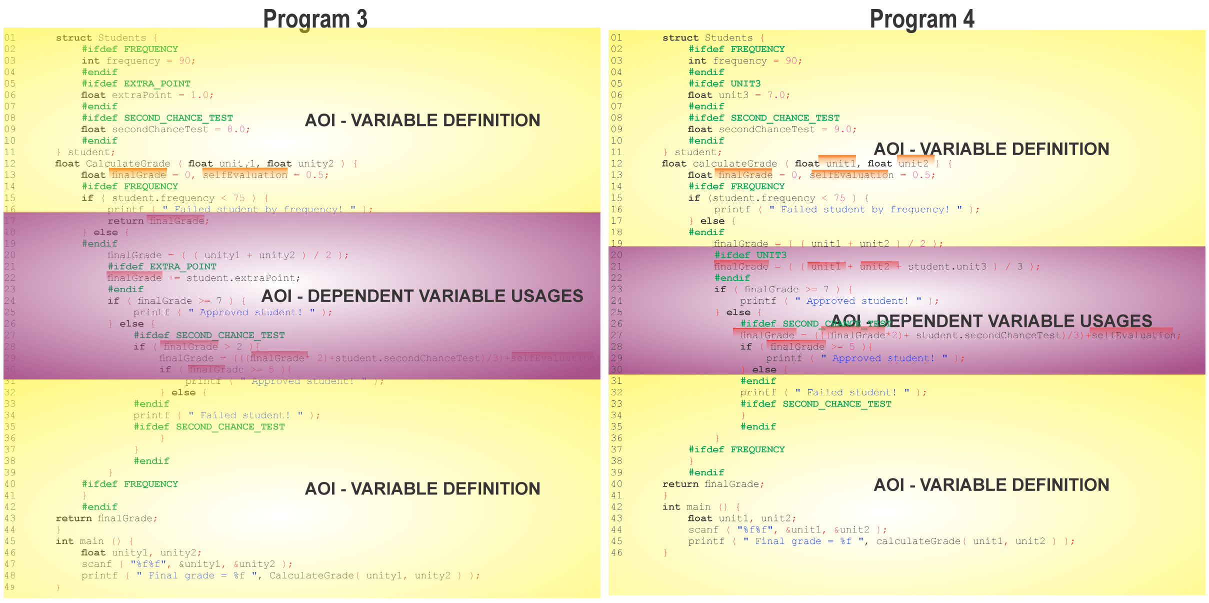

Program 1
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION
Program 2
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION

```
struct Properties {
float salesPrice = 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    if ( property.available == 0 ) {
        printf ( " Property blocked! Sale not made! " );
        property.salesPrice = 0;
        return property.salesPrice;
    } else {
        #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ) {
            serviceFee = 100;
        } else {
            serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        serviceFee += property.taxes;
        printf ( " Value of Property taxes = %.f ", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        serviceFee += property.commissionValue;
        printf ( " Commission = %f ", property.commissionValue );
        #endif
        property.salesPrice = costPrice + serviceFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

```
struct Properties {
float salesPrice= 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
    float serviceFee = 0;
    float extraFee = 0;
    if ( property.available == 0 ){
        printf( " Property blocked! Sale not made! " );
         property.salesPrice = 0;
         return property.salesPrice;
    } else {
        #ifdef SERVICE_FEE_DISCOUNT
        if ( costPrice > 5000 ){
            serviceFee = 100;
        } else {
            serviceFee = 50;
        }
        #endif
        #ifdef TAX
        property.taxes = costPrice / 10;
        property.salesPrice += property.taxes;
        printf( " Value of property taxes = %f ", property.taxes );
        #endif
        #ifdef COMMISSION
        property.commissionValue = costPrice / 20;
        extraFee += property.commissionValue;
        printf( " Comission = %f ", property.commissionValue );
        #endif
        property.salesPrice += costPrice + serviceFee + extraFee;
        return property.salesPrice;
    }
}
int main() {
    float costPrice = 0;
    scanf ( "%f", &costPrice );
    printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

Figure 5.4 Areas of Interest of Program 1 and Program 2.

Figure 5.4 and Figure 5.5 show our AOIs. We defined two main AOIs: (i) the first AOI is the region where dependent variables are defined, and (ii) the second AOI is the region where dependent variables are used. We defined those regions because we want to analyze the visual effort of developers analyzing source codes with different numbers of dependent variables. In this case, the regions of variable definitions and variable usages


Program 3
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION
Program 4
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION

```
struct Students {
    #ifdef FREQUENCY
    int frequency = 90;
    #endif
    #ifdef EXTRA_POINT
    float extraPoint = 1.0;
    #endif
    #ifdef SECOND_CHANCE_TEST
    float secondChanceTest = 8.0;
    #endif
} student;
float CalculateGrade ( float unity1, float unity2 ) {
    float finalGrade = 0, selfEvaluation = 0.5;
    #ifdef FREQUENCY
    if ( student.frequency < 75 ) {
        printf ( " Failed student by frequency! " );
        return finalGrade;
    } else {
    #endif
        finalGrade = ( ( unity1 + unity2 ) / 2 );
        #ifdef EXTRA_POINT
        finalGrade += student.extraPoint;
        #endif
        if ( finalGrade >= 7 ) {
            printf ( " Approved student! " );
        } else {
            #ifdef SECOND_CHANCE_TEST
            if ( finalGrade > 2 ){
                finalGrade = (((finalGrade* 2)+student.secondChanceTest)/3)+selfEvaluation;
                if ( finalGrade >= 5 ){
                    printf ( " Approved student! " );
                } else {
            #endif
            printf ( " Failed student! " );
            #ifdef SECOND_CHANCE_TEST
                }
            }
            #endif
        }
    #ifdef FREQUENCY
    }
    #endif
return finalGrade;
}
int main () {
    float unity1, unity2;
    scanf ( "%f%f", &unity1, &unity2 );
    printf ( " Final grade = %f ", CalculateGrade( unity1, unity2 ) );
}
```

```
struct Students {
    #ifdef FREQUENCY
    int frequency = 90;
    #endif
    #ifdef UNIT3
    float unit3 = 7.0;
    #endif
    #ifdef SECOND_CHANCE_TEST
    float secondChanceTest = 9.0;
    #endif
} student;
float calculateGrade ( float unit1, float unit2 ) {
    float finalGrade = 0, selfEvaluation = 0.5;
    #ifdef FREQUENCY
    if (student.frequency < 75 ) {
        printf ( " Failed student by frequency! " );
    } else {
    #endif
        finalGrade = ( ( unit1 + unit2 ) / 2 );
        #ifdef UNIT3
        finalGrade = ( ( unit1 + unit2 + student.unit3 ) / 3 );
        #endif
        if ( finalGrade >= 7 ) {
            printf ( " Approved student! " );
        } else {
            #ifdef SECOND_CHANCE_TEST
            finalGrade = (((finalGrade*2)+ student.secondChanceTest)/3)+selfEvaluation;
            if ( finalGrade >= 5 ){
                printf ( " Approved student! " );
            } else {
            #endif
            printf ( " Failed student! " );
            #ifdef SECOND_CHANCE_TEST
            }
            #endif
        }
    #ifdef FREQUENCY
    }
    #endif
return finalGrade;
}
int main () {
    float unit1, unit2;
    scanf ( "%f%f", &unit1, &unit2 );
    printf ( " Final grade = %f ", calculateGrade( unit1, unit2 ) );
}
```

Figure 5.5 Areas of Interest of Program 3 and Program 4.

can reveal the behavior of developers when analyzing source code with different numbers of dependent variables.

5.5 PILOT STUDIES

Before executing the actual experiment, we carried out two pilot studies. In the first one, the participants had to find variability bugs in the programs. The participants of this first pilot study also participated in our Study 2 (Chapter 4). Because of this they already known about the types of variability bugs they had to find, which were of the same types of some of the bugs we used in Study 2. Therefore, we noticed they browsed the program focusing on find those types of bugs insteady of trying to understand the program. As we did not have many alternatives, other potential participants of this study could also have participated in Study 2. Thus, using variability bug searching as task of this study would threat its internal validity. As a consequence, we decided to change the task.

We, then, performed a second pilot study with a different kind of tasks for the participants. They had to correctly specified the program output given a set of input data and a specific configuration of enabled and disabled features. In this case, the pilot went well. We performed the pilot studies with two master's degrees and one doctoral degree.

In the two pilot study runs, participants analyzed codes with 2 and 4 dependent variables, and the results of time and visual effort already indicated significant differences between the codes. These initial results allowed us to conclude that it was possible to compare codes with few and many dependent variables using this design. We did not consider their results in our analysis.

5.6 EXPERIMENT PROCEDURES

We performed each trial of the experiment with each participant individually. Before starting the experiment tasks, we cleaned all materials and equipment with alcohol gel, as the COVID-19 pandemic was still ongoing in many parts of the world, there were variations in the intensity and measures taken by different countries. Finally, we asked the participant to fill out a consent form. All participants signed the consent form.

Then, the participant put on the smartwatch. The participants were instructed not to make sudden movements to prevent inaccurate measurements by the smartwatch. Before the participant start analyzing the first program, we asked her or him to watch a two minute full-screen video of a fish swimming in an aquarium. We did the same before the participant start analyzing the second program. The video was intended to help participants relax and allow us to record a baseline of her or his heart rate. In previous studies, we saw that a persons biometric features drop back to a baseline after about a minute of watching the video (FRITZ et al., 2014; MÜLLER; FRITZ, 2015). After the video, we calibrated the eye-tracking device and the smartwatch and we synced the clock of the smartwatch with the time of the eye-tracking computer.

We asked the participants if they had any cardiac problems. Only one participant claimed to have cardiac arrhythmia. However, during the experiment, individual results did not show any type of alteration. Another participant claimed to be experiencing family-related stress. However, the detected change was that the baseline heart rate was elevated, which did not compromise the analysis of the results.

The participant analyzed the programs as we planned in our Latin square design 5.1. We observed the participant and monitored the initial and final time that her or him spent for completing each task. The participant had three tasks per program. We give more details about the tasks in Section 5.7. We used the tool that record the gaze data to record the time. For each new task, the tool reseted the timer. We check if the participant correctly specify the output and then finished the record. The participant was not allowed to proceed to the next task until she or he correctly answered the task in progress. We counted and registered the number of attempts the participant needed to correctly answered each task.

We presented each program to the participants as static images displayed on a screen. Participants did not have access to tools, IDEs or browsers. For each participant, we recorded `x` and `y` coordinates (fixations) via an eye tracker, and heart-related biometrics via a smartwatch.

We performed each experiment trial individually in the same lab using the same monitor to avoid unintended effects from different software and hardware environments. The screen resolution was set to 1920 by 1080 pixels into a 32 inch LCD screen. We recorded all of the eye tracking data using the open-source tool OGAMA (VOSSKÜHLER et al., 2008). We used the Tobii Eye tracker 5[5] Device and the Garmin Fenix 5s[6] smartwatch.

[5]https://gaming.tobii.com/product/eye-tracker-5/
[6]https://www.garmin.com/en-US/p/552237/

5.7 TASKS

Each participant received a task instruction form that explains the experiment. Each participant had to understand and realize the mental execution of two programs, one program with more dependent variables and the other with fewer dependent variables. The order of programs depends on Latin Square. Each participant also had to answer three tasks about each program. We motivated the participant not to direct their eyes off the screen while performing the task.

We explained to the participant the proposed scenarios and initial values of each task before she or he started. The participant could also find these same instructions in the instruction form. The three tasks force the participant to mentally simulate different configurations involving dependent variables. To ensure the same difficulty level in all sets of tasks, we defined the same three feature configuration scenarios for all programs: (i) Task 01: all features enabled, (ii) Task 02: one feature disabled and two features enabled, (iii) Task 03: all features disabled.

For example, the first task about the two programs on domain 1 (one with 2 dependent variables, the other with 4) should be answered considering all features enabled. The task was presented to the participants as follows:

TASK 1: Consider:
FEATURE ENABLED: `COMISSION`, `TAX` and `SERVICE_FEE_DISCOUNT`
FEATURES DISABLED: none.
INITIAL VALUES: `priceCost` = 2000 in line 41.
QUESTION: "What will be printed on line 42 when the `int main( )` function on line 39 is executed?"

The tasks about Program 1 (with 2 dependent variables) and of Program 2 (with 4 dependent variables) were the same. And, according our Latin Square, we allocated half of the participants to execute the tasks based on Program 1, and the other half to execute the tasks based on Program 2.

5.8 EXPERIMENTAL RESULTS

In this section, we test our hypotheses and discuss the results. We measured comprehensibility according to: (i) time to provide the correct answer to the tasks, (ii) number of attempts until the participants provide the correct answer to the tasks, (iii) visual effort with number of fixations, gaze transitions, and attention maps, and (iv) heart-related biometrics.

In this study, as in previous studies (Chapter 3 and Chapter 4), we used Analysis of Variance (ANOVA) tests for hypothesis testing. ANOVA is a statistical test used to analyze the difference between the means of more than two groups. We used p-value $<$ 0.05 as the probability for rejecting null hypotheses. The only exception, was the *number of attempts until the developers provide the correct answer* variable. ANOVA does not apply for it as it holds binary values. Thus, we used inferential statistics to evaluate it. We ran our tests with the support of R[7]. All artifacts used in our experiment are available

[7]http://www.r-project.org/

at our website[8] and our research share website.[9] In the following, we present the results regarding each metric.

5.8.1 Time to provide the correct answer

Table 5.1 Mean time to provide the correct answer (in seconds)

With 2 dependent variables			With 4 dependent variables		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
66	74	70	81	128	105

Similarly to our previous studies (Chapters 3 and 4), we measured the time (in seconds) each participant took to analyze each program. Our null hypothesis about this metric is:

H_0t: There is no significant difference in the time to provide the correct answer to the tasks when comparing programs with different numbers of dependent variables.

Rows in Table 5.1 show the mean time spent by participants for each program. We had 6 participants who answered 3 tasks for each program, which summed up a total of 18 answers per program. In total of four programs we had 72 observations. Shapiro test confirmed that the data about time to provide the correct answer was normally distributed.

Table 5.1 shows the mean time spent by all participants for programs with 2 and 4 dependent variables. They spent a mean time of 70 seconds analyzing programs with 2 dependent variables and 105 seconds for programs with 4 dependent variables. Our data revealed that there was a significant difference in time for the developers to analyze programs with different numbers of dependent variables (p-value = 0.04373). We, thus, reject our null hypothesis (H_0t).

Result 1: Programs with more dependent variables required more time for the participants to answer the tasks correctly than programs with fewer dependent variables.

5.8.2 Number of attempts needed until correct answer

Table 5.2 Total number of attempts needed until correct answer

With 2 dependent variables			With 4 dependent variables		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
18	19	37	18	20	38

Similarly to our Study 1 (Chapter 4), we measured the total number of attempts needed for the participants until they specify the correct output of the programs. The participant scored one for each attempt for each task correctly answered. This metric

[8]http://www.djansantos.com.br/projects/dependentVariables/
[9]https://doi.org/10.5281/zenodo.7982409

sum the total of attempts for all participants for each program. Our null hypothesis about this metric is:

H_0a: There is no significant difference in number of attempts needed until correct answer when comparing programs with different numbers of dependent variables.

Table 5.2 shows the total number of attempts needed to specify the output correctly by all participants. They needed 37 attempts for programs with 2 dependent variables and 38 attempts for programs with 4 dependent variables. The χ^2 test (Pearson's Chi-squared test) (CAMILLI; HOPKINS, 1978) revealed no significant difference between the number of attempts needed to specify the output correctly for programs with 2 and 4 dependent variables. The value χ^2 is 0.69, and the p-values is 0.4034. Based on this, we cannot reject our null hypothesis H_0a.

Result 2: There was no significant difference in the number of attempts needed for the participants until giving the correct answer when comparing programs with different numbers of dependent variables.

5.8.3 Visual effort

Total number of fixations

Table 5.3 Number of fixations

With 2 dependent variable			With 4 dependent variable		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
195	151	346	218	219	437

The number of fixations increases when a text is difficult to comprehend (RAYNER et al., 2006). We counted the number of fixations per program. Our null hypothesis about this metric is:

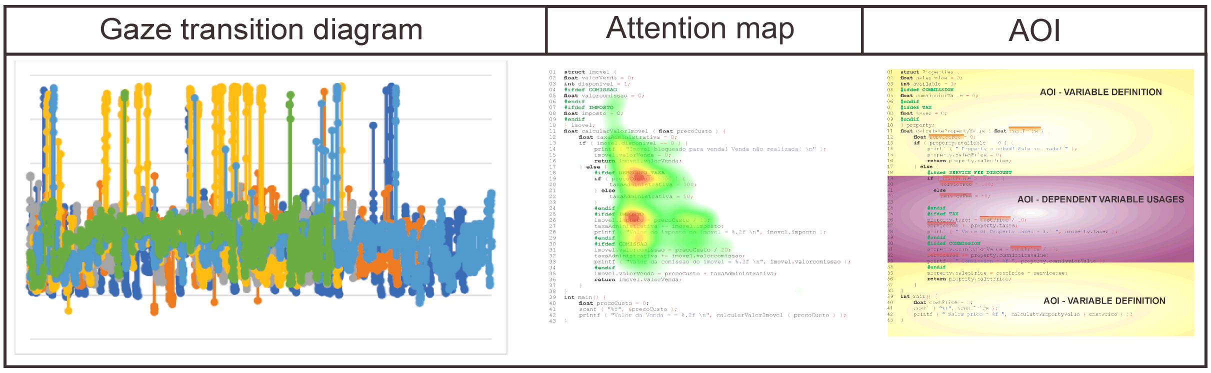
H_0f: There is no significant difference in the number of fixations to specify the output when comparing programs with different numbers of dependent variables.

Table 5.3 shows the total number of fixations the participants executed when analyzing the programs in order to specify their correct output. They executed 346 fixations in programs with 2 dependent variables and 437 fixations in programs with 4 dependent variables. Our data revealed that the number of fixations was significantly different between programs with different numbers of dependent variables (p-value = 0.04479). We reject our null hypothesis (H_0f)

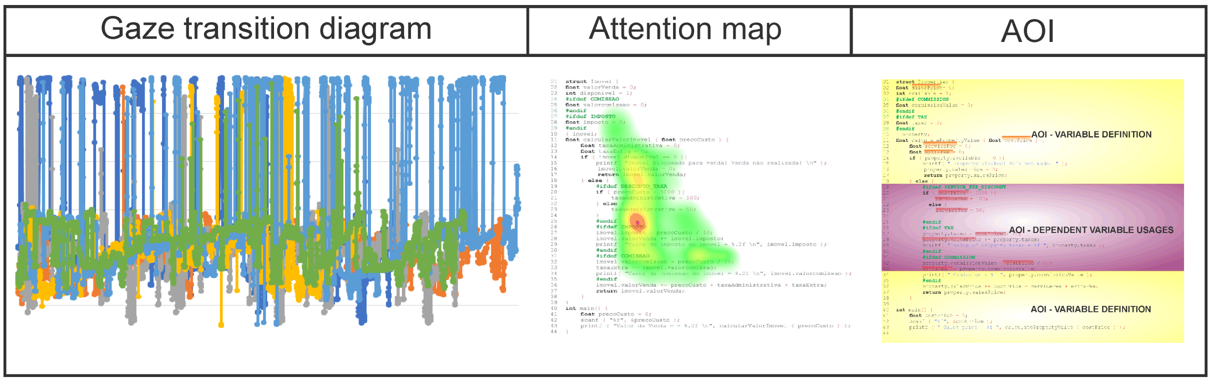
Result 3: Developers made more fixations to analyze programs with more dependent variable.

Gaze transitions and attention map

We already used and discussed data related to gaze transitions and attention maps in our Study 2 (Chapter 4). Gaze transitions (saccades) are rapid eye movements from one place to another separated by pauses (RAYNER et al., 2006). A larger number of saccades in both directions indicates difficulty associated with understanding (RAYNER, 1998; RAYNER et al., 2006; RAYNER, 2009). We analyzed gaze transitions based on areas of interest (AOI) of each program defined in Section 5.4. An attention map is a heat map that displays an aggregation of fixations.



(a) Program 1 with 2 dependent variables



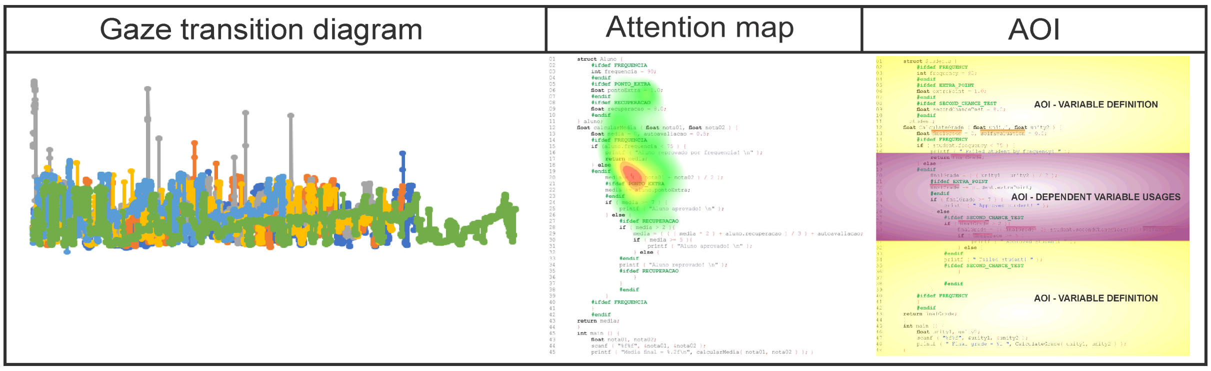
(b) Program 2 with 4 dependent variables

Figure 5.6 Gaze transitions diagram and attention map of programs 1 and 2.

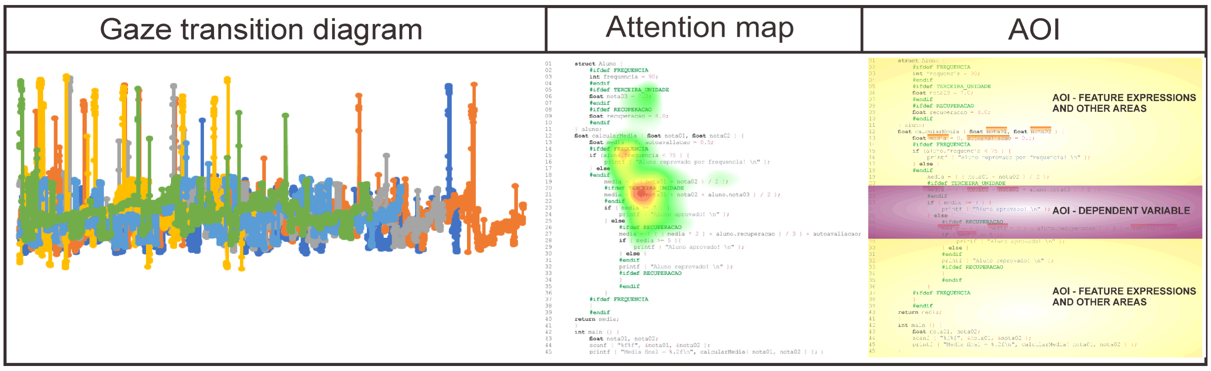
Figure 5.6a and Figure 5.6b show our gaze transitions diagrams and attention maps related to programs 1 and 2 respectively. Figure 5.7a and Figure 5.7b show our gaze transitions diagrams and attention maps related to programs 3 and 4 respectively. We superimposed all individual gaze transitions and attention maps of each participant. Each gaze transition diagram and attention map is, thus, composed by the overlapping of six individual gaze transitions and attention maps.

The gaze transitions diagrams reveal that participants executed more transitions on Program 2 (Figure 5.6b), which has four dependent variables, than on Program 1 (Figure 5.6a) with fewer dependent variables. Program 1 and Program 2 are programs developed in the same domain 1. It is possible to observe in Figure 5.6b a high number of transitions toward the top of the source code than in Figure 5.6a. We hypothesize that this occurs because programs with more dependent variables force the participant to examine more parts of the source code because dependencies are distributed to more variables.

The attention map of Program 1 (Figure 5.6a) reveals that participants' attention was directed to the usages of `finalGrade` variable. Participants concentrated their attention on the dependent variable AOI. We hypothesize that this occurs because `finalGrade` variable concentrated the most of dependencies in this program generating more cognitive effort. The attention map of Program 2 (Figure 5.6b) reveals that participants concentrated their attention in two red regions closed the `finalGrade` variable. Participants' attention was directed to the usages and definition of `finalGrade` variable. We hypothesize that this occurs because, for programs with more dependent variables, the developer needs to look at more variable definitions.



(a) Program 3 with 2 dependent variables



(b) Program 4 with 4 dependent variables

Figure 5.7 Gaze transitions diagram and attention map of programs 3 and 4.

For programs in domain 2, the gaze transitions diagrams reveal that participants executed more transitions on Program 4 (Figure 5.7b), which has four dependent variables, than Program 3 (Figure 5.7a), which has two dependent variables. This scenario is the same as domain 1. We observed in Figure 5.7b a higher number of transitions toward the top of the source code than in Figure 5.7a. Again, we hipothetize that programs with more dependent variables force developers to look more times at the region of variable definitions.

The attention map of Program 3 (Figure 5.7a) reveals that participants' attention was directed to the usages of `serviceFee` variable. The attention maps show the two red regions in the dependent variable AOI. We hypothesize that this occurs because `serviceFee` variable concentrated the most of dependencies in this program generating more attention in this area. The attention map of Program 4 (Figure 5.7b) reveals that participants, when analyzing programs with more dependent variables, needed to navigate over all source code, and, as a consequence, the attentions are more distributed along the AOIs. One AOI encapsulates variable usages and the other AOI encapsulates variable definitions.

In summary, the attention maps and the gaze transitions diagram lead to the following result.

Result 4: Programs with more dependent variables force the developer to direct their attention to more regions, causing a more spread distribution of attention and transitions over distinct parts of source code.

5.8.4 Heart-related biometrics

Previous research has shown that heart-related biometrics can be linked to difficulty in comprehending small code snippets (WALTER; PORGES, 1976; FRITZ et al., 2014; NAKAGAWA et al., 2014; MÜLLER; FRITZ, 2015). The general concepts behind these studies are that heart-related biometrics can be used to determine cognitive or mental effort required to perform a task. The more difficult a task is, the higher the cognitive effort, and the higher the Heart Rate Variability (HIJAZI et al., 2021). In this study, we examine the places in the source code where developers had a variance in Heart Rate and Stress Level and are therefore more likely to have comprehension problems.

Heart Rate Variability and Stress Level

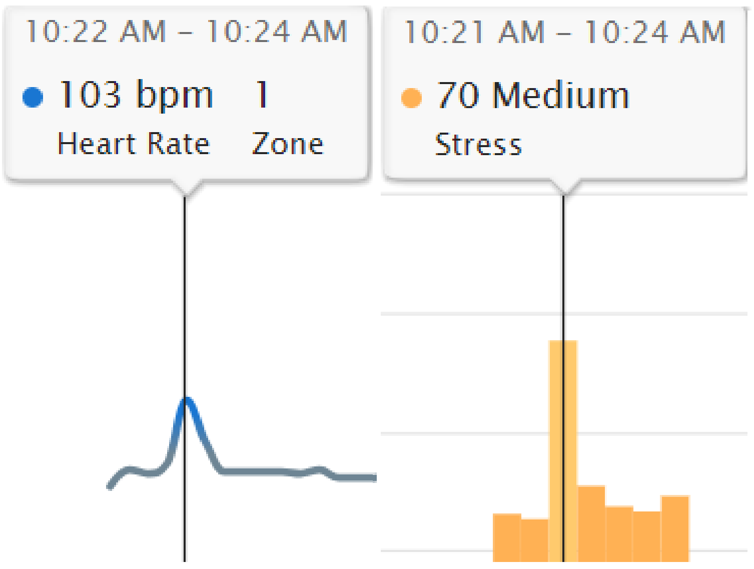
We used a low-cost smartwatch in our study. The Garmin Fenix 5s smartwatch does not provide raw data of heart rate and Stress Level. We performed our analyzes based only on calculating averages, variance, and standard deviation according to the graphs collected by the smartwatch.

Table 5.4 Total number of HRV and Stress Level variations

	With 2 dependent variables			With 4 dependent variables		
Programs	**Program 1**	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
HRV	2	2	4	4	4	8
total of participants	2	2	4	3	3	6

Figure 5.8 shows an example of data captured by the smartwatch. Visually, in Figure 5.8a we detected a variation in the heart rate of one of the participants. Between 10:22 AM and 10:24 AM the participant registered 103 heartbeats per minute. Then, we check if this value is above the mean and standard deviation. If confirmed, we check in the values of Stress Level graphics (Figure 5.8b) if the moment of variation of Stress Level is the same as the heart rate. If confirmed, we mark that point as a heart rate variation (HRV).

All HRV and Stress Level measurements were normalized using the baseline measurements that we collected during the second minute of the two-minute swimming fish video. In summary, the period evaluated for each participant for each program was from the last minute of the video until the end of the third task of each program.



(a) HRV (b) Stress Level

Figure 5.8 HRV and Stress Level collected by smartwatch

We counted the number of HRV for programs with 2 and 4 dependent variables. Our null hypothesis about this metric is:

H_0h: There is no significant difference in the number of HRV to specify the output of the programs when comparing programs with different numbers of dependent variables.

Table 5.4 shows the sum of number of HRV of all participants during the tasks and the total number of different participants accounted for these HRV. Four participants had 4 HRV for programs with 2 dependent variables and 6 participants had 8 HRV for programs with 4 dependent variables. The χ^2 test revealed no significant difference between the number of HRV needed to specify the output correctly in programs with 2 and 4 dependent variables. The value χ^2 is 0.9, and the p-values are 0.3428. Based on this, we cannot reject our null hypothesis H_0h.

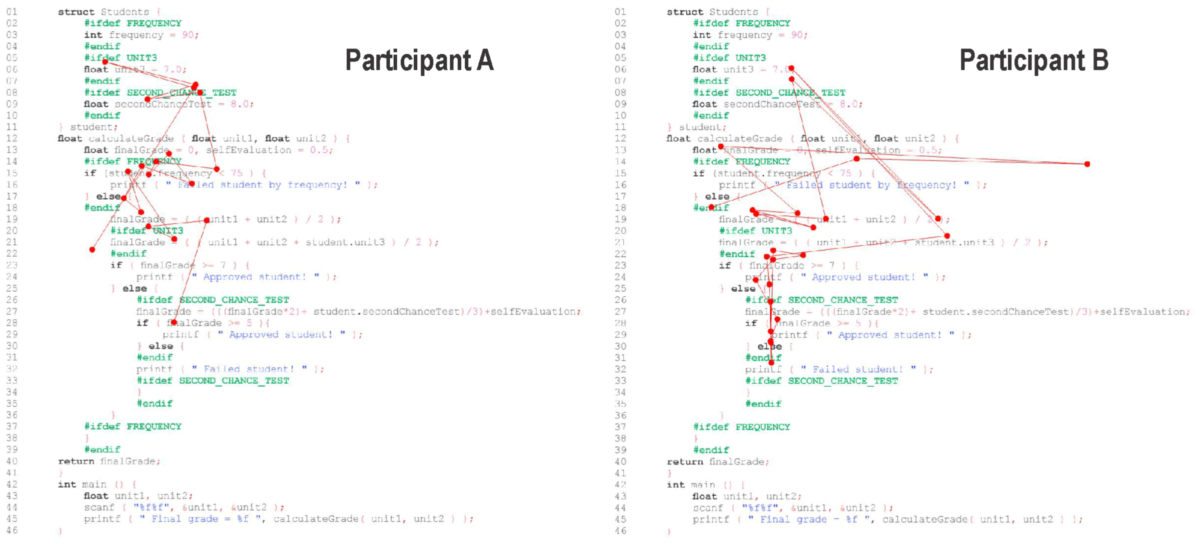
Result 5: There was no significant difference in the number of HRV until giving the correct answer when comparing programs with different numbers of dependent variables.

5.8.5 Discussion

Here we answer our research question *How do different numbers of dependent variables affect the comprehensibility of configurable system source code??* by discussing different aspects of our findings.

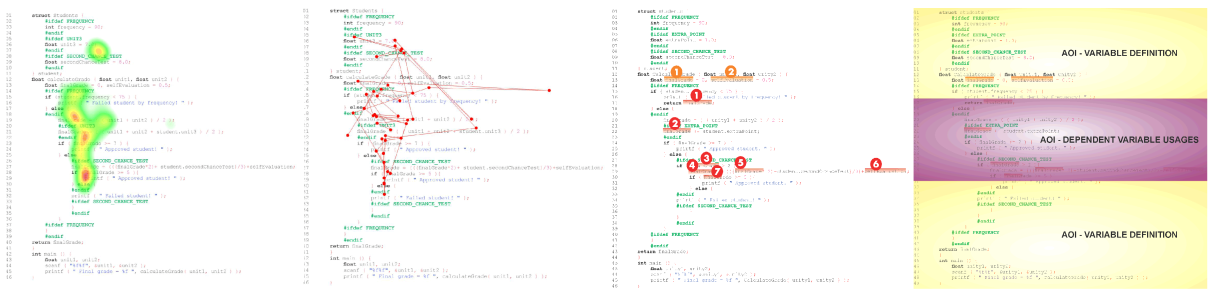
Programs with more dependent variables were more difficult to understand.

Our results 1, 3, and 4 show that programs with more dependent variables demanded more comprehension effort from participants. Developers spent more time and more fixations when compared with programs with fewer dependent variables. We hypothesize



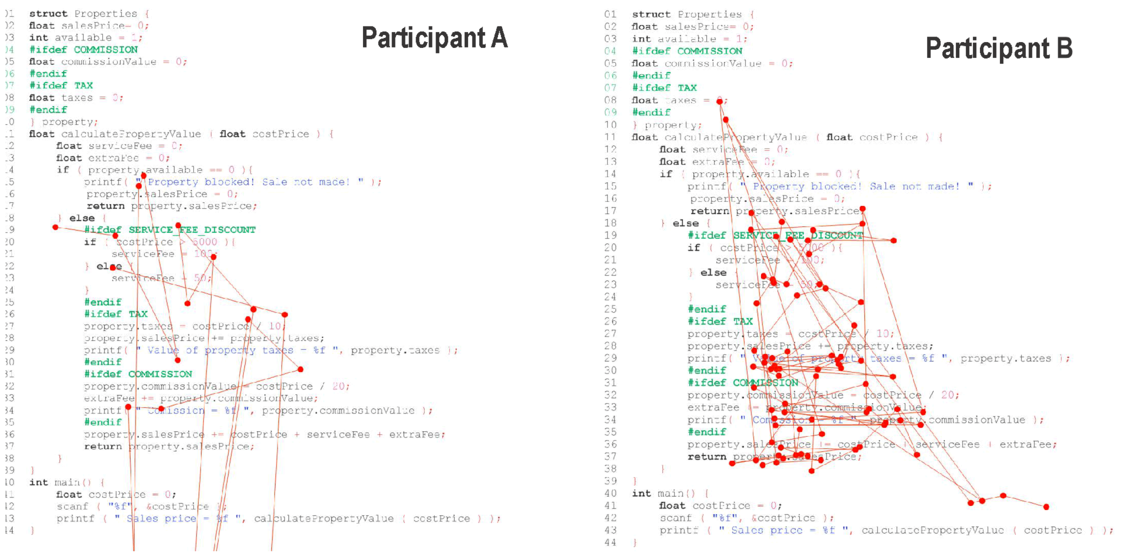
```
struct Students {
    #ifdef FREQUENCY
    int frequency = 90;
    #endif
    #ifdef UNIT3
    float unit3 = 7.0;
    #endif
    #ifdef SECOND_CHANCE_TEST
    float secondChanceTest = 8.0;
    #endif
} student;
float calculateGrade ( float unit1, float unit2 ) {
    float finalGrade = 0, selfEvaluation = 0.5;
    #ifdef FREQUENCY
    if (student.frequency < 75 ) {
        printf ( " Failed student by frequency! " );
    } else {
    #endif
        finalGrade = ( (unit1 + unit2 ) / 2 );
        #ifdef UNIT3
        finalGrade = ( ( unit1 + unit2 + student.unit3 ) / 2 );
        #endif
        if ( finalGrade >= 7 ) {
            printf ( " Approved student! " );
        } else {
            #ifdef SECOND_CHANCE_TEST
            finalGrade = (((finalGrade*2)+ student.secondChanceTest)/3)+selfEvaluation;
            if ( finalGrade >= 5 ){
                printf ( " Approved student! " );
            } else {
            #endif
            printf ( " Failed student! " );
            #ifdef SECOND_CHANCE_TEST
            }
            #endif
        }
    #ifdef FREQUENCY
    }
    #endif
return finalGrade;
}
int main () {
    float unit1, unit2;
    scanf ( "%f%f", &unit1, &unit2 );
    printf ( " Final grade = %f ", calculateGrade( unit1, unit2 ) );
}
```

(a) Individual scan path of participants analyzing Program 1.


1
2
1
2
3
4
5
6
7
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION

(b) Gaze movements of participants analyzing Program 1.

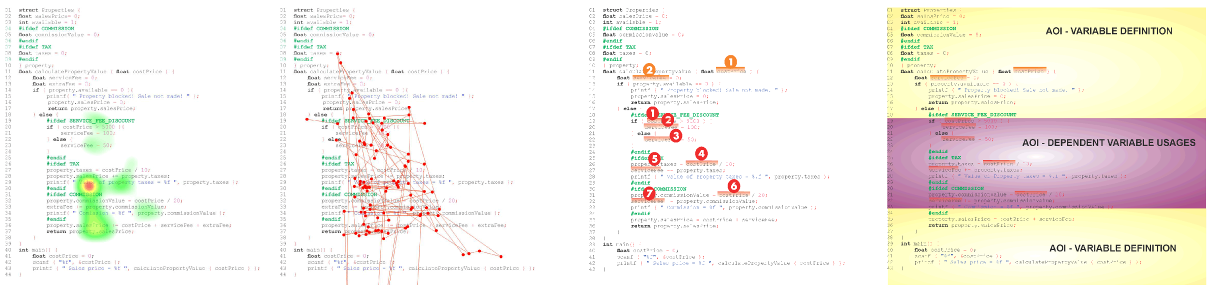
Figure 5.9 Gaze movements of Program 1 in the moment of HRV



```
struct Properties {
float salesPrice= 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
     float serviceFee = 0;
     float extraFee = 0;
     if ( property.available == 0 ){
          printf( " Property blocked! Sale not made! " );
          property.salesPrice = 0;
          return property.salesPrice;
     } else {
          #ifdef SERVICE_FEE_DISCOUNT
          if ( costPrice > 5000 ){
               serviceFee = 100;
          } else {
               serviceFee = 50;
          }
          #endif
          #ifdef TAX
          property.taxes = costPrice / 10;
          property.salesPrice += property.taxes;
          printf( " Value of property taxes = %f ", property.taxes );
          #endif
          #ifdef COMMISSION
          property.commissionValue = costPrice / 20;
          extraFee += property.commissionValue;
          printf( " Comission = %f ", property.commissionValue );
          #endif
          property.salesPrice += costPrice + serviceFee + extraFee;
          return property.salesPrice;
     }
}
int main() {
     float costPrice = 0;
     scanf ( "%f", &costPrice );
     printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

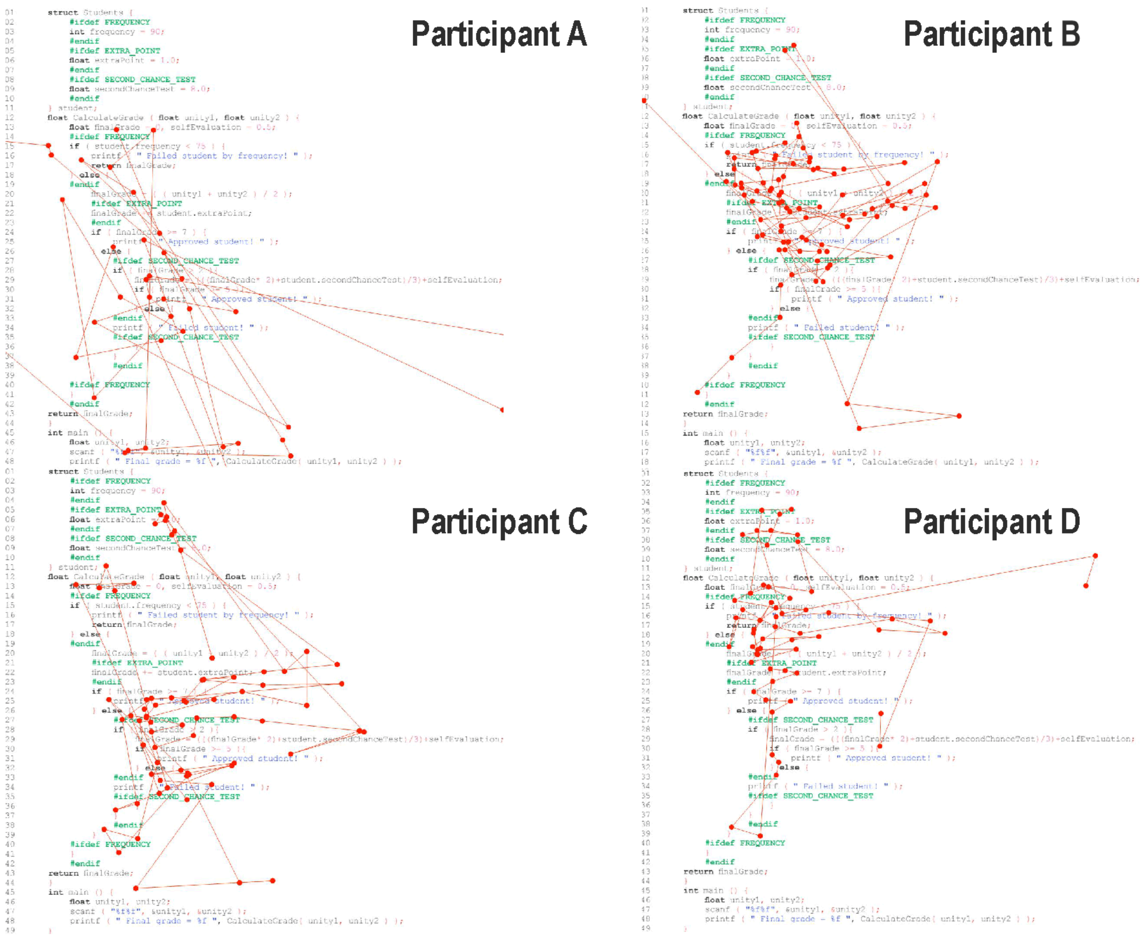
```
struct Properties {
float salesPrice= 0;
int available = 1;
#ifdef COMMISSION
float commissionValue = 0;
#endif
#ifdef TAX
float taxes = 0;
#endif
} property;
float calculatePropertyValue ( float costPrice ) {
     float serviceFee = 0;
     float extraFee = 0;
     if ( property.available == 0 ){
          printf( " Property blocked! Sale not made! " );
          property.salesPrice = 0;
          return property.salesPrice;
     } else {
          #ifdef SERVICE_FEE_DISCOUNT
          if ( costPrice > 5000 ){
               serviceFee = 100;
          } else {
               serviceFee = 50;
          }
          #endif
          #ifdef TAX
          property.taxes = costPrice / 10;
          property.salesPrice += property.taxes;
          printf( " Value of property taxes = %f ", property.taxes );
          #endif
          #ifdef COMMISSION
          property.commissionValue = costPrice / 20;
          extraFee += property.commissionValue;
          printf( " Commission = %f ", property.commissionValue );
          #endif
          property.salesPrice += costPrice + serviceFee + extraFee;
          return property.salesPrice;
     }
}
int main() {
     float costPrice = 0;
     scanf ( "%f", &costPrice );
     printf ( " Sales price = %f ", calculatePropertyValue ( costPrice ) );
}
```

(a) Individual scan path of participants analyzing Program 3.


1
2
3
4
5
6
7
AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION

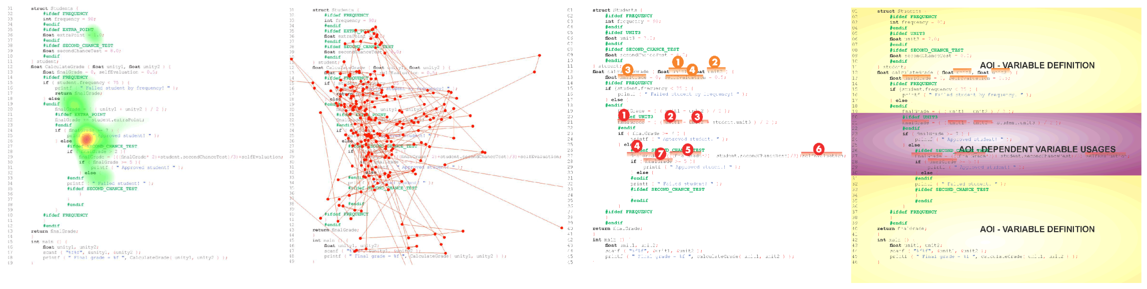
(b) Gaze movements of participants analyzing Program 3.

Figure 5.10 Gaze movements of Program 3 in the moment of HRV



```
struct Students {
    #ifdef FREQUENCY
    int frequency = 90;
    #endif
    #ifdef EXTRA_POINT
    float extraPoint = 1.0;
    #endif
    #ifdef SECOND_CHANCE_TEST
    float secondChanceTest = 8.0;
    #endif
} student;
float CalculateGrade ( float unity1, float unity2 ) {
    float finalGrade = 0, selfEvaluation = 0.5;
    #ifdef FREQUENCY
    if ( student.frequency < 75 ) {
        printf ( " Failed student by frequency! " );
        return finalGrade;
    } else {
    #endif
        finalGrade = ( ( unity1 + unity2 ) / 2 );
        #ifdef EXTRA_POINT
        finalGrade += student.extraPoint;
        #endif
        if ( finalGrade >= 7 ) {
            printf ( " Approved student! " );
          } else {
            #ifdef SECOND_CHANCE_TEST
            if ( finalGrade > 2 ) {
                finalGrade = (((finalGrade* 2)+student.secondChanceTest)/3)+selfEvaluation;
                if ( finalGrade >= 5 ) {
                    printf ( " Approved student! " );
                  } else {
            #endif
            printf ( " Failed student! " );
            #ifdef SECOND_CHANCE_TEST
                }
            }
            #endif
          }
    #ifdef FREQUENCY
    }
    #endif
return finalGrade;
}
int main () {
    float unity1, unity2;
    scanf ( "%f%f", &unity1, &unity2 );
    printf ( " Final grade = %f ", CalculateGrade( unity1, unity2 ) );
}
```

(a) Individual scan path of participants from Program 2 with 4 dependent variables.



(b) Gaze movements diagram of Program 2 with 4 dependent variables.

Figure 5.11 Gaze movements of Program 2 in the moment of HRV

that this occurred because, with more dependent variables, the developer needs to look at more variable definitions. Furthermore, if the local of dependent variables definitions are far from their usage participant realize longer gaze transitions. This find confirms our results in Study 2 (Chapter 4).

Programs with more dependent variables required more visual effort. Results 3 and 4 show that programs with more dependent variables may increase visual effort. To specify the output of the programs, participants needed to perform more gaze transitions and focused more time on distinct parts of the source code.

There was no significant difference in the number of HRV when comparing programs with different numbers of dependent variables.

Result 5 shows that there was no significant difference in terms of HRV while participants analyzed programs with different numbers of dependent variables. Despite this, we decided to investigate what was happening in terms of gaze movements during HRV, as follows.

Scan paths of HRV

Scan paths are the sequence of all fixations points of a participant as a connect-the-dots visualization. We generated individual scan paths of all participants when happened a variation in the heart rate. Figure 5.8 shows an example of HRV of one of the participants. Based on Figure 5.8, what could have caused a variation in the participant's heart rate during source code analysis? We used scan paths to identify where he or she was looking and why it might have generated cognitive effort.

Table 5.4 shows that 2 participants had 2 HRV analyzing Program 1 (2 dependent variables). We generated two scan paths at the moments the two HRVs happened. In both cases, the participants where executing tasks with all features enabled. Figure 5.9a shows two scan paths of participants analyzing Program 1 (2 dependent variables). The Participants looked for information about variable definitions at the top of the code. After that, participants followed the normal flow of reading. In Figure 5.9b we superimposed the two individual scan paths corresponding to HRV. We also generated attention maps corresponding to the HRV of the participants. Figure 5.9b shows the gaze movements of participants were concentrated in the region of variable definitions and variable usages when an HRV occurred. Three red regions indicate that participants' attention was directed to the definition and usages of `finalGrade` variable. We hypothesize that this occurs because `finalGrade` variable concentrated the most of dependencies in this program generating more cognitive effort.

Figure 5.10a represents scan paths of participants in Program 3, another program with 2 dependent variables. Two participants register HRV. In the first scan path, the participant had to consider all features disabled. We observed that the participant realized long saccades and regressions to the bottom of the screen, exceeding the limits of the screen. We suppose that the participant did not well understand the task and had to look to the instructions in the paper sometimes. This may have caused the HRV. In the second scan path, the participant had to consider the `TAX` feature disabled. The participant navigated along the source code trying to resolve the task. Figure 5.10b is composed by the overlapping of the two individual scan paths and attention maps during the HRV. The red region on `COMISSION` feature indicates that participants' attention was directed to the definition and usages of variables of the enabled feature. We hypothesize that HRV occurred because analyzing programs with disabled features is not a trivial task. Participants concentrated their gaze movements in #ifdef of `COMISSION` feature and around `serviceFee` dependent variable.

Figure 5.11a shows scan paths participants performed in Program 2 (4 dependent variables). The participants performed long saccades and regressions between variable definitions and variable usages. In Figure 5.11b we had the overlapping of four individual scan paths and attention maps corresponding to participants' gaze movements during the four HRV occurrences. Figure 5.11b confirms that gaze movements concentrated in the region of variable definitions and variable usages when an HRV occurred. The red region indicates that participants' attention was directed to the usage of `finalGrade` variable. Figure 5.11b shows only one red region in programs with 4 dependent variables, and Figure 5.9b shows four red regions in programs with 2 dependent variables. We hypothesize that this occurs because programs with more dependent variables force the participant to examine more parts of the source code because dependencies are distributed to more variables. The same happens in Program 4 also with 4 dependent variables. Participants performed long saccades and regressions causing a higher distribution of attention over distinct parts of source code as shown in Figures 5.12a and 5.12b.

In summary, participants seem to have performed long saccades and regressions during HRV. Also, HRV concentrated fixations in the region of variable definitions and variable usages. However, it is important to highlight that these findings were based on limited observations and information.

Number of dependent variables did not affect number of attempts to specify Program outputs.

Result 2 revealed that the number of attempts needed for the participants until giving the correct answer was not affected by different numbers of dependent variables. This means that number of dependent variables may increase time and visual effort to specify the output, but does not decrease developers' ability to specify the output. This result confirms previous studies that showed that most participants correctly executed tasks in programs with #ifdef (MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017; SANTOS; SANT'ANNA, 2019).

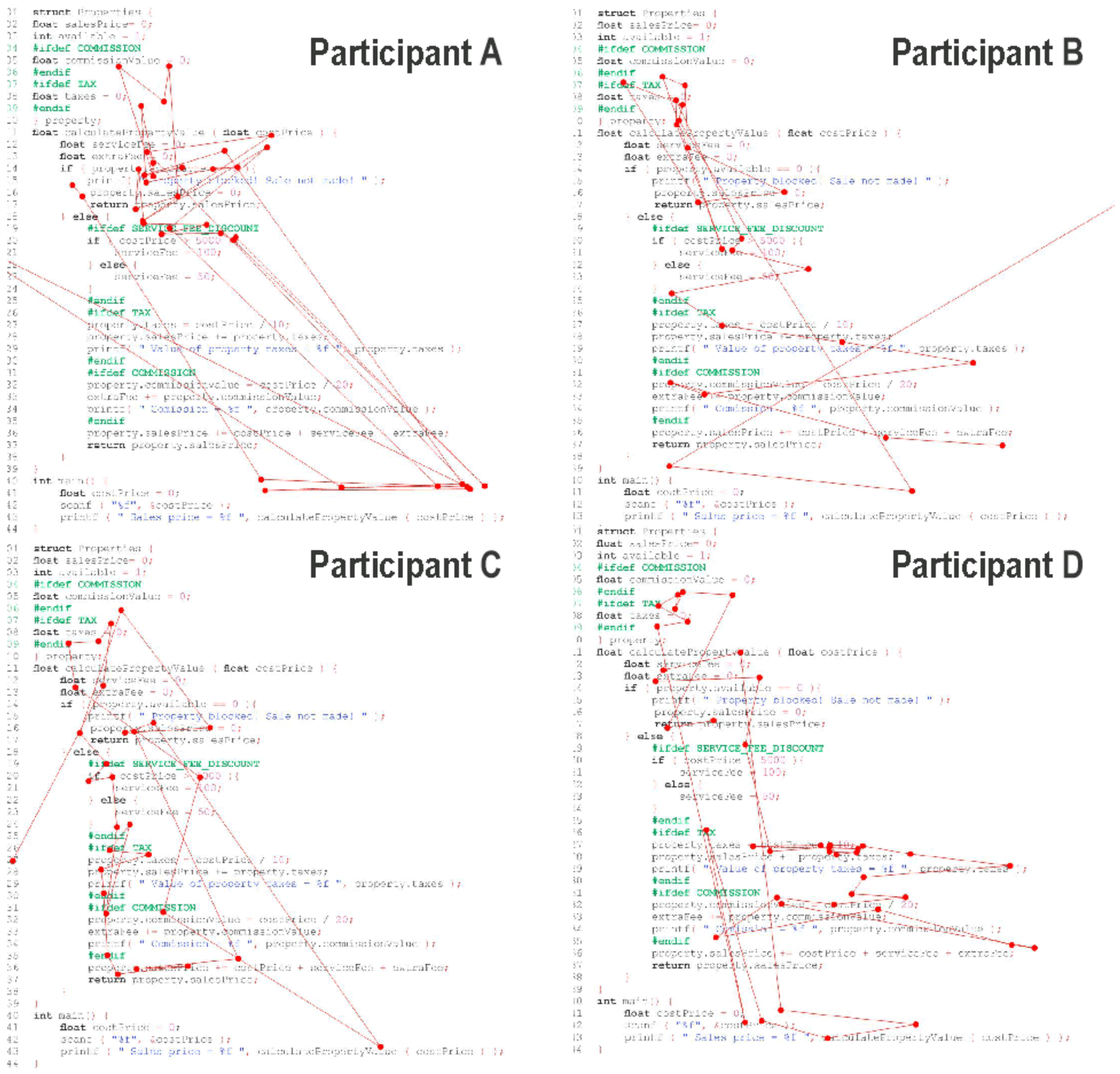
5.9 THREATS TO VALIDITY

This experiment design is similar to the experiment design of Study 2 describes in Chapter 4, thus, the threats to validity of this study is similar to the threats to validity described in Section 4.6.

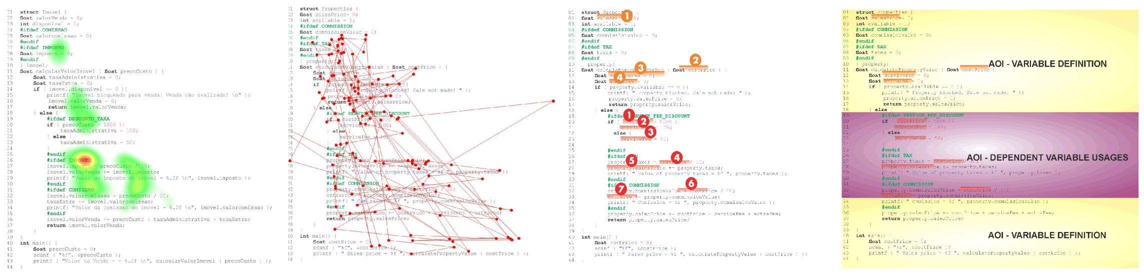
5.9.1 Internal validity

Programming language. We wrote our programs in C, because most of the studies and repositories of programs containing #ifdef are in C language. The knowledge of participants in C could influence our results. To minimize that, we only admitted participants with previous experience on C.

Participants' experience. No participant was an expert in #ifdef, thus the distribution of participants in the Latin square was balanced. We controlled confounding factors via the Latin square design and randomization. We selected 12 participants experts in language C. They are programming language professors and developers from



(a) Individual scan path of participants from Program 4 with 4 dependent variables.


AOI - VARIABLE DEFINITION
AOI - DEPENDENT VARIABLE USAGES
AOI - VARIABLE DEFINITION

(b) Gaze movements diagram of Program 4 with 4 dependent variables.

Figure 5.12 gaze movements of Program 4 in the moment of HRV

industry.

Lab settings. All experiment trials were done in the same lab and with our supervision. We observed temperature and brightness conditions. At the moment of the execution of the experiment, the lab had only the participant and the authors.

5.9.2 External validity

Programs. Due to limitations we used small programs. But, our programs were inspired by real configurable systems. For this reason, our results may hold to other programs. However, for programs over 45 lines of code and more than three features, there may be additional effects that we have not observed.

Meaningful variables names. The maintenance and comprehensibility of source code can be hindered by choosing variables with non-meaningful names. Choosing meaningful variable names is also important for domain comprehension. To minimize this threat, we select relevant and meaningful variable names.

Mental simulation of scenarios. In practice, programmers encounter codes containing many features, and they don't always test or compile all possible configurations. Our results are limited to programs with a few features where the developer mentally simulates all possible configurations. Additionally, we utilize functional features, which are easier to analyze than architectural features, for example.

Lab settings. Our results are also limited to the environment we adopted. Participants did not interact with the source code or use tools or IDEs. However, this design would not be attractive for many participants, since it would require more time for execution. In addition, we have the limitation that the source code should fit on the screen due to the eye-tracking device. We used a default color scheme and font style in most Integrated Development Environments (IDEs), such as Eclipse, CodeBlocks, DevC++, and Microsoft Visual Studio.

5.9.3 Construct validity

Comprehensibility measurement. Measuring comprehensibility is not trivial because it involves human factors. Therefore, it is always a threat to construct validity. To minimize this threat, we quantified comprehensibility by means of different metrics, all of them already been used in previous studies.

Chapter

6

STUDY 4 - AN EXPERIMENT ON HOW DEGREES OF VARIABILITY AFFECT PROGRAM COMPREHENSIBILITY

Our previous studies showed that feature dependencies affect comprehensibility in different degrees (Chapter 3) depending on the type of feature dependency (Chapter 4) or number of dependent variables (Chapter 5). These studies focused on dependent variables. Other aspects also need to be considered when referring to feature dependencies, like degrees of variability. In this context, we decided to complement our investigation by answering the following research question:
-- *How do degrees of variability affect the comprehensibility of configurable system?*

Degrees of variability in configurable systems refer to the number of different program variants from common source code. In other words, degrees of variability mean programs with different numbers of features. For programs with less variability, the developer can control whether to include or exclude a fewer number of features and consequently it has fewer possible scenarios to be analyzed. For programs with more variability, the developer needs to analyze more features and consequently a greater number of possible scenarios. We described variability in Section 2.1.

6.1 DESIGN

To answer our research question, we carried out a controlled experiment with 12 developers, who analyzed programs trying to specify their output.

We performed this experiment with the same participants of Study 3 (Chapter 5). We took advantage of the availability of the participants and also executed the tasks of this experiment just after the tasks of Study 3. So, we controlled confounding factors using the same experimental design and procedures of Study 3 (Section 5.1).

Developers analyzed 4 similar programs in terms of lines of code, cyclomatic complexity, and number of feature dependencies. The differences between the programs were the domain and degrees of variability. We implemented different degrees of variability with variations in the number of feature expressions and feature constants. We considered

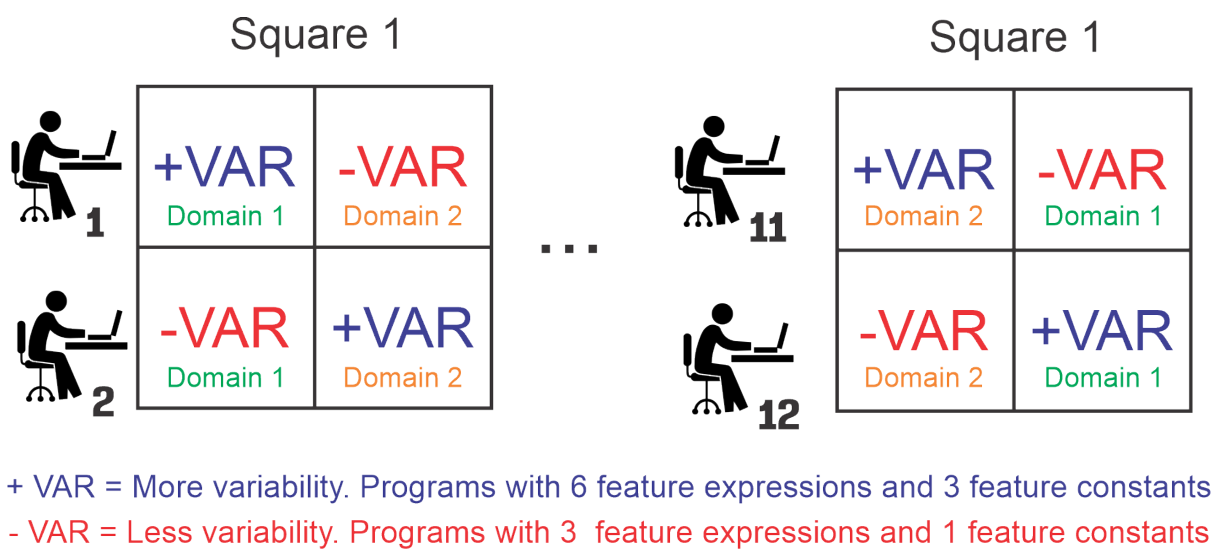

Square 1
+VAR
Domain 1
-VAR
Domain 2
1
-VAR
Domain 1
+VAR
Domain 2
2
...
Square 1
+VAR
Domain 2
-VAR
Domain 1
11
-VAR
Domain 2
+VAR
Domain 1
12
+ VAR = More variability. Programs with 6 feature expressions and 3 feature constants
- VAR = Less variability. Programs with 3 feature expressions and 1 feature constants

Figure 6.1 Latin Square design (2x2).

programs with more variability all the programs implemented with 6 feature expressions and 3 feature constants, and programs with less variability all the programs implemented with 3 feature expressions and 1 feature constant. This should make any performance differences manifest themselves clearly. We used programs with a maximum of 6 feature expressions due to display limitations of the code on the screen.

We compared the comprehension effort participants spent to analyze each program by (i) time to analyze each program, (ii) number of attempts until developers provide the correct answer, (iii) visual effort and, (iv) heart-related biometrics. Participants used eye-tracking and a smartwatch to collect data. In order to avoid learning effect, we selected two different domains. Domain 1 (sale of products) and Domain 2 (game of hit the target) each one with 6 dependencies. We fixed the number of feature dependencies and the number of dependent variables on all programs and varied degrees of variability. Then, two configurable system programs were implemented with less variability and the other two programs were implemented with more variability. In summary, for each domain, we implemented two programs with different degrees of variability.

We designed our experiment as a standard Latin Square. Our previous studies discuss Latin Square design (Section 3.1 and Section 4.1). Figure 6.1 explains our 2x2 Latin Square. In its columns, we have the treatment, in this case, the number of feature expressions and feature constant. The lines represent developers. The acronyms in the cells represent the program characteristics, +VAR represents a program with more variability, and -VAR represents a program with less variability. Developer 1 firstly analyzed a program with more variability, and, afterward, a program with less variability. Developer 2 also analyzed both programs, but in reverse order.

To avoid learning effects due repetition of domains we distributed the domains along our Latin square column as in our Study 3 (Section 5.1). Thus, each developer analyzed two different programs, each one in a domain different to the other.

Listing 6.1 Program 1: Sale of products domain with 3 feature expressions and 1 feature constant

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef PRODUCT_CONTROL
    int totalProductsPurchased = 20;
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
    float sellProduct ( int numberOfProducts ) {
        float unitaryValue = 5.0;
        #ifdef PRODUCT_CONTROL
        if ( product.totalProductsForSale - numberOfProducts < 0 ) {
            printf( "Insufficient number of products!" );
            return 0;
        } else {
            if ( product.totalProductsForSale < product.
                minimumNumberProducts ) {
                printf ( "Last units! Readjusted price" );
                unitaryValue = unitaryValue + 2.0;
                product.totalProductsForSale += product.
                    totalProductsPurchased;
            }
            #endif
            printf( "Product sold." );
            product.totalProductsForSale -= numberOfProducts;
            product.totalProductsSold += numberOfProducts;
        #ifdef PRODUCT_CONTROL
        }
        #endif
        return numberOfProducts * unitaryValue;
    }
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale price = %f", sellProduct ( numberOfProducts ) );
}
```

6.2 PARTICIPANTS

In total 12 participants run our experiment: two graduates, three postgraduate, one postgraduate student, four with master's degrees, and two with doctoral degrees. Six of participants are developers from the industry and six are professors. Regarding the experience with programming languages, all participants reported having experience with C for more than six years. Regarding their #ifdef background knowledge, two participants are researchers working on topics related to #ifdef, and all of them reported having some experience with source code containing #ifdef.

6.3 PROGRAMS

We implemented the programs for this experiment inspired by our previous studies (Section 3.3 and Section 4.3) and common programming tasks. In the following, we describe each program.

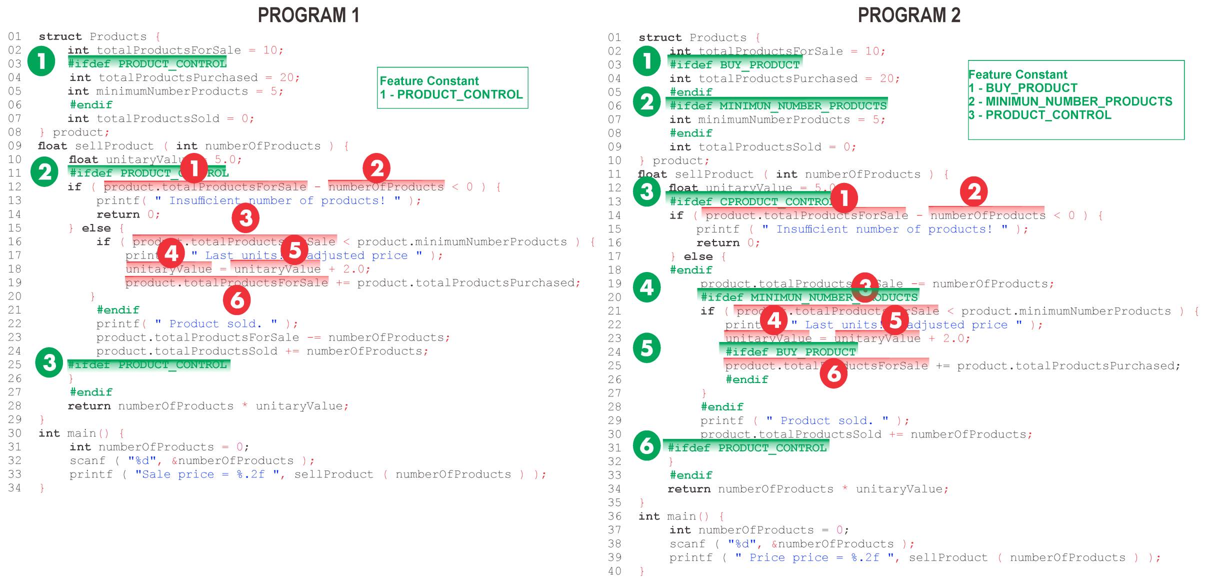
Listing 6.2 Program 2: Sale of products domain with 6 feature expressions and 3 feature constant

```c
struct Products {
    int totalProductsForSale = 10;
    #ifdef BUY_PRODUCT
    int totalProductsPurchased = 20;
    #endif
    #ifdef MINIMUN_NUMBER_PRODUCTS
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef PRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf ( "Insufficient number of products!" );
        return 0;
    } else {
    #endif
        product.totalProductsForSale -= numberOfProducts;
        #ifdef MINIMUN_NUMBER_PRODUCTS
        if ( product.totalProductsForSale < product.
           minimumNumberProducts ) {
            printf ( "Last units! Readjusted price" );
            unitaryValue = unitaryValue + 2.0;
            #ifdef BUY_PRODUCT
            product.totalProductsForSale += product.
               totalProductsPurchased;
            #endif
        }
        #endif
        printf ( "Product sold." );
        product.totalProductsSold += numberOfProducts;
        #ifdef PRODUCT_CONTROL
    }
        #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale Price = %f", sellProduct (numberOfProducts));
}
```

Program 1: sale of products with less variability. Listing 6.1 shows the source code of Program 1. It has only one feature constant labeled `PRODUCT_CONTROL`. The feature `PRODUCT_CONTROL` verifies the minimum number of products available for sale, controls the number of products available for sale, and increases products for sale. This program has 3 feature expressions. The first one is in line 3, the second one in line 11, and the last one in line 25.

Listing 6.3 Program 3: Game of hit the target domain with 3 feature expressions and 1 feature constant

```c
struct Players {
    int totalPoints = 0;
    int hits = 0;
    #ifdef HARD
    int attempt = 0;
    int errors = 0;
    #endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    if ( distance > 100 ) {
        printf ( "Hit the target" );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef HARD
    player.attempt ++;
    if ( distance >= 80 ) {
        roundPoints += 60;
    } else
    if ( distance >= 60 && distance < 80 ) {
        roundPoints -= 30;
        player.errors ++;
    }
    #endif
    else {
        printf ( "Missed the target" );
        roundPoints -= 50;
    }
    #ifdef HARD
    printf ( "Hit percentage = %f", player.hits / player.attempt);
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance );
    printf ( "Total points = %d", calculatePoints ( distance ) );
}
```



PROGRAM 1

Feature Constant
1 - PRODUCT_CONTROL

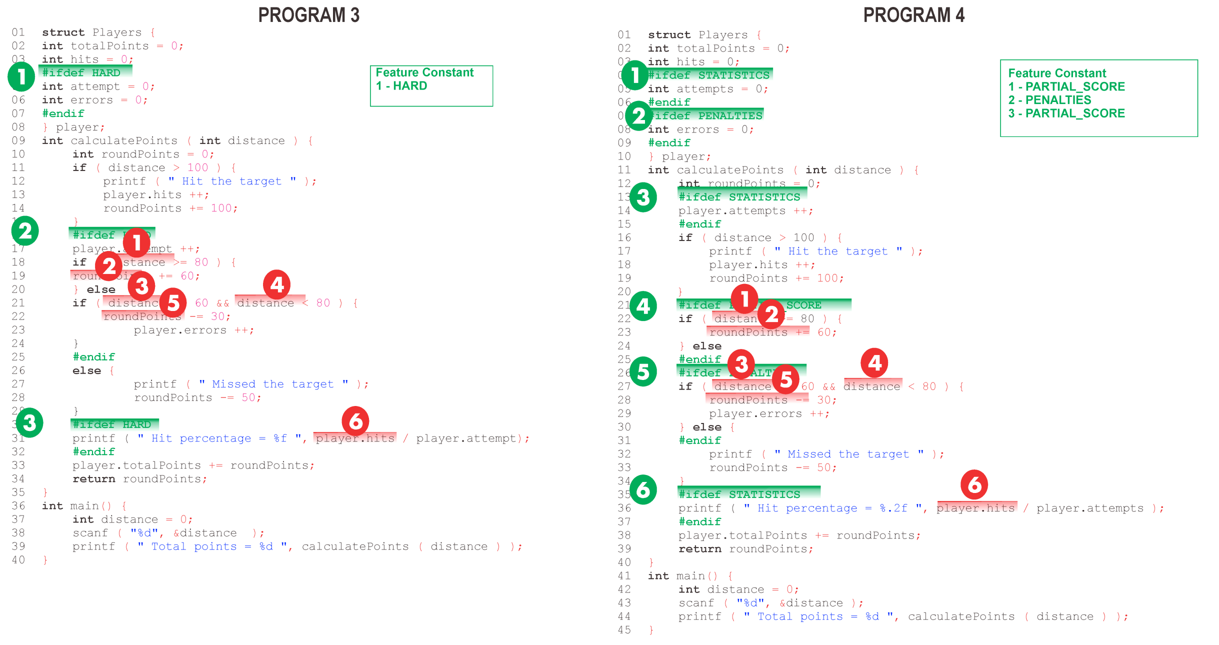
```
struct Products {
    int totalProductsForSale = 10;
    #ifdef PRODUCT_CONTROL
    int totalProductsPurchased = 20;
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef PRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf( " Insufficient number of products! " );
        return 0;
    } else {
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf( " Last units! adjusted price " );
            unitaryValue = unitaryValue + 2.0;
            product.totalProductsForSale += product.totalProductsPurchased;
        }
        #endif
        printf( " Product sold. " );
        product.totalProductsForSale -= numberOfProducts;
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale price = %.2f ", sellProduct ( numberOfProducts ) );
}
```

PROGRAM 2

Feature Constant
1 - BUY_PRODUCT
2 - MINIMUN_NUMBER_PRODUCTS
3 - PRODUCT_CONTROL

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef BUY_PRODUCT
    int totalProductsPurchased = 20;
    #endif
    #ifdef MINIMUN_NUMBER_PRODUCTS
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef CPRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf ( " Insufficient number of products! " );
        return 0;
    } else {
    #endif
        product.totalProductsForSale -= numberOfProducts;
        #ifdef MINIMUN_NUMBER_PRODUCTS
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf ( " Last units! adjusted price " );
            unitaryValue = unitaryValue + 2.0;
            #ifdef BUY_PRODUCT
            product.totalProductsForSale += product.totalProductsPurchased;
            #endif
        }
        #endif
        printf ( " Product sold. " );
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( " Price price = %.2f ", sellProduct ( numberOfProducts ) );
}
```

Figure 6.2 Program 1 and Program 2 on domain 1 highlighting feature expressions, feature constants, and dependent variables.



PROGRAM 3

Feature Constant
1 - HARD

```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef HARD
int attempt = 0;
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef HARD
    player.attempt ++;
    if ( distance >= 80 ) {
    roundPoints += 60;
    } else
    if ( distance > 60 && distance < 80 ) {
        roundPoints -= 30;
            player.errors ++;
    }
    #endif
    else {
            printf ( " Missed the target " );
            roundPoints -= 50;
    }
    #ifdef HARD
    printf ( " Hit percentage = %f ", player.hits / player.attempt);
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance  );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

PROGRAM 4

Feature Constant
1 - PARTIAL_SCORE
2 - PENALTIES
3 - PARTIAL_SCORE

```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef STATISTICS
int attempts = 0;
#endif
#ifdef PENALTIES
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    #ifdef STATISTICS
    player.attempts ++;
    #endif
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef PARTIAL_SCORE
    if ( distance >= 80 ) {
        roundPoints += 60;
    } else
    #endif
    #ifdef PENALTIES
    if ( distance > 60 && distance < 80 ) {
        roundPoints -= 30;
        player.errors ++;
    } else {
    #endif
        printf ( " Missed the target " );
        roundPoints -= 50;
    }
    #ifdef STATISTICS
    printf ( " Hit percentage = %.2f ", player.hits / player.attempts );
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

Figure 6.3 Program 3 and Program 4 on domain 2 highlighting feature expressions, feature constant and dependent variables.

Program 2: sale of products with more variability. Listing 6.2 shows the source code of Program 2. It has three feature constants: `BUY_PRODUCT`, `PRODUCT_CONTROL`, and `MINIMUN_NUMBER_PRODUCTS`. The feature `BUY_PRODUCT` increases products for sale. The feature `MINIMUN_NUMBER_PRODUCTS` verifies the minimum number of products available for sale, and the feature `PRODUCT_CONTROL` controls the number of products available for sale. This program has 6 feature expressions defined in lines 3, 6, 13, 20, 24, and 31.

Listing 6.4 Program 4: Game of hit the target domain with 6 feature expressions and 3 feature constants

```
 1 struct Players {
 2     int totalPoints = 0;
 3     int hits = 0;
 4     #ifdef STATISTICS
 5     int attempts = 0;
 6     #endif
 7     #ifdef PENALTIES
 8     int errors = 0;
 9     #endif
10 } player;
11 int calculatePoints ( int distance ) {
12     int roundPoints = 0;
13     #ifdef STATISTICS
14     player.attempts ++;
15     #endif
16     if ( distance > 100 ) {
17         printf ( "Hit the target" );
18         player.hits ++;
19         roundPoints += 100;
20     }
21     #ifdef PARTIAL_SCORE
22     if ( distance >= 80 ) {
23         roundPoints += 60;
24     } else
25     #endif
26     #ifdef PENALTIES
27     if ( distance >= 60 && distance < 80 ) {
28         roundPoints -= 30;
29         player.errors ++;
30     } else {
31     #endif
32         printf ( "Missed the target" );
33         roundPoints -= 50;
34     }
35     #ifdef STATISTICS
36     printf ( "Hit percentage = %f", player.hits / player.attempts );
37     #endif
38     player.totalPoints += roundPoints;
39     return roundPoints;
40 }
41 int main() {
42     int distance = 0;
43     scanf ( "%d", &distance );
44     printf ( "Total points = %d ", calculatePoints ( distance ) );
45 }
```

Figure 6.2 shows Program 1 and Program 2 side by side highlighting feature expressions, feature constants, and dependent variables. In green are the feature expressions and in red are the dependent variable usage points.

Program 3: game of hit the target with less variability. Listing 6.3 shows the source code of Program 3. It has only one feature constant labeled `HARD`. The feature `HARD` calculates the partial points and penalties of players and calculates statistics. This program has 3 feature expressions. The first one is in line 4, the second one in line 16, and the last one in line 30.

Program 4: game of hit the target with more variability. Listing 6.4 shows the source code of Program 4. It has three feature constants: `STATISTICS`, `PENALTIES`, and `PARTIAL_SCORE`. The feature `STATISTICS` calculates statistics. The feature `PENALTIES` calculates penalties of players, and the feature `PARTIAL_SCORE` calculates partial points of players. This program has 6 feature expressions defined in lines 4, 7, 13, 21, 26, and 35.

Figure 6.3 shows Program 3 and Program 4 side by side highlighting feature expressions, feature constants, and dependent variables. In green are the feature expressions and in red are the dependent variable usage points.

6.4 AREAS OF INTEREST (AOI)

Our AOIs in this study comprise regions of source code that contain feature expressions and dependent variable usages. We decided to include dependent variable usage in these AOIs because, as we are studying feature dependency, we want to know how dependent variables attract attention of participants when programs have different degrees of variability.

Figure 6.4 and Figure 6.5 show our AOIs. We define two main AOIs. The first AOI is the region of feature expressions, and the second AOI is the region of dependent variable usages.

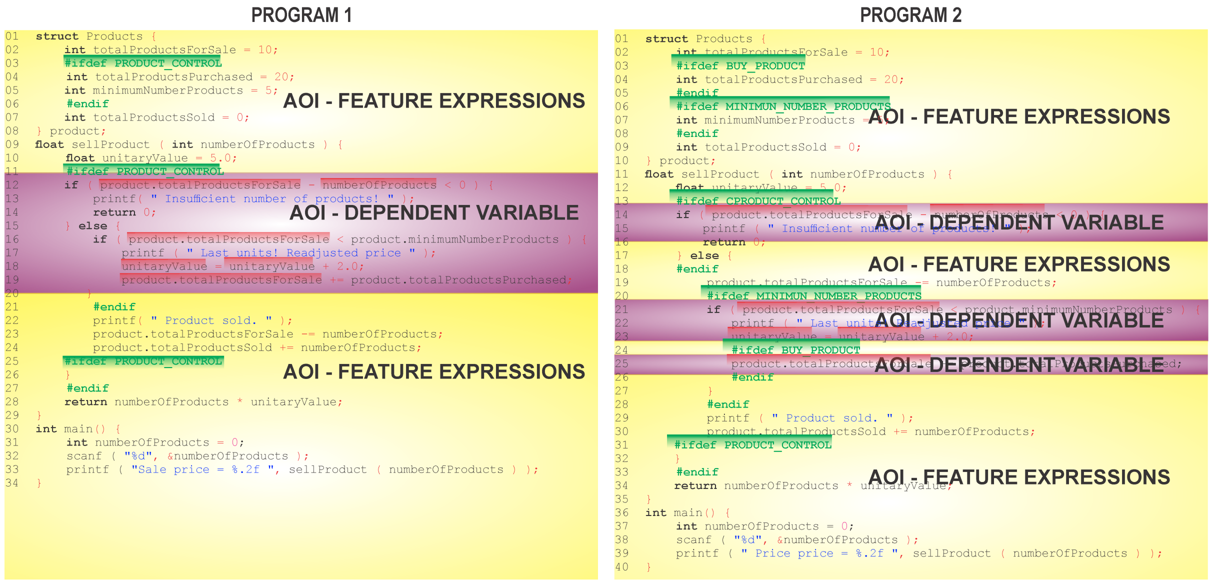
6.5 EXPERIMENT PROCEDURES

Before executing the actual experiment, we carried out two pilot studies. They were described in Section 5.5. The procedures of this study were the of Study 3, described in Section 5.6.

6.6 TASKS

This experiment was carried out together with the experiment of study 3 (Chapter 5). Thus, the tasks of both experiments should be similar to avoid confounding factors. For example, different types of tasks could have generated discomfort or apprehension in participants, demanding more time for new tasks and more explanations of new procedures. Thus, we decided to also have, for this study, three tasks in which participants should specify the output for each program.

Each participant had to understand and realize the mental execution of two programs, one program with more variability and the other with less variability. The order of


PROGRAM 1
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS
PROGRAM 2
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS

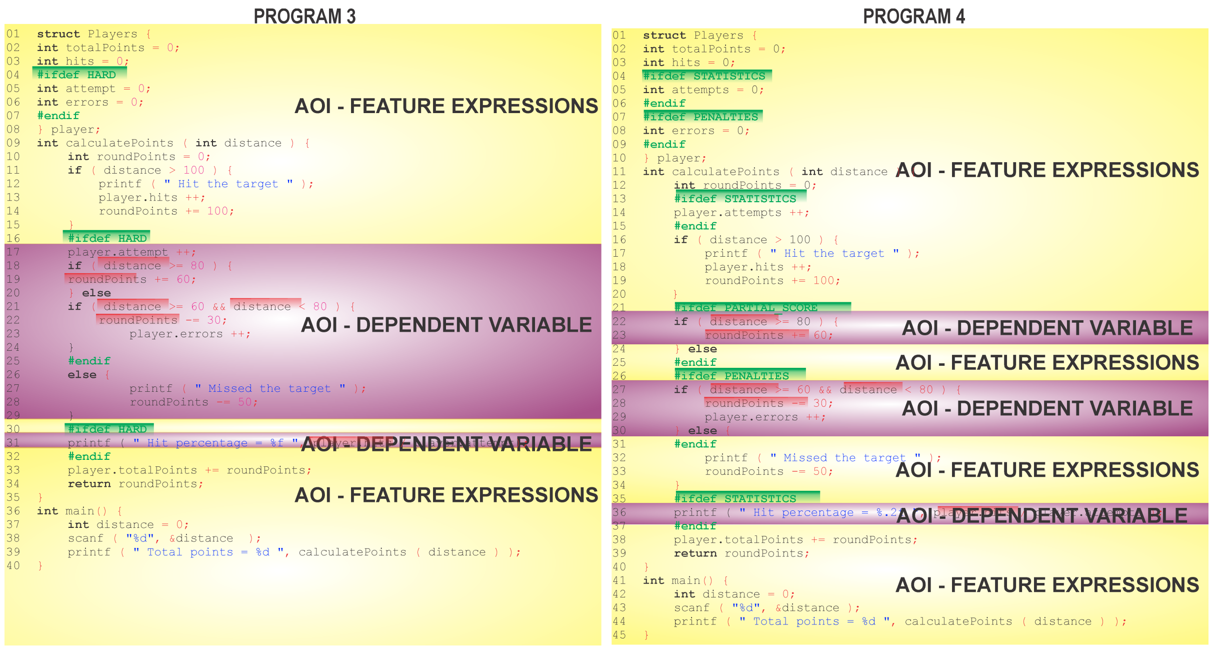
PROGRAM 1

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef PRODUCT_CONTROL
    int totalProductsPurchased = 20;
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef PRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf( " Insufficient number of products! " );
        return 0;
    } else {
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            product.totalProductsForSale += product.totalProductsPurchased;
        }
        #endif
        printf( " Product sold. " );
        product.totalProductsForSale -= numberOfProducts;
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale price = %.2f ", sellProduct ( numberOfProducts ) );
}
```

PROGRAM 2

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef BUY_PRODUCT
    int totalProductsPurchased = 20;
    #endif
    #ifdef MINIMUN_NUMBER_PRODUCTS
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef CPRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf ( " Insufficient number of products! " );
        return 0;
    } else {
    #endif
        product.totalProductsForSale -= numberOfProducts;
        #ifdef MINIMUN_NUMBER_PRODUCTS
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            #ifdef BUY_PRODUCT
            product.totalProductsForSale += product.totalProductsPurchased;
            #endif
        }
        #endif
        printf ( " Product sold. " );
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( " Price price = %.2f ", sellProduct ( numberOfProducts ) );
}
```

Figure 6.4 Areas of Interest of Program 1 and Program 2.


PROGRAM 3
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS
PROGRAM 4
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS

PROGRAM 3

```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef HARD
int attempt = 0;
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef HARD
    player.attempt ++;
    if ( distance >= 80 ) {
    roundPoints += 60;
    } else
    if ( distance >= 60 && distance < 80 ) {
        roundPoints -= 30;
            player.errors ++;
    }
    #endif
    else {
            printf ( " Missed the target " );
            roundPoints -= 50;
    }
    #ifdef HARD
    printf ( " Hit percentage = %f ", player.hits / player.attempt );
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance  );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

PROGRAM 4

```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef STATISTICS
int attempts = 0;
#endif
#ifdef PENALTIES
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    #ifdef STATISTICS
    player.attempts ++;
    #endif
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef PARTIAL_SCORE
    if ( distance >= 80 ) {
        roundPoints += 60;
    } else
    #endif
    #ifdef PENALTIES
    if ( distance >= 60 && distance < 80 ) {
        roundPoints -= 30;
        player.errors ++;
    } else {
    #endif
        printf ( " Missed the target " );
        roundPoints -= 50;
    }
    #ifdef STATISTICS
    printf ( " Hit percentage = %.2f ", player.hits / player.attempts );
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

Figure 6.5 Areas of Interest of Program 3 and Program 4.

programs depended on Latin Square.

The programs with 1 feature constant can have only two scenarios to be analyzed: (i) with the feature enabled and (ii) with the feature disabled. Thus, to ensure the same difficulty level in all sets of tasks, we defined three tasks for all programs as follows:

(i) Task 01: all features enabled, (ii) Task 02: all features enabled and new initial values, (iii) Task 03: all features disabled.

We explained to participants proposed scenarios and initial values of each task before he or she started. The participant could also find these same instructions in the instruction form. For example, the first task about the two programs should be answered considering all features enabled. The task was presented to the participants as follows:

TASK 1: Consider:

FEATURE ENABLED: `PRODUCT_CONTROL`.

FEATURES DISABLED: none.

INITIAL VALUES: `numberOfProducts` = 6 in line 32.

QUESTION: "What will be printed on line 33 when the `int main( )` function on line 30 is executed?"

Tasks of Program 1 (with less variability) and of Program 2 (with more variability) were the same. The difference is only the names of features enabled and disabled. And, according our Latin Square, we allocated half of the participants to execute the tasks based on Program 1, and the other half to execute the tasks based on Program 2.

All artifacts used in our experiment are available at our website[1] and our research share website.[2] In the following, we present the results regarding each metric.

6.7 EXPERIMENTAL RESULTS

6.7.1 Time to provide the correct answer

Table 6.1 Mean time to provide the correct answer (in seconds)

With less variability			With more variability		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
94	72	83	95	93	94

We measured the time (in seconds) each participant took to analyze each program. Our null hypothesis about this metric is:

H_0t: There is no significant difference in the time to provide the correct answer to the tasks when comparing programs with different degrees of variability.

Rows in Table 6.1 show the mean time spent by participants for each program. Shapiro test confirmed that the data about the time to specify output was normally distributed. Table 6.1 shows the mean time spent by all participants for programs with different degrees of variability. They spent a mean time of 83 seconds analyzing programs with less variability and 94 seconds for programs with more variability. Our data revealed that was no significant difference in time for developers to analyze programs with different degrees of variability (p-value = 0.3735). We, thus, cannot reject our null hypothesis (H_0t).

Result 1: Programs with more variability did not require more time for participants to answer the tasks correctly than programs with less variability.

[1]http://www.djansantos.com.br/projects/variability/

[2]https://doi.org/10.5281/zenodo.7982409

Table 6.2 Total number of attempts needed until correct answer

With less variability			With more variability		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
19	19	38	19	20	39

6.7.2 Number of attempts needed until correct answer

We counted the total number of attempts participants needed to specify the output of the programs correctly, he or she scored one for each attempt for each task. Our null hypothesis about this metric is:

H_0a: There is no significant difference in number of attempts needed until correct answer when comparing programs with different degrees of variability.

Table 6.2 shows the total number of attempts needed to specify the output correctly by all participants. They needed 38 attempts for programs with less variability and 39 attempts for programs with more variability. The χ^2 test (Pearson's Chi-squared test) (CAMILLI; HOPKINS, 1978) revealed no significant difference between the number of attempts needed to specify the output correctly in programs with different degrees of variability. The p-values is 1. Based on this, we cannot reject our null hypothesis H_0a.

Result 2: There was no significant difference in the number of attempts needed for participants until giving the correct answer when comparing programs with different degrees of variability.

6.7.3 Visual effort

Total number of fixations

Table 6.3 Number of fixations

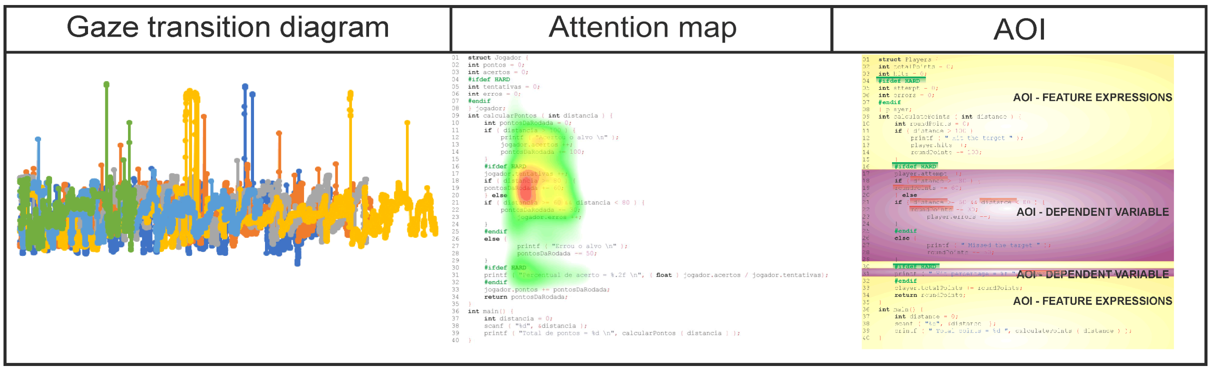
With less variability			With more variability		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
195	151	346	218	219	437

We counted the number of fixations per program. Our null hypothesis about this metric is:

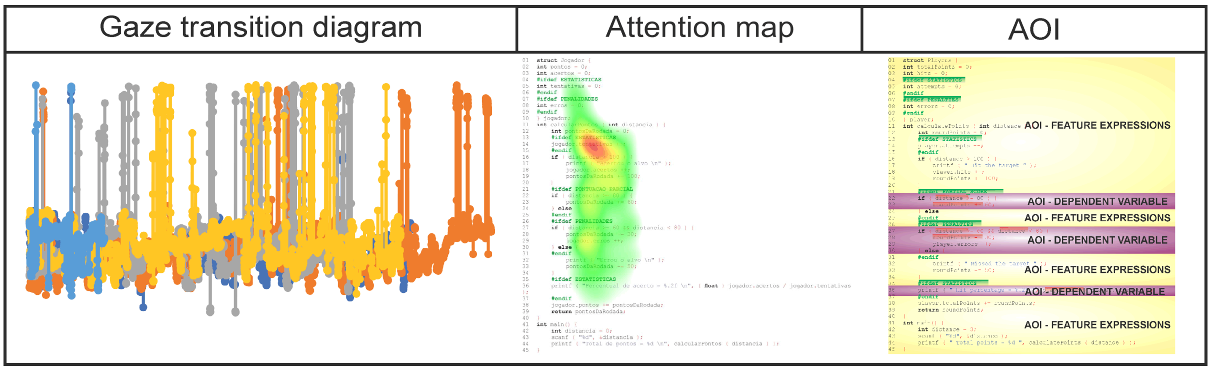
H_0f: There is no significant difference in the number of fixations to specify the output when comparing programs with different degrees of variability.

Table 6.3 shows the total number of fixations the participants executed when analyzing the programs in order to specify their correct output. They executed 171 fixations for programs with less variability and 213 fixations for programs with more variability. Our data revealed that the number of fixations was not significantly different between programs with different degrees of variability (p-value = 0.11485). We cannot reject our null hypothesis (H_0f)

Result 3: Developers did not make more fixations to understand programs with different degrees of variability.



(a) Program 1 with less variability



(b) Program 2 with more variability

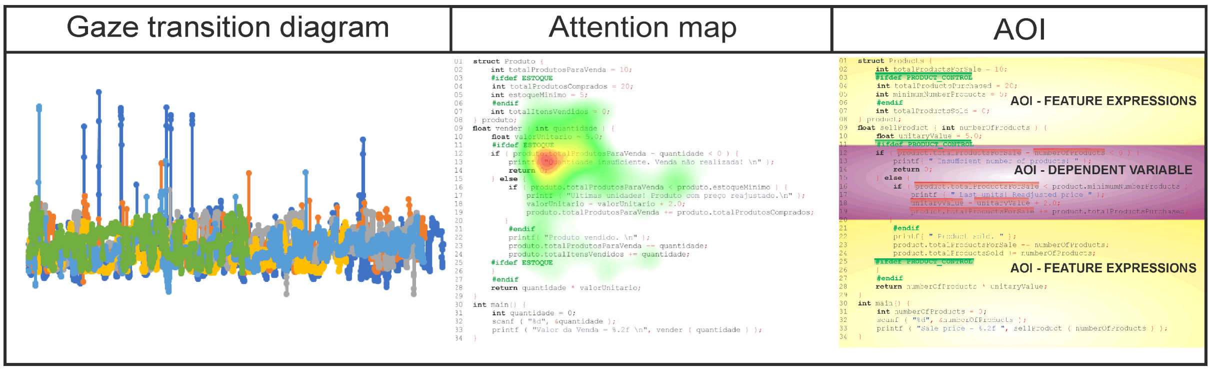
Figure 6.6 Gaze transitions diagram and attention map of programs 3 and 4.

Gaze transitions and attention map

Figure 6.6a and Figure 6.6b show our gaze transitions diagrams and attention maps related to programs 1 and 2 respectively. Figure 6.7a and Figure 6.7b show our gaze transitions diagrams and attention maps related to programs 3 and 4 respectively. We superimposed all individual gaze transitions and attention maps of each participant. Each gaze transition diagram and attention map are, thus, composed by the overlapping of six individual gaze transitions and attention maps.

The gaze transitions diagrams reveal that participants executed more transitions on Program 2 (Figure 6.6b), which has more variability, than Program 1 (Figure 6.6a), which has more variability. Program 1 and Program 2 are programs developed in the same domain 1. It is possible to observe in Figure 6.6b a greater distribution of transitions toward the top of the source code than in Figure 6.6a. We hypothesize that this occurs because programs with more variability force the participant to simulate different configurations of enabled/disabled features.

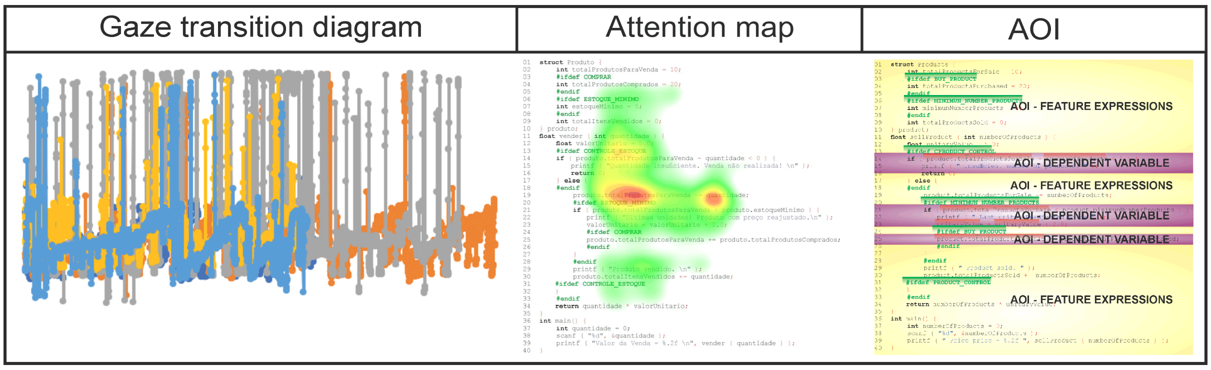
The attention map of Program 1 (Figure 6.6a) reveals that participants' attention was directed to the usages of `roundPoints` variable. Participants concentrated their attention on the dependent variable AOI and not on the feature expression AOI. In this case, participants did not concentrate their attention on #ifdef clauses. We hypothesize that this occurs because `roundPoints` variable concentrated the most of dependencies in this program generating more cognitive effort. The attention map of Program 2 (Figure 6.6b) reveals that participants, when analyzing programs with more variability, needed to navigate over all source code, and, as a consequence, the attentions are more distributed along the AOIs. Programs with more variability force the participant to simulate different configurations of enabled/disabled features.



```
struct Produto {
    int totalProdutosParaVenda = 10;
    #ifdef ESTOQUE
    int totalProdutosComprados = 20;
    int estoqueMinimo = 5;
    #endif
    int totalItensVendidos = 0;
} produto;
float vender ( int quantidade ) {
    float valorUnitario = 5.0;
    #ifdef ESTOQUE
    if ( produto.totalProdutosParaVenda - quantidade < 0 ) {
        printf( "Quantidade insuficiente. Venda não realizada! \n" );
        return 0;
    } else {
        if ( produto.totalProdutosParaVenda < produto.estoqueMinimo ) {
            printf ( "Ultimas unidades! Produto com preço reajustado.\n" );
            valorUnitario = valorUnitario + 2.0;
            produto.totalProdutosParaVenda += produto.totalProdutosComprados;
        }
        #endif
        printf( "Produto vendido. \n" );
        produto.totalProdutosParaVenda -= quantidade;
        produto.totalItensVendidos += quantidade;
    #ifdef ESTOQUE
    }
    #endif
    return quantidade * valorUnitario;
}
int main() {
    int quantidade = 0;
    scanf ( "%d", &quantidade );
    printf ( "Valor da Venda = %.2f \n", vender ( quantidade ) );
}
```

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef PRODUCT_CONTROL
    int totalProductsPurchased = 20;
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef PRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf( " Insufficient number of products! " );
        return 0;
    } else {
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            product.totalProductsForSale += product.totalProductsPurchased;
        }
        #endif
        printf( " Product sold. " );
        product.totalProductsForSale -= numberOfProducts;
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale price - %.2f ", sellProduct ( numberOfProducts ) );
}
```

(a) Program 3 with less variability



```
struct Produto {
    int totalProdutosParaVenda = 10;
    #ifdef COMPRAR
    int totalProdutosComprados = 20;
    #endif
    #ifdef ESTOQUE_MINIMO
    int estoqueMinimo = 5;
    #endif
    int totalItensVendidos = 0;
} produto;
float vender ( int quantidade ) {
    float valorUnitario = 5.0;
    #ifdef CONTROLE_ESTOQUE
    if ( produto.totalProdutosParaVenda - quantidade < 0 ) {
        printf ( "Quantidade insuficiente. Venda não realizada! \n" );
        return 0;
    } else {
    #endif
        produto.totalProdutosParaVenda -= quantidade;
        #ifdef ESTOQUE_MINIMO
        if ( produto.totalProdutosParaVenda < produto.estoqueMinimo ) {
            printf ( "Ultimas unidades! Produto com preço reajustado.\n" );
            valorUnitario = valorUnitario + 2.0;
            #ifdef COMPRAR
            produto.totalProdutosParaVenda += produto.totalProdutosComprados;
            #endif
        }
        #endif
        printf ( "Produto vendido. \n" );
        produto.totalItensVendidos += quantidade;
    #ifdef CONTROLE_ESTOQUE
    }
    #endif
    return quantidade * valorUnitario;
}
int main() {
    int quantidade = 0;
    scanf ( "%d", &quantidade );
    printf ( "Valor da Venda = %.2f \n", vender ( quantidade ) );
}
```

```
struct Products {
    int totalProductsForSale = 10;
    #ifdef BUY_PRODUCT
    int totalProductsPurchased = 20;
    #endif
    #ifdef MINIMUN_NUMBER_PRODUCTS
    int minimunNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef CPRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf ( " Insufficient number of products! " );
        return 0;
    } else {
    #endif
        product.totalProductsForSale -= numberOfProducts;
        #ifdef MINIMUN_NUMBER_PRODUCTS
        if ( product.totalProductsForSale < product.minimunNumberProducts ) {
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            #ifdef BUY_PRODUCT
            product.totalProductsForSale += product.totalProductsPurchased;
            #endif
        }
        #endif
        printf ( " Product sold. " );
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( " Price price - %.2f ", sellProduct ( numberOfProducts ) );
}
```

(b) Program 4 with more variability

Figure 6.7 Gaze transitions diagram and attention map of programs 3 and 4.

For programs in domain 2, the gaze transitions diagram reveals that participants executed more transitions on Program 4 (Figure 6.7b), which has more variability, than Program 3 (Figure 6.7a) which has less variability. This scenario is the same as domain 1. We observed in Figure 6.7b a greater distribution of transitions toward the top of the source code than transitions in Figure 6.7a. Participants, when analyzing programs with more variability, navigated between feature expression regions, and, as a consequence, the gaze transitions are more distributed along the source code. In these cases, participants needed, thus, to realize long transitions to feature expression regions and variable definitions regions.

The attention map of Program 3 (Figure 6.7a) reveals that participants' attention was directed to the usages of `totalProductsForSale` variable. The attention maps show one red regions in the dependent variable AOI. We hypothesize that this occurs because `totalProductsForSale` variable concentrated most of the dependencies of this program generating more attention in this area. The attention map of Program 4 (Figure 5.7b), also shows that participants' attention was directed to the usages of `totalProductsForSale` variable. However, the attentions are more distributed along the AOIs. The attention maps show a more widespread distribution of attention over all source codes. This leads to the following result.

Result 4: Programs with more variability forced developers to direct their attention to more regions, causing a more widespread distribution of attention and transitions over distinct parts of source code.

6.7.4 Heart-related biometrics

Heart Rate Variability (HRV) and Stress Level

Table 6.4 Total number of HRV and Stress Level variations

	With less variability			With more variability		
Programs	**Program 1**	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
HRV	1	4	5	3	5	8
total of participants	1	3	4	3	3	6

We counted the number of HRV for programs with different degrees of variability. Our null hypothesis about this metric is:

H_0h: There is no significant difference in the number of HRV to specify the output of the programs when comparing programs with different degrees of variability.

Table 6.4 shows the total number of HRV of participants during the tasks. Four participants had 5 HRV for programs with less variability and six participants had 8 HRV for programs with more variability. The χ^2 test revealed no significant difference between the number of HRV needed to specify the output correctly in programs with different degrees of variability. The value χ^2 is 0.375, and the p-values are 0.54. Based on this, we cannot reject our null hypothesis H_0h.

Result 5: There was no significant difference in the number of HRV until giving the correct answer when comparing programs with different degrees of variability.

6.7.5 Discussion

Here we answer our research question *How do different degrees of variability affect the comprehensibility of configurable system?* by discussing different aspects of our findings.

Degrees of variability did not affect comprehensibility.

Our results 1, 2, 3, and 5 show that programs with more variability did not demand more comprehension effort from participants. Developers did not spend more time or make more fixations when compared with programs with less variability. It is important to note that, in this study, we fixed the number of dependent variables, i.e. all programs had the same number of dependent variables and usages of them. Therefore, the result is somehow aligned with the results of Study 3 (Chapter 5) in which comprehensibility was hinder when we increased the number of dependent variables. This result contradicts a previous study that says that more variability increases debugging time (MELO et al., 2017). It is important to say that, in their study, Melo *et al.* did not fix the number of dependent variables, their tasks was debugging, and the programs was not implemented with #ifdef.

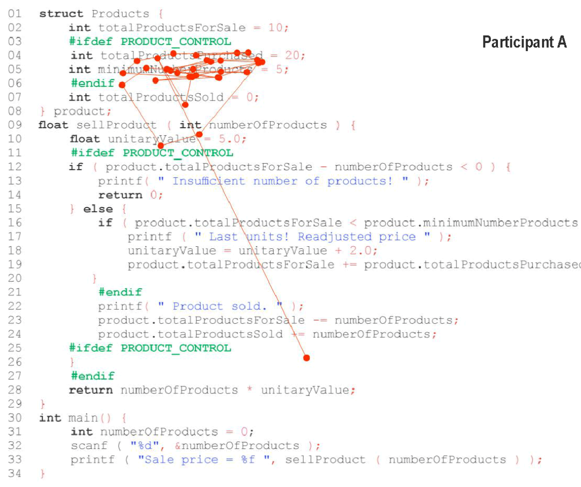
Programs with more variability required more visual effort.

Results 4 show that programs with more variability may increase visual effort. To specify the output, participants needed to perform more gaze transitions and focused more time on distinct parts of the source code.

There was no significant difference in the number of heart variation points when comparing programs with different degrees of variability.

Result 5 shows that there was no significant difference in terms of HRV while participants analyzed programs with different degrees of variability. However, we decided to investigate what was happening in terms of gaze movements during HRV.

Scan paths of HRV



```
struct Products {
    int totalProductsForSale = 10;
    #ifdef PRODUCT_CONTROL
    int totalProductsPurchased = 20;
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef PRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf( " Insufficient number of products! " );
        return 0;
    } else {
        if ( product.totalProductsForSale < product.minimumNumberProducts ) {
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            product.totalProductsForSale += product.totalProductsPurchased;
        }
        #endif
        printf( " Product sold. " );
        product.totalProductsForSale -= numberOfProducts;
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( "Sale price = %f ", sellProduct ( numberOfProducts ) );
}
```

Figure 6.8 Individual scan path of participants from program 1 (less variability).

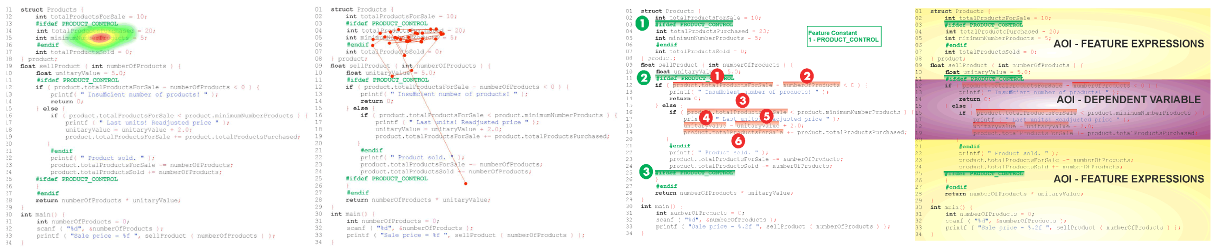


Figure 6.9 Gaze movements diagrams of Program 1 (less variability).

We generated scan paths of all participants' HRV to identify what happened when they had a variance in Heart Rate and Stress Level.

Figure 6.8 shows a scan path of the only participant that had HRV while analyzing Program 1. We generated a scan path at the moment the HRV happened. This heart variation happened while he/she answered the third task, which had the program with all features disabled. In Figure 6.9 we have the scan path image and the attention map during the HRV, as well as the AOI map, and the source code. Figure 6.9 shows that the gaze movements were concentrated in the region of `CONTROL_PRODUCT` feature. We did not expect the participant's gaze movements to focus on a disabled feature. We hypothesized that HRV happened when the participant perceived that he or she was analyzing a piece of code that contained a disabled feature. Analyzing programs with disabled features is not a trivial task. The participant concentrated their gaze movements, during the HRV, in a disabled feature.

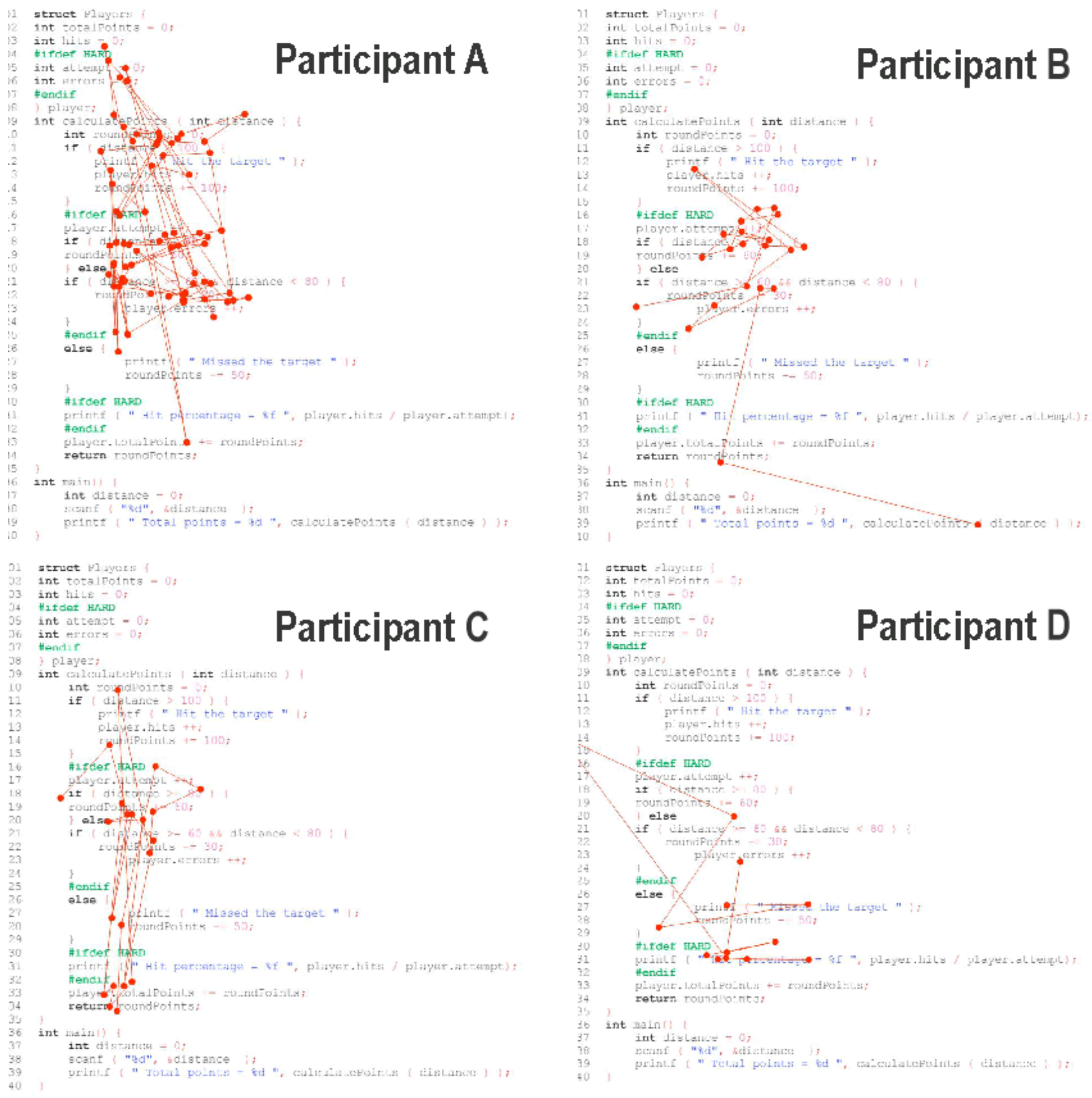
Figure 6.10a represents scan paths of participants that analyzed Program 3, another program with less variability. Four participants registered HRV. Similarly to what we found in Study 3 (Section 5.8), most of the moments that participants had an HRV they were executing long gaze transitions. In this case, Figure 6.10a shows participants B, C, and D executing long gaze transitions. Two of the four participants had HRV while they were answering the third task, which has all features disabled. Figure 6.10b is composed of the overlapping of the four individual scan paths and attention maps during the heart variation of the four participants. The red regions indicate that participants' attention was directed to the usages of `roundPoints` variable. We hypothesize that this occurs because `roundPoints` variable concentrated most of the dependencies of this program generating more cognitive effort.

Figure 6.11a shows scan paths of participants that analyzed Program 2, which has more variability. Participants performed long gaze transitions with fixations distributed over distinct parts of the source code. In Figure 6.11b we had the overlapping of the four individual scan paths and attention maps that occured during HRV. Figure 6.11b confirms the gaze movements distributed in the feature expressions region and dependent variables region. The same happens in Program 4 also with more variability. Participants performed long gaze transitions causing a widespread distribution of attention over distinct parts of source code as shown in Figures 6.12a and 6.12b. Again, most of participants were answering the third task, which has all features disabled while having an HRV.

In summary, participants seem to have performed long gaze transitions during HRV and have made fixations distributed over distinct parts of the source code. Also, analyzing programs with disabled features is not a trivial task. Some participants were analyzing programs with all features disabled when they had HRV occurrence. However, it is important to highlight again that these findings were based on limited observations and information.

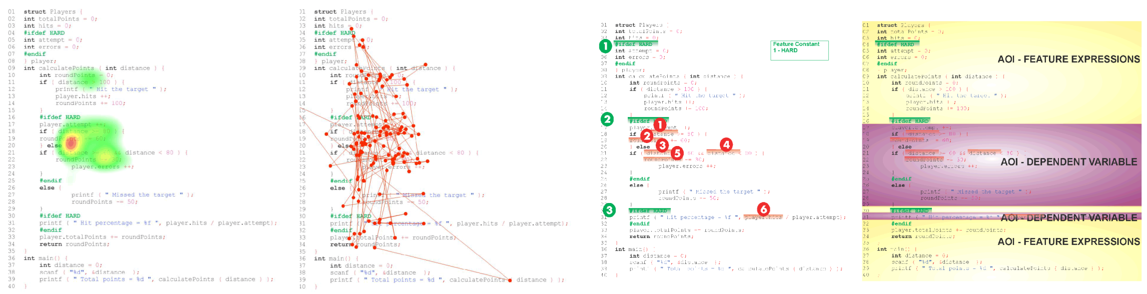
Programs with more variability did not affect number of attempts to specify program outputs.

Result 2 revealed that the number of attempts needed for participants until giving the correct answer was not affected by different degrees of variability. This result con-



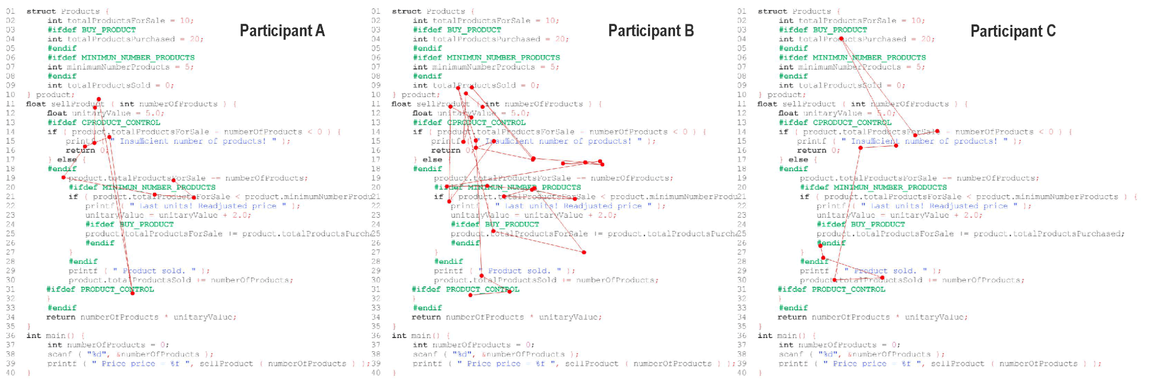
```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef HARD
int attempt = 0;
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef HARD
    player.attempt ++;
    if ( distance > 80 ) {
        roundPoints += 60;
    } else
    if ( distance >= 60 && distance < 80 ) {
        roundPoints += 30;
        player.errors ++;
    }
    #endif
    else {
        printf ( " Missed the target " );
        roundPoints -= 50;
    }
    #ifdef HARD
    printf ( " Hit percentage = %f ", player.hits / player.attempt );
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

(a) Individual scan path of participants from program 3 (less variability).


Feature Constant 1 - HARD
1
2
3
4
5
6
AOI - FEATURE EXPRESSIONS
AOI - DEPENDENT VARIABLE
AOI - DEPENDENT VARIABLE
AOI - FEATURE EXPRESSIONS

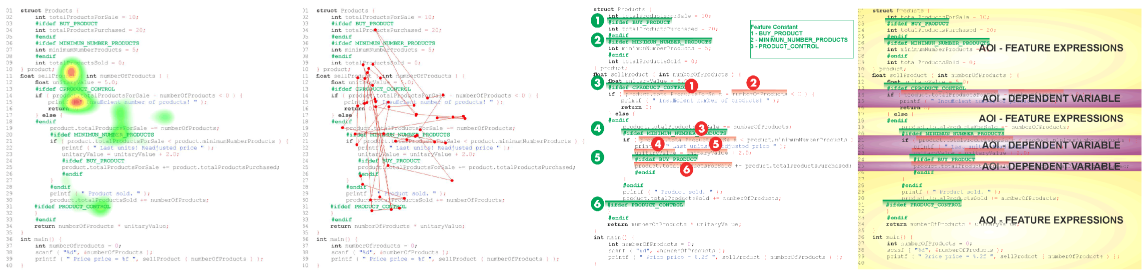
(b) Gaze movements diagram of Program 3 (less variability).

Figure 6.10 Gaze movements of Program 3 in the moment of HRV



```
struct Products {
    int totalProductsForSale = 10;
    #ifdef BUY_PRODUCT
    int totalProductsPurchased = 20;
    #endif
    #ifdef MINIMUN_NUMBER_PRODUCTS
    int minimumNumberProducts = 5;
    #endif
    int totalProductsSold = 0;
} product;
float sellProduct ( int numberOfProducts ) {
    float unitaryValue = 5.0;
    #ifdef CPRODUCT_CONTROL
    if ( product.totalProductsForSale - numberOfProducts < 0 ) {
        printf ( " Insufficient number of products! " );
        return 0;
    } else {
    #endif
        product.totalProductsForSale -= numberOfProducts;
        #ifdef MINIMUN_NUMBER_PRODUCTS
        if ( product.totalProductsForSale < product.minimumNumberProdu
            printf ( " Last units! Readjusted price " );
            unitaryValue = unitaryValue + 2.0;
            #ifdef BUY_PRODUCT
            product.totalProductsForSale += product.totalProductsPurch
            #endif
        }
        #endif
        printf ( " Product sold. " );
        product.totalProductsSold += numberOfProducts;
    #ifdef PRODUCT_CONTROL
    }
    #endif
    return numberOfProducts * unitaryValue;
}
int main() {
    int numberOfProducts = 0;
    scanf ( "%d", &numberOfProducts );
    printf ( " Price price = %f ", sellProduct ( numberOfProducts ) );
}
```

(a) Individual scan path of participants from Program 2 (more variability).



(b) Gaze movements diagram of Program 2 (more variability).

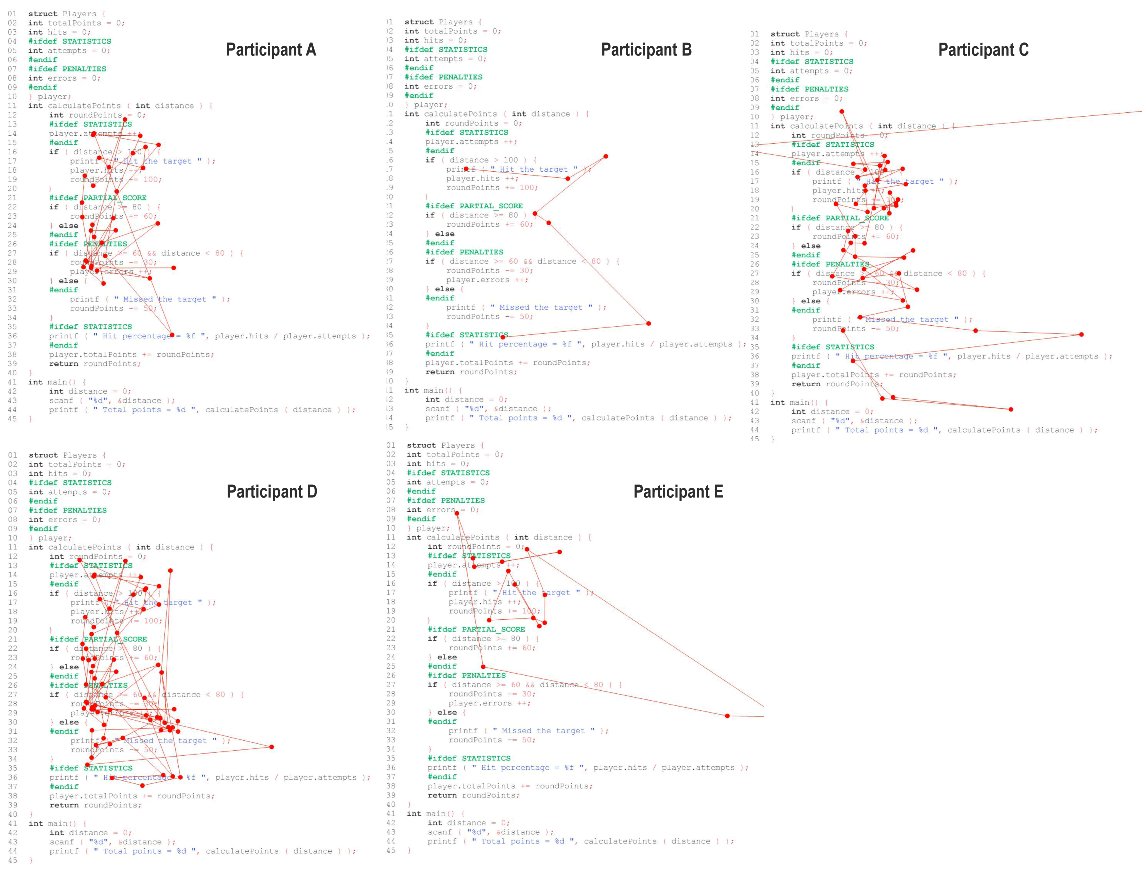
Figure 6.11 Gaze movements of Program 2 in the moment of HRV

firms previous studies that showed that most participants answer the tasks correctly in programs with #ifdef (MELO; BRABRAND; WASOWSKI, 2016; MELO et al., 2017; SANTOS; SANT'ANNA, 2019).

6.8 THREATS TO VALIDITY

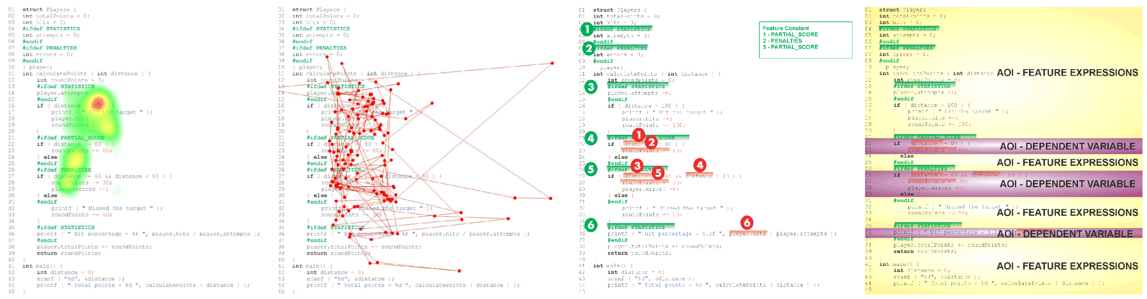
The procedures of this study were the same of Study 3, described in section 5.6. Furthermore, the programs and tasks also were similar to the ones of Study 3. Thus, the threats to validity of this study are the same of Study 3, already described in Section 5.9, with the exception of the external validity described below.

Programs. Program 1 has an undisciplined #ifdef. We became aware of this inclusion only after completing the studies. Although the literature suggests that such undisciplined code may hinder comprehensibility, our findings indicate that the presence of this undisciplined #ifdef did not have a significant impact on comprehension. The metrics presented in our results demonstrate no significant difference in comprehension between Program 1 and Program 2.



```
struct Players {
int totalPoints = 0;
int hits = 0;
#ifdef STATISTICS
int attempts = 0;
#endif
#ifdef PENALTIES
int errors = 0;
#endif
} player;
int calculatePoints ( int distance ) {
    int roundPoints = 0;
    #ifdef STATISTICS
    player.attempts ++;
    #endif
    if ( distance > 100 ) {
        printf ( " Hit the target " );
        player.hits ++;
        roundPoints += 100;
    }
    #ifdef PARTIAL_SCORE
    if ( distance >= 80 ) {
        roundPoints += 60;
    } else
    #endif
    #ifdef PENALTIES
    if ( distance >= 60 && distance < 80 ) {
        roundPoints -= 30;
        player.errors ++;
    } else {
    #endif
        printf ( " Missed the target " );
        roundPoints -= 50;
    }
    #ifdef STATISTICS
    printf ( " Hit percentage = %f ", player.hits / player.attempts );
    #endif
    player.totalPoints += roundPoints;
    return roundPoints;
}
int main() {
    int distance = 0;
    scanf ( "%d", &distance );
    printf ( " Total points = %d ", calculatePoints ( distance ) );
}
```

(a) Individual scan path of participants from Program 4 (more variability).



(b) Gaze movements diagram of Program 4 (more variability).

Figure 6.12 Gaze movements of Program 4 in the moment of HRV

Chapter

7

CONCLUSION AND FUTURE WORK

The #ifdef is commonly used in configurable systems because it provides useful and necessary variability. However, variability implemented with #ifdef is criticized because it obfuscate source code, making it harder to understand. Often pieces of source code within with #ifdefs share program elements like variables causing feature dependency. In this work, we describe four empirical studies that we conducted to investigate how feature dependency affect the comprehensibility of configurable system source code.

First, in Study 1, we conduct an online experiment with developers to investigate how feature dependencies affect the comprehensibility of configurable systems implemented with #ifdefs. Participants had to analyze programs implemented with #ifdefs and specify their outputs. We compared programs with and without feature dependency containing different types of feature dependency in the same source code. We measured comprehensibility by the time and number of attempts that each developer took to answer the tasks. Our results showed that feature dependencies affected the comprehensibility of configurable system. Developers spent more time to analyze source code containing feature dependency. We hypothesize that this may have happened thanks to how feature dependency obligates developers to worry more about dependent variables and makes it harder to simulate different configurations of enabled/disabled features.

Second, in Study 2, we executed a controlled experiment with human subjects to investigate how different types of feature dependency affect the comprehensibility of configurable systems implemented with #ifdefs. We asked the participants to try to find different types of bugs in programs with different types of feature dependency. Then, we measured their performance in terms of spent time, number of found bugs and visual effort. Our results show that different types of feature dependency affect comprehensibility in different degrees. We observed that: (i) global and interprocedural dependencies demanded more time, (ii) interprocedural dependencies required more visual effort and (iii) #ifdefs did not impact the comprehensibility of programs with intraprocedural dependencies. These results lead us to hypothesize that comprehensibility is more negatively affected when a variable which is shared between features is defined in a point far from the points where it is used.

Third, in Study 3, we carried out a controlled experiment with developers to analyze how the number of dependent variables and the number of uses of such variables influences the comprehensibility of source code of configurable systems. Participants had to analyze programs and specify output of programs with different number of dependent variables and variable usages. Then, we measured the time and the number of attempts that each developer took to specify the output. Furthermore, we recorded gaze movements and heart rate variations of each participant using a eye-tracking device and a smartwatch. Our results showed that developers spent more time to analyze source code containing more dependent variables. Also, they made more fixations to understand source code with more dependent variables. We hypothesize that this happened because, when a developer analyze a variable, he or she need to direct their attention to the local where it is defined. Then, if the source code has more dependent variables developers need to direct their attention more times. However, there was no significant difference in the number of participants' heart rate variations until giving the correct answer when comparing programs with different numbers of dependent variables.

Finally, in Study 4, we carried out a controlled experiment with 12 participants to analyze the impact of degrees of variability on the comprehensibility of source code with #ifdefs. Again, we measured time and number of attempts for participants to specify the output of programs implemented with #ifdefs. We defined programs with two different degrees of variability and the same number of feature dependencies. We also measured visual effort and heart rate variability using an eye-tracking device and a smartwatch. The results indicate that the time and number of attempts are relatively independent of the degree of variability when programs had the same number of feature dependencies. Results indicate that the visual effort appears to increase with the degree of variability. We hypothesize that this happened because analyze programs with more dependent variables increases attention and effort. It is important to note that, in this study, we fixed the number of dependent variables, i.e. all programs had the same number of dependent variables and usages of them. This result contradicts a previous study carried out by Melo et al. (MELO et al., 2017). Their study showed that more variability increases debugging time. However, it is important to say that, in their study, they did not fix the number of dependent variables, their tasks were debugging, and the programs were not implemented with #ifdef.

In summary, we concluded that feature dependency may affect the comprehensibility of configurable system source code, in different ways: (i) when #ifdef contain dependent variables, (ii) when #ifdef contain intraprocedural and interprocedural dependencies, and (iii) when the source code contains many dependent variables. We hypothesize that this happened because if the source code has dependent variables developers need to direct more attention to them. Also, comprehensibility is more negatively affected when a dependent variable is defined at a point far from the points where it is used.

The insights obtained with our studies can, in the future, support developers of configurable systems to know the parts of the source code they should take more care about. These parts would be the ones with dependent variables that cause a certain type of feature dependency. However, new issues should be investigated as future work:

- **Programming language.** We wrote our programs in C, using #ifdefs. Future works might investigate dependent variables in other programming languages and techniques, like using the feature-oriented programming or conventions of Cide (KäSTNER; APEL; KUHLEMANN, 2008). Also, other comprehension code scenarios and tasks should also be explored.

- **Real systems.** Due to limitations we used small programs. We plan to accomplish another experiment in order to evaluate comprehensibility of dependent variables using real systems in a more realistic environment, with IDEs and source code with multiple files.

- **Feature dependency.** For the research community, we encourage more empirical studies investigating the impact of feature dependencies caused by functions, data flow, control-flow, among others.

- **Biometrics.** We used the Tobii Eyex device, Tobii tracker 5 device, and the Garmin Fenix 5s smartwatch. Both are low-cost equipment. We suggest other eye-tracking and heart rate devices should also be used. Also, other biometrics may be explored. We plan to include Electrodermal activity (EDA) in future works. EDA refers to the variation of the electrical conductance of the skin in response to sweat secretion.

- **Tool support.** Tool support has been out of the scope of this research. However, developing a new tool or extending an existing one for code comprehension support will be an essential contribution to this area.

- **More empirical studies.** This Thesis presented the design, procedures, programs, tasks, analyzes of a set of empirical studies. We suggest carrying out new studies in different contexts, including various participants and other domains.

- **Degrees of granularity.** We do not discuss degrees of granularity, such as statement and expression extensions or function signature changes. Although the #ifdef allows a programmer to annotate code even at the finest level of granularity, not much is known about the necessity to make such fine-grained extensions. A high number of fine-grained extensions can incur in #ifdef undisciplined. We suggest carrying out new studies including degrees of granularity.

BIBLIOGRAPHY

ABAL, I.; BRABRAND, C.; WASOWSKI, A. 42 variability bugs in the linux kernel: a qualitative analysis. In: ACM. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. New York, NY, USA: Association for Computing Machinery, 2014. p. 421–432. Disponível em: <10.1145/2642937.2642990>.

ABAL, I. et al. Variability bugs in highly configurable systems: a qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 26, n. 3, p. 10, 2018. Disponível em: <10.1145/3149119>.

APEL, S. et al. Feature-oriented software product lines: Concepts and implementation, berlin/heidelberg, 2013, 308 pages. *URL http://www. springer. com/computer/swe/book/978-3-642-37520-0*, 2013.

APEL, S.; BEYER, D. Feature cohesion in software product lines: an exploratory study. In: IEEE. *2011 33rd International Conference on Software Engineering (ICSE)*. 2011. p. 421–430. Disponível em: <https://doi.org/10.1145/1985793.1985851>.

APEL, S. et al. Exploring feature interactions in the wild: the new feature-interaction challenge. In: ACM. *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. 2013. p. 1–8. Disponível em: <10.1145/2528265.2528267>.

BAILEY, R. A. *Design of comparative experiments*. [S.l.]: Cambridge University Press, 2008.

BAKSHY, E.; ECKLES, D.; BERNSTEIN, M. S. Designing and deploying online field experiments. In: *Proceedings of the 23rd international conference on World wide web*. [S.l.: s.n.], 2014. p. 283–292.

BANIASSAD, E.; MURPHY, G. Conceptual module querying for software reengineering. In: *Proceedings of the 20th International Conference on Software Engineering*. [S.l.: s.n.], 1998. p. 64–73.

BERGER, T. et al. Three cases of feature-based variability modeling in industry. In: SPRINGER. *International Conference on Model Driven Engineering Languages and Systems*. 2014. p. 302–319. Disponível em: <10.1007/978-3-319-11653-2_19>.

BERGER, T. et al. A survey of variability modeling in industrial practice. In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2013. (VaMoS '13). ISBN 9781450315418. Disponível em: <https://doi.org/10.1145/2430502.2430513>.

BERGER, T. et al. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 12, p. 1611–1640, 2013.

BOX, G. E.; HUNTER, J. S.; HUNTER, W. G. *Statistics for experimenters: design, innovation, and discovery*. [S.l.]: Wiley-Interscience New York, 2005.

BRAZ, L. et al. A change-centric approach to compile configurable systems with #ifdefs. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. New York, NY, USA: Association for Computing Machinery, 2016. (GPCE 2016), p. 109119. ISBN 9781450344463. Disponível em: <https://doi.org/10.1145/2993236.2993250>.

CAFEO, B. B. et al. Feature dependencies as change propagators: an exploratory study of software product lines. *Information and Software Technology*, Elsevier, v. 69, p. 37–49, 2016. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584915001512>.

CAFEO, B. B. et al. Analysing the impact of feature dependency implementation on product line stability: An exploratory study. In: IEEE. *2012 26th Brazilian Symposium on Software Engineering*. [S.l.], 2012. p. 141–150.

CAMILLI, G.; HOPKINS, K. D. Applicability of chi-square to 2× 2 contingency tables with small expected cell frequencies. *Psychological Bulletin*, American Psychological Association, v. 85, n. 1, p. 163, 1978. Disponível em: <10.1037/0033-2909.85.1.163>.

CATALDO, M. et al. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, IEEE, v. 35, n. 6, p. 864–878, 2009.

CAVALCANTE, E. et al. Exploiting software product lines to develop cloud computing applications. In: ACM. *Proceedings of the 16th International Software Product Line Conference-Volume 2*. [S.l.], 2012. p. 179–187.

CLEMENTS, P.; NORTHROP, L. *Software product lines: practices and patterns*. [S.l.]: Addison-Wesley Reading, 2002.

CLEMENTS, P.; NORTHROP, L. Software product lines. *Product Line systems Program*, 2003.

COSTA, J. A. S. da et al. Evaluating refactorings for disciplining# ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, Springer, v. 26, n. 5, p. 1–35, 2021. Disponível em: <10.1007/s10664-021-10002-8>.

COSTA, J. A. S. da et al. Evaluating refactorings for disciplining# ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, Springer, v. 26, n. 5, p. 92, 2021.

COUCEIRO, R. et al. Biofeedback augmented software engineering: Monitoring of programmers' mental effort. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. [S.l.: s.n.], 2019. p. 37–40.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, Wiley Online Library, v. 10, n. 1, p. 7–29, 2005.

DUCHOWSKI, A. T. *Eye tracking methodology: Theory and practice*. Springer, 2017. Disponível em: <10.1007/978-3-319-57883-5>.

ERNST, M. D.; BADROS, G. J.; NOTKIN, D. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, IEEE, v. 28, n. 12, p. 1146–1170, 2002.

FENSKE, W.; SCHULZE, S.; SAAKE, G. How preprocessor annotations (do not) affect maintainability: a case study on change-proneness. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 52, n. 12, p. 77–90, 2017.

FENSKE, W.; SCHULZE, S.; SAAKE, G. How preprocessor annotations (do not) affect maintainability: a case study on change-proneness. In: ACM. *ACM SIGPLAN Notices*. New York, NY, USA: Association for Computing Machinery, 2017. p. 77–90. Disponível em: <10.1145/3170492.3136059>.

FOROUZAN, B. A.; GILBERG, R. F. *Computer Science: A structured programming approach using C*. [S.l.]: Brooks/Cole Publishing Company, 2000.

FRITZ, T. et al. 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: Association for Computing Machinery, 2014. (ICSE 2014), p. 402413. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568266>.

GARMIN. *fenix 5s Owners Manual*. [S.l.], 2017.

GARVIN, B. J.; COHEN, M. B. Feature interaction faults revisited: An exploratory study. In: IEEE. *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. [S.l.], 2011. p. 90–99.

GOLDBERG, J. H.; KOTVAL, X. P. Eye movement-based evaluation of the computer interface. *Advances in occupational ergonomics and safety*, IOS PRESS, p. 529–532, 1998.

HIJAZI, H. et al. Intelligent biofeedback augmented content comprehension (tellback). *IEEE Access*, v. 9, p. 28393–28406, 2021.

HOFMEISTER, J. et al. Comparing novice and expert eye movements during program comprehension. *FACHBEREICH MATHEMATIK UND INFORMATIK SERIE B INFORMATIK*, v. 17, 2017.

HRISTOVA, M. et al. Identifying and correcting java programming errors for introductory computer science students. In: ACM. *ACM SIGCSE Bulletin*. 2003. p. 153–156. Disponível em: <10.1145/792548.611956>.

IEEE. The institute of electrical and eletronics engineers. ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1–84, Dec 1990.

JACOB, R. J.; KARN, K. S. Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. In: *The mind's eye*. Elsevier, 2003. p. 573–605. Disponível em: <10.1016/B978-044451020-4/50031-1>.

JR, C. C.; STAUB, A.; RAYNER, K. Eye movements in reading words and sentences. *Eye movements*, Elsevier, p. 341–371, 2007.

KÄSTNER, C.; APEL, S.; KUHLEMANN, M. Granularity in software product lines. In: ACM. *Proceedings of the 30th international conference on Software engineering*. 2008. p. 311–320. Disponível em: <https://doi.org/10.1145/1368088.1368131>.

KäSTNER, C.; APEL, S.; KUHLEMANN, M. Granularity in software product lines. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2008. (ICSE '08), p. 311320. ISBN 9781605580791. Disponível em: <https://doi.org/10.1145/1368088.1368131>.

KEVIC, K. et al. Tracing software developers' eyes and interactions for change tasks. In: ACM. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015. p. 202–213. Disponível em: <10.1145/2786805.2786864>.

KOHAVI, R. et al. Controlled experiments on the web: survey and practical guide. *Data mining and knowledge discovery*, Springer, v. 18, p. 140–181, 2009.

KOLERS, P. A.; DUCHNICKY, R. L.; FERGUSON, D. C. Eye movement measurement of readability of crt displays. *Human Factors*, SAGE Publications Sage CA: Los Angeles, CA, v. 23, n. 5, p. 517–527, 1981. Disponível em: <10.1177/001872088102300502>.

KRAJBICH, I.; ARMEL, C.; RANGEL, A. Visual fixations and the computation and comparison of value in simple choice. *Nature neuroscience*, Nature Publishing Group US New York, v. 13, n. 10, p. 1292–1298, 2010.

LANZA, M.; MARINESCU, R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. [S.l.]: Springer Science & Business Media, 2007.

LE, D.; WALKINGSHAW, E.; ERWIG, M. # ifdef confirmed harmful: Promoting understandable software variation. In: IEEE. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.], 2011. p. 143–150.

LIEBIG, J. et al. An analysis of the variability in forty preprocessor-based software product lines. In: ACM. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010. p. 105–114. Disponível em: <10.1145/1806799.1806819>.

MAALEJ, W. et al. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 23, n. 4, p. 1–37, 2014. Disponível em: <https://doi.org/10.1145/2622669>.

MALAQUIAS, R. et al. The discipline of preprocessor-based annotations does #ifdef TAG n't #endif matter. In: *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22- 23, 2017*. [S.l.: s.n.], 2017. p. 297–307.

MAYRHAUSER, A. V.; VANS, A. M.; HOWE, A. E. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, Wiley Online Library, v. 9, n. 5, p. 299–327, 1997.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, p. 308–320, 1976.

MCCONNELL, S. *Code complete*. [S.l.]: Pearson Education, 2004.

MEDEIROS, F. et al. The love/hate relationship with the c preprocessor: An interview study. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *LIPIcs-Leibniz International Proceedings in Informatics*. 2015. v. 37, p. 495–518. Disponível em: <10.4230/LIPIcs.ECOOP.2015.495>.

MEDEIROS, F.; RIBEIRO, M.; GHEYI, R. Investigating preprocessor-based syntax errors. In: *Proceedings of the Generative Programming: Concepts and Experiences*. [S.l.: s.n.], 2013. (GPCE '13), p. 75–84.

MEDEIROS, F.; RIBEIRO, M.; GHEYI, R. Investigating preprocessor-based syntax errors. In: ACM. *ACM SIGPLAN Notices*. 2013. p. 75–84. Disponível em: <10.1145/2517208.2517221>.

MEDEIROS, F. et al. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 5, p. 453–469, 2017.

MEDEIROS, F. et al. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 5, p. 453–469, 2017.

MEDEIROS, F. et al. An empirical study on configuration-related code weaknesses. In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2020. p. 193–202.

MEDEIROS, F. et al. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015*. [S.l.: s.n.], 2015. p. 35–44.

MELO, J.; BRABRAND, C.; WASOWSKI, A. How does the degree of variability affect bug finding? In: ACM. *Proceedings of the 38th International Conference on Software Engineering*. 2016. p. 679–690. Disponível em: <10.1145/2884781.2884831>.

MELO, J. et al. Variability through the eyes of the programmer. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2017. p. 34–44.

MÜLLER, S. C.; FRITZ, T. Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 688–699.

MüLLER, S. C.; FRITZ, T. Using (bio)metrics to predict code quality online. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (ICSE '16), p. 452463. ISBN 9781450339001. Disponível em: <https://doi.org/10.1145/2884781.2884803>.

MUNIZ, R. et al. A qualitative analysis of variability weaknesses in configurable systems with #ifdefs. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2018. (VAMOS '18), p. 5158. ISBN 9781450353984. Disponível em: <https://doi.org/10.1145/3168365.3168382>.

NAKAGAWA, T. et al. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE Companion 2014), p. 448451. ISBN 9781450327688. Disponível em: <https://doi.org/10.1145/2591062.2591098>.

OLIVEIRA, R.; CAFEO, B.; HORA, A. On the evolution of feature dependencies: An exploratory study of preprocessor-based systems. In: *VaMoS*. [s.n.], 2019. p. 14–1. Disponível em: <https://doi.org/10.1145/3302333.3302342>.

PFLEEGER, S. L.; KITCHENHAM, B. A. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 26, n. 6, p. 16–18, 2001.

POHL, K.; BÖCKLE, G.; LINDEN, F. J. van D. *Software product line engineering: foundations, principles and techniques*. [S.l.]: Springer Science & Business Media, 2005.

QUEIROZ, F. et al. Towards a better understanding of feature dependencies in preprocessor-based systems. In: *Proceedings of the 6th Latin American Workshop on Aspect-Oriented Software Development: Advanced Modularization Techniques (LA-WASP)*. [S.l.: s.n.], 2012.

RAYNER, K. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, American Psychological Association, v. 124, n. 3, p. 372, 1998. Disponível em: <10.1037/0033-2909.124.3.372>.

RAYNER, K. Eye movements and attention in reading, scene perception, and visual search. *The quarterly journal of experimental psychology*, Taylor & Francis, v. 62, n. 8, p. 1457–1506, 2009.

RAYNER, K. et al. Eye movements as reflections of comprehension processes in reading. *Scientific studies of reading*, Taylor & Francis, v. 10, n. 3, p. 241–255, 2006. Disponível em: <10.1207/s1532799xssr1003_3>.

RIBEIRO, M.; BORBA, P.; KÄSTNER, C. Feature maintenance with emergent interfaces. In: ACM. *Proceedings of the 36th International Conference on Software Engineering*. 2014. p. 989–1000. Disponível em: <10.1145/2568225.2568289>.

RIBEIRO, M. et al. Emergent feature modularization. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. [S.l.: s.n.], 2010. p. 11–18.

RIBEIRO, M. et al. On the impact of feature dependencies when maintaining preprocessor-based software product lines. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 47, n. 3, p. 23–32, 2012. Disponível em: <10.1145/2189751.2047868>.

RODRIGUES, I. et al. Assessing fine-grained feature dependencies. *Information and Software Technology*, Elsevier, v. 78, p. 27–52, 2016. Disponível em: <https://doi.org/10.1016/j.infsof.2016.05.006>.

SACKMAN, H.; ERIKSON, W. J.; GRANT, E. E. *Exploratory experimental studies comparing online and offline programing performance*. [S.l.], 1966.

SAMMET, J. Software psychology: human factors in computer and information systems. *ACM SIGCHI Bulletin*, ACM, v. 14, n. 4, p. 19–20, 1983. Disponível em: <https://doi.org/10.1145/1044188.1044193>.

SANTOS, A. R. et al. Comparing the influence of using feature-oriented programming and conditional compilation on comprehending feature-oriented software. *Empirical Software Engineering*, Springer, v. 24, p. 1226–1258, 2019.

SANTOS, D.; SANT'ANNA, C. How does feature dependency affect configurable system comprehensibility? In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2019. p. 19–29.

SCHMID, K.; RUMMLER, A. Cloud-based software product lines. In: CITESEER. *SPLC (2)*. [S.l.], 2012. p. 164–170.

SCHULZE, S. et al. Does the discipline of preprocessor annotations matter?: a controlled experiment. In: ACM. *ACM SIGPLAN Notices*. 2013. p. 65–74. Disponível em: <10.1145/2517208.2517215>.

SHAFT, T. M.; VESSEY, I. The relevance of application domain knowledge: The case of computer program comprehension. *Information systems research*, INFORMS, v. 6, n. 3, p. 286–299, 1995.

SHARIF, B.; FALCONE, M.; MALETIC, J. I. An eye-tracking study on the role of scan time in finding source code defects. In: ACM. *Proceedings of the Symposium on Eye Tracking Research and Applications*. 2012. p. 381–384. Disponível em: <10.1145/2168556.2168642>.

SHARIF, B. et al. An empirical study assessing the effect of seeit 3d on comprehension. In: IEEE. *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. [S.l.], 2013. p. 1–10.

SIEGMUND, J. Program comprehension: Past, present, and future. In: IEEE. *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. [S.l.], 2016. v. 5, p. 13–20.

SIEGMUND, J. et al. Understanding understanding source code with functional magnetic resonance imaging. In: ACM. *Proceedings of the 36th International Conference on Software Engineering*. 2014. p. 378–389. Disponível em: <10.1145/2568225.2568252>.

SINGER, J. et al. An examination of software engineering work practices. In: *CASCON First Decade High Impact Papers*. [s.n.], 2010. p. 174–188. Disponível em: <https://doi.org/10.1145/1925805.1925815>.

SINGH, N.; GIBBS, C.; COADY, Y. C-clr: a tool for navigating highly configurable system software. In: ACM. *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. 2007. p. 9. Disponível em: <10.1145/1233901.1233910>.

SOLOWAY, E.; EHRLICH, K. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, IEEE, n. 5, p. 595–609, 1984.

ŠPAKOV, O.; MINIOTAS, D. Visualization of eye gaze data using heat maps. *Elektronika ir elektrotechnika*, v. 74, n. 2, p. 55–58, 2007. Disponível em: <https://eejournal.ktu.lt/index.php/elt/article/view/10372>.

SPENCER, H.; COLLYER, G. # ifdef considered harmful, or portability experience with c news. *Usenix Summer 1992 Technical Conf.*, Citeseer, p. 185–197, 1992. Disponível em: <http://usenix.org/publications/library/proceedings/sa92/spencer.pdf>.

TARVAINEN, M.; RANTA-AHO, P.; KARJALAINEN, P. An advanced detrending method with application to hrv analysis. *IEEE Transactions on Biomedical Engineering*, v. 49, n. 2, p. 172–175, 2002.

TIARKS, R. What maintenance programmers really do: An observational study. In: CITESEER. *Workshop on Software Reengineering*. [S.l.], 2011. p. 36–37.

UWANO, H. et al. Analyzing individual performance of source code review using reviewers' eye movement. In: ACM. *Proceedings of the 2006 symposium on Eye tracking research & applications*. 2006. p. 133–140. Disponível em: <10.1145/1117309.1117357>.

VILLELA, K. et al. A survey on software variability management approaches. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. New York, NY, USA: Association for Computing Machinery, 2014. (SPLC '14), p. 147156. ISBN 9781450327404. Disponível em: <https://doi.org/10.1145/2648511.2648527>.

VOSSKÜHLER, A. et al. Ogama (open gaze and mouse analyzer): open-source software designed to analyze eye and mouse movements in slideshow study designs. *Behavior research methods*, Springer, v. 40, n. 4, p. 1150–1162, 2008. Disponível em: <10.3758/BRM.40.4.1150>.

WALTER, G. F.; PORGES, S. W. Heart rate and respiratory responses as a function of task difficulty: The use of discriminant analysis in the selection of psychologically sensitive physiological responses. *Psychophysiology*, Wiley Online Library, v. 13, n. 6, p. 563–571, 1976. Disponível em: <https://doi.org/10.1111/j.1469-8986.1976.tb00882.x>.