Journal of Software Engineering Research and Development, 2025, 13:1, doi: 10.5753/jserd.2025.4803

Understanding How Feature Dependent Variables Affect Configurable System Comprehensibility

Djan Santos [**Federal Institute of Bahia** | ***djan.santos@ifba.edu.br***]
Márcio Ribeiro [**Federal University of Alagoas** | ***marcio@ic.ufal.br***]
Cláudio Sant'Anna [**Federal University of Bahia** | ***claudionsa@ufba.br***]

Abstract Background: `#ifdefs` allow developers to define source code related to features that should or should not be compiled. A feature dependency occurs in a configurable system when source code snippets of different features share code elements, such as variables. Variables that produce feature dependency are called dependent variables. 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 use make the analysis of variability scenarios more complex. In fact, previous studies show that `#ifdefs` may affect comprehensibility, especially when their use implies feature dependency. **Aims:** In this sense, our goal is to understand how feature dependent variables affect the comprehensibility of configurable system source code. We conducted two complementary empirical studies. In Study 1, we evaluate if the comprehensibility of configurable system source code varies according to the number of dependent variables. Testing this hypothesis is important so that we can recommend practitioners and researchers the extent to which writing `#ifdef` code with dependencies is harmful. In study 2, we carried out an experiment in which developers analyzed programs with different degrees of variability. Our results show that the degree of variability did not affect the comprehensibility of programs with feature dependent variables. **Method:** We executed a controlled experiment with 12 participants who analyzed programs trying to specify their output. We quantified comprehensibility using metrics based on time and attempts to answer tasks correctly, participants' visual effort, and participants' heart rate. **Results:** Our results indicate that the higher the number of dependent variables the more difficult it was to understand programs with feature dependency. **Conclusions:** In practice, our results indicate that comprehensibility is more negatively affected in programs with higher number of dependent variables and when these variables are defined at a point far from the points where they are used.

Keywords: *Feature dependency, Program comprehension, Configurable Systems, Dependent variable, Eye-tracking, Heart rate monitor*

1 Introduction

Configurable systems address variability by means of features that can be enabled or disabled (Garvin and Cohen, 2011). One of the techniques most used to allow variability is conditional compilation. By means of preprocessor directives, like `#ifdef`, conditional compilation uses feature expressions to allow developers to include or exclude code fragments that will or will not be compiled (Liebig et al., 2010; Garvin and Cohen, 2011). Thus, conditional compilation forces developers to consider multiple scenarios of enabled and disabled features while trying to understand source code. We call these scenarios as variability scenarios.

Previous studies have shown that the presence of `#ifdef` may hinder code comprehensibility (Melo et al., 2017; Schulze et al., 2013; Spencer and Collyer, 1992). For instance, Melo et al. (Melo et al., 2017) compared pieces of source code with and without `#ifdef` and found that `#ifdefs` increase debugging time and require higher visual effort from developers. In addition, studies have identified bugs that occur due to the use of preprocessor directives (Garvin and Cohen, 2011; Medeiros et al., 2013; Tartler et al., 2014; Abal et al., 2014).

Configurable systems usually include a high number of features. Thus, two or more features are likely to share code fragments. When different features refer to the same program element, such as a variable, we have an occurrence of *feature dependency* (Rodrigues et al., 2016). Variables which are shared by features and, as consequence, make those features dependent to each other are called *dependent variables* (Rodrigues et al., 2016).

A configurable system can contain different numbers of features and varying amounts of `#ifdef` usages to define the code segments of those features. In this context, feature dependencies may occur (when features share variables with each other) or not. A study by Melo et al. (2016) identified that the increase of the number of features and `#ifdefs` hindered comprehensibility. However, they were not concerned with whether there were feature dependencies in the source code they analyzed. In addition, our previous study (Santos and Sant'Anna, 2019) found that different types of feature dependencies affected comprehensibility in distinct ways. We analyzed the three types of feature dependencies defined by Rodrigues et al. (2016): intraprocedural, interprocedural and global. These types differ from each other according to the scope of the dependent variables (Rodrigues et al., 2016). For example, global dependency occurs when different features refer to the same global variable and intraprocedural dependency occurs when different features inside a function refer to the same local variable (Rodrigues et al., 2016).

In our previous study, we observed that intraprocedural dependencies impaired comprehensibility less than global and interprocedural dependencies (Santos and Sant'Anna, 2019). In intraprocedural dependencies, the usages of dependent

variables, although occurring across different features, are close to each other as they are located within the same function. This led us to hypothesize that analyzing the content of dependent variables, which is changed by different features, may be the reason that makes the developers' task of understanding source code with `#ifdef` more difficult. To investigate this hypothesis, we carried out the two controlled experiments we report in this paper. We decided to separately investigate the impact of the number of dependent variables (and their usages) (Study 1 in Section 2) and the impact of the number of features and `#ifdef` (Study 2 in Section 3). We reported Study 1 in (Santos et al., 2023) and described and discussed it again here together with Study 2. Therefore, this paper is an extension of our previous conference paper (Santos et al., 2023). In summary, the three studies (including our previous study (Santos and Sant'Anna, 2019) provide a detailed analysis of how different characteristics of source code with `#ifdef` can affect comprehensibility.

In Study 1, we study how the number of dependent variables affects the comprehensibility of configurable systems implemented with `#ifdefs`. We executed our experiment with 12 participants who analyzed programs trying to specify their output.

In Study 2, We investigated how the comprehensibility of configurable systems was affected by degrees of variability taking into account dependent variables. We define two degrees of variability: We considered 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. A feature constant is used in feature expressions for conditional compilation. In study 2, we fixed the number of feature dependencies and the number of dependent variables on all programs and varied degrees of variability.

Our experiments are multifaceted in terms of the metrics we collected to analyze comprehensibility. In particular, we collected time and number of attempts the participants had taken to finish tasks and we used two equipment: (i) eye tracking, to measure a variety of visual efforts and cognitive processes of participants, such as fixations, gaze transition diagrams, scan paths, and heat maps, and (ii) a heart rate monitor on a smartwatch, to measure a variety of physiological processes of participants, such as heart rate variability (HRV). HRV measures the variation in time between heartbeats, providing information on the balance between the sympathetic and parasympathetic branches of the autonomic nervous system. HRV is often associated with stress and emotion regulation (Walter and Porges, 1976; Fritz et al., 2014; Hijazi et al., 2021).

Our findings show that feature dependency affects comprehensibility in different ways. We observed that: (i) participants spent more time to analyze programs with more dependent variables, (ii) programs with more dependent variables required more visual effort. On the other hand, we could not find significant difference in the number of participants' stressful moments when trying to understand programs with different numbers of dependent variables. Also, the number of dependent variables did not affect the number of attempts needed to specify program outputs and the degree of variability did not affect the comprehensibility of programs with feature dependency. In a nutshell, our experiment results support that dependent variables may hinder the comprehensibility of configurable systems. Therefore, if a program has a high number of dependent variables, developers need to direct their attention to these variables more frequently, increasing the time and effort required for comprehension. We recommend that, whenever possible, developers should use few dependent variables.

Listing 1: Program 1: Sale of real estate domain

```
1  struct Properties {
2    float salesPrice = 0;
3    int available = 1;
4
5    #ifdef COMMISSION
6    float commissionValue = 0;
7    #endif
8
9    #ifdef TAX
10   float taxes = 0;
11   #endif
12 } property;
13
14 float calculatePropertyValue(float costPrice) {
15   float serviceFee = 0;
16
17   if (property.available == 0) {
18     printf("Property blocked! Sale not made!");
19     property.salesPrice = 0;
20     return property.salesPrice;
21   } else {
22
23     #ifdef SERVICE_FEE_DISCOUNT
24     if (costPrice > 5000) {
25       serviceFee = 100;
26     } else {
27       serviceFee = 50;
28     }
29     #endif
30
31     #ifdef TAX
32     property.taxes = costPrice / 10;
33     serviceFee += property.taxes;
34     printf("Property taxes = %.f", property.
           taxes);
35     #endif
36
37     #ifdef COMMISSION
38       property.commissionValue = costPrice /
             20;
39       serviceFee += property.commissionValue;
40       printf ("Commission = %f", property.
             commissionValue);
41     #endif
42
43     property.salesPrice = costPrice +
           serviceFee;
44
45     return property.salesPrice;
46   }
47 }
48
49 int main() {
50     float costPrice = 0;
51     scanf("%f", &costPrice);
52     printf("Sales price = %f",
           calculatePropertyValue (costPrice));
53 }
```

Conditional compilation implies feature dependency whenever a developer defines a variable in a feature and

uses it in another feature or when two or more features use the same variable (Baniassad and Murphy, 1998; Ribeiro et al., 2010, 2014; Rodrigues et al., 2016). For instance, in Listing 1, the `serviceFee` variable is defined outside any `#ifdef`, meaning that it is defined in a mandatory feature, which, in this example, we can call as the sale of real estate feature. In addition, `serviceFee` is used in features `SERVICE_FEE_DISCOUNT` (lines 25 and 27), `TAX` (line 33), and `COMMISSION` (line 39). Therefore, there is feature dependency between (i) `SERVICE_FEE_DISCOUNT` and `TAX`, (ii) `SERVICE_FEE_DISCOUNT` and `COMMISSION` and (iii) `TAX` and `COMMISSION`. Also, there are feature dependencies between the mandatory feature and each of the other three features. In this example, the `serviceFee` variable is called *dependent variable* as it is the element that causes the dependency.

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). Degrees of variability in configurable systems refer to the number of different program variants from common source code (Melo et al., 2017). In source code with `#ifdef`, we can quantify degrees of variability by the number of different feature constants and the number of different feature expressions. Taking Listing 1 as a motivating example, this program has 5 feature expressions (lines 5, 9, 23, 31, and 37) and 3 feature constants (`COMMISSION`, `TAX`, and `SERVICE_FEE_DISCOUNT`). Considering the programs used in this study, Listing 1 is a program with more variability.

In Listing 1, we can see that, to understand the behavior of `serviceFee` dependent variable, the developer should at least consider: (i) `SERVICE_FEE_DISCOUNT`, `TAX`, and `COMMISSION` features enabled, (ii) `SERVICE_FEE_DISCOUNT`, `TAX`, and `COMMISSION` features disabled, (iii) `SERVICE_FEE_DISCOUNT` feature enabled and `TAX`, and `COMMISSION` features disabled, (iv) `TAX` feature enabled and `SERVICE_FEE_DISCOUNT`, and `COMMISSION` features disabled, (v) `COMMISSION` feature enabled and `SERVICE_FEE_DISCOUNT` and `TAX` features disabled, (vi) `SERVICE_FEE_DISCOUNT` and `TAX` features enabled and `COMMISSION` feature disabled, (vii) `COMMISSION` and `SERVICE_FEE_DISCOUNT` features enabled and `TAX` feature disabled, and (viii) `SERVICE_FEE_DISCOUNT` feature disabled and `TAX` and `COMMISSION` features enabled.

In this case, we considered a piece of source code with only one dependent variable. Mentally simulating all these scenarios with a larger number of dependent variables and feature dependencies may increase the program comprehension effort. Also, we can see that, for programs with less variability, i.e. fewer number of features, the developer has fewer possible scenarios to analyze. For programs with more variability, the developer needs to analyze more features and consequently a greater number of possible scenarios.

The main goal of our experiments is to evaluate how dependent variables impact the comprehensibility of configurable systems. A dependent variable is what imposes a dependency relationship between features. The variation in the amount of variable definition and variable usage can affect the comprehensibility of configurable systems. In this sense, the following research question guides our study 1:

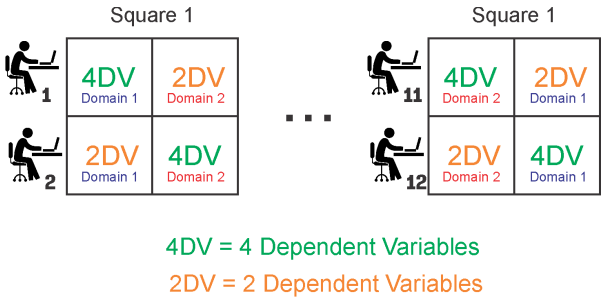


Figure 1. Latin Square design (2x2).

RQ1 – *How does the number of dependent variables affect the comprehensibility of configurable system source code?*

Other aspects also need to be considered when referring to feature dependent variables, like degrees of variability. In this context, we decided to complement our investigation by Study 2 by answering the following research question:
RQ2 – *How do degrees of variability affect the comprehensibility of configurable system?*

2 Study 1: How the number of dependent variables affects comprehensibility.

To answer our research question one (RQ1), we carried out a controlled experiment with 12 developers who analyzed programs trying to specify their output.

2.1 Experiment Settings

We implemented four programs for the experiment. They are similar to each other in terms of number of lines of code and cyclomatic complexity. As we were only concerned with the number of dependent variables, we fixed the number of features, number of feature expressions and the number of feature dependencies. All programs have three features and seven feature dependencies.

In order to avoid learning effect, two of the programs are on the sale of real estate domain (Domain 1) and the other two are on the grade calculation domain (Domain 2). For each domain, one program has two dependent variables and the other one has four dependent variables. In summary, for each domain, we implemented two programs with different numbers of dependent variables.

We compared the comprehension effort the developers spent to analyze each program. We quantified comprehension effort from different perspectives: (i) time until the developers provide the correct answer for the tasks, (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 designed our experiment as a standard Latin Square, which is a common solution for this kind of experiment (Bailey, 2008; Melo et al., 2017). The Latin square design controls one factor and its variations, ensuring that no row or column contains the same treatment twice. Figure 1 explains our 2x2 Latin Square. The rows represent developers. In its

columns, we have the different domains. The acronyms in the cells represent the treatments: 4DV represents a program with 4 dependent variables, and 2DV represents a program with 2 dependent variables. Developer 1 first analyzed a program with 4 dependent variables and, afterward, a program with 2 dependent variables. Developer 2 also analyzed programs with 4 and 2 dependent variables but in reverse order.

Listing 2: Program 2: Sale of real estate domain with 4 dependent variables

```
1  struct Properties {
2    float salesPrice = 0;
3    int available = 1;
4
5    #ifdef COMMISSION
6    float commissionValue = 0;
7    #endif
8
9    #ifdef TAX
10   float taxes = 0;
11   #endif
12 } property;
13
14 float calculatePropertyValue(float costPrice) {
15   float serviceFee = 0;
16   float extraFee = 0;
17
18   if (property.available == 0) {
19     printf("Property blocked! Sale not made!");
20     property.salesPrice = 0;
21     return property.salesPrice;
22   } else {
23
24     #ifdef SERVICE_FEE_DISCOUNT
25     if (costPrice > 5000) {
26       serviceFee = 100;
27     } else {
28       serviceFee = 50;
29     }
30     #endif
31
32     #ifdef TAX
33     property.taxes = costPrice / 10;
34     property.salesPrice += property.taxes;
35     printf("Value of property taxes = %f",
           property.taxes);
36     #endif
37
38     #ifdef COMMISSION
39     property.commissionValue = costPrice / 20;
40     extraFee += property.commissionValue;
41     printf("Comission = %f", property.
           commissionValue);
42     #endif
43
44     property.salesPrice += costPrice +
           serviceFee + extraFee;
45
46     return property.salesPrice;
47   }
48 }
49
50 int main() {
51   float costPrice = 0;
52   scanf("%f", &costPrice);
53   printf("Sales price = %f",
         calculatePropertyValue(costPrice));
54 }
```

To avoid learning effects due to 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 with different numbers of dependent variables and different domains.

We counted on 12 participants to run our experiment: six programming professors, and six developers from the industry. We selected professors from the Federal Institute of Bahia, a university in Brazil, and developers from three companies in the same country. No compensation was provided for the participants. All participants had experience with the C language and were already familiar with `#ifdef`, as shown in Table 1.

Table 1. Participants' experience summary

	Programming professor	Developer from the industry
Participants	6	6
Gender	3 male and 3 female	5 male and 1 female
Degree	4 Masters and 2 Ph.D.	1 Master and 5 Graduates
C Experience	4: more than 15 years 2: between 10-15 years	1: more than 15 years 3: between 10-15 years 2: between 5-10 years
Prior Experience with #ifdef	6	6

2.2 Programs

We implemented the programs for our experiment inspired by real configurable systems (Abal et al., 2014; Melo et al., 2017). We avoided pieces of code from 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, we wrote our programs in Portuguese, the participant's native language.

We used an eye-tracking device on a 32-inch screen to record all gaze movements of participants. Our programs should fit on a 50-line display window so that the eye-tracking device could record all gaze movements of participants.

Table 2. Metrics

Domain	Program	LOC	CC	NFE	NOFC	NDV	NFD
Domain 1	**Program 1**	53	6	6	3	2	7
	Program 2	54	6	6	3	4	7
Domain 2	**Program 3**	49	7	8	3	2	7
	Program 4	46	8	8	3	4	7

In the same domain, the programs are similar in terms of number of lines of code (LOC) (Lanza and Marinescu, 2007), McCabe cyclomatic complexity (CC) (McCabe, 1976), number of feature expressions (NFE) (Liebig et al., 2010), number of features (NOFC) (Liebig et al., 2010) and number of feature dependencies (NFD), as shown in Table 2. All programs have three features and seven feature dependencies. In this case, we were particularly concerned with the variation in the number of dependent variables (NDV). In the following, we describe each program.

Program 1: sale of real estate with 2 dependent variables. Listing 1 (Section 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` is defined in a mandatory feature in line 14 and `service- Fee` is also defined in a mandatory feature but in line 15. The variable `costPrice` had 3 uses that generate dependencies: (i) in line 24 it is used within `SERVICE_FEE_DISCOUNT` feature, (ii) in line 32 it is used within `TAX` feature and (iii) in line 38 it is used within `COMMISSION` feature. The variable `serviceFee` had 4 uses that generate dependencies, already described in Section 1.

Program 2: sale of real estate with 4 dependent variables. Listing 2 shows Program 2. We rewrote the program shown in Listing 1 by including 2 dependent variables: `salesPrice`, defined in line 2, and `extraFee`, defined in line 16.

The variable `salesPrice` is defined within a mandatory feature. It causes a dependency because it is used within `TAX` feature in line 34. The variable `extraFee` is also defined within a mandatory feature. It causes a dependency because it is used within `COMMISSION` feature in line 40.

Program 3 and Program 4: We replicated the same characteristics of Programs 1 and 2 in Programs 3 and 4. They are available at our research share website.[1] Grade calculation is the domain of these programs.

2.3 Experiment Procedures

This study was conducted in accordance with the ethical guidelines and regulations established by the Federal Institute of Bahia Ethics Committee. Approval for the study was obtained from the committee before data collection.

We performed each trial of the experiment with each participant individually. Before starting the experiment tasks, we cleaned all material and equipment with alcohol gel and asked the participants to fill out a consent form. All participants signed the consent form.

Then, the participant put on the smartwatch and we calibrated the eye-tracking device and the smartwatch. We also synced the clock of the smartwatch with the time of the eye-tracking computer. Before the participant started 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 participants started 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. We learned, from previous studies, that a person's biometric features drop back to a baseline after about a minute of watching the video (Fritz et al., 2014; Müller and Fritz, 2015).

After the video, the participant analyzed the programs as we planned in our Latin square design described in Section 2. We observed the participant and monitored the time that she or he spent completing each task. The participant had three tasks per program. We give more details about the tasks in Section 2.4. We used the tool that record the gaze data to record the time. For each new task, the tool reset the timer. We checked when the participant correctly specified the output of the program and then stopped recording the time. 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 answer 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 device[2] and the Garmin Fenix 5s smartwatch[3].

2.4 Tasks

Each participant received a task instruction form that explained the experiment. Each participant had to understand and realize the mental execution of two programs, one with four dependent variables and the other with two dependent variables. The order of programs depends on the 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 as follows:

TASK 1: Consider:

FEATURES ENABLED: `SERVICE_FEE_DISCOUNT`, `COMMISSION` and `TAX`.

FEATURES DISABLED: none.

INITIAL VALUES: `costPrice` = 2000 in line 52.

QUESTION: "What will be printed on line 50 when the `int main( )` function on line 39 is executed?"

2.5 Experimental Results

In this section, we test our hypotheses and discuss the results. In this study, we use Analysis of Variance (ANOVA) 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

[1] https://zenodo.org/records/13308778

[2] https://gaming.tobii.com/product/eye-tracker-5/

[3] https://www.garmin.com/en-US/p/552237/

hypotheses. We ran our tests with the support of R.[4] In the following, we present the results regarding each metric.

2.6 Time to provide the correct answer

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.

Table 3. 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

Table 3 shows 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, for four programs we had 72 observations. Shapiro test confirmed that the data about time was normally distributed.

Table 3 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 participants 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 four dependent variables required more time for the participants to answer the tasks correctly than programs with two dependent variables.

2.7 Number of attempts needed until correct answer

We measured the total number of attempts needed for the participants until they specified the correct output of the programs. The participant scored one for each attempt for each task correctly answered. This metric sums 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 4. 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

Table 4 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. Our data revealed no significant difference between the number of attempts needed to specify the output correctly for programs with 2 and 4 dependent variables. The p-value is 0.56163. Based on this, we cannot reject our null hypothesis H_0a.

This result confirms previous studies that showed that most participants correctly executed tasks in programs with `#ifdef` (Melo et al., 2016, 2017; Santos and Sant'Anna, 2019).

Result 2: We could not find significant difference in the number of attempts needed for the participants until giving the correct answer when comparing programs with two and four dependent variables.

2.8 Visual effort

Regarding visual effort, we quantified the total number of fixations executed by the participants. Also, we analyzed their gaze transitions and attention maps.

2.8.1 *Total number of fixations*

The number of fixations increases when a text is difficult to comprehend (Rayner, 2009). 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 needed until providing the correct answer when comparing programs with different numbers of dependent variables.

Table 5. 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

Table 5 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).

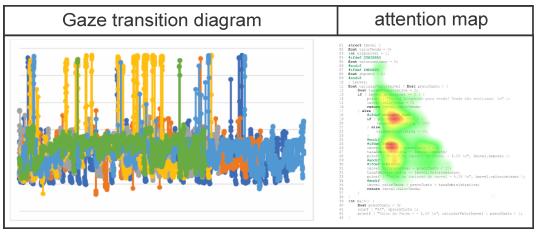
Result 3: Participants made more fixations to analyze programs with four dependent variables.

2.8.2 *Gaze transitions and attention maps*

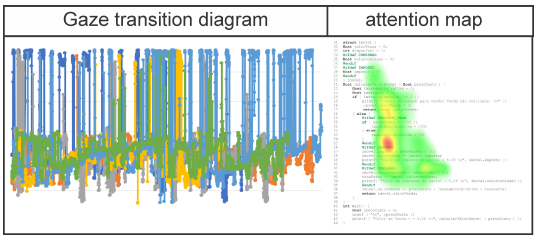
Gaze transitions (saccades) are rapid eye movements from one place to another separated by pauses (Rayner, 2009). A larger number of saccades in both directions indicates difficulty associated with understanding (Rayner, 1998, 2009). An attention map is a heat map that displays an aggregation of fixations.

Figure 2a and Figure 2b show our gaze transitions diagrams and attention maps related to programs 1 and 2 respectively. Figure 3a and Figure 3b show our gaze transitions

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

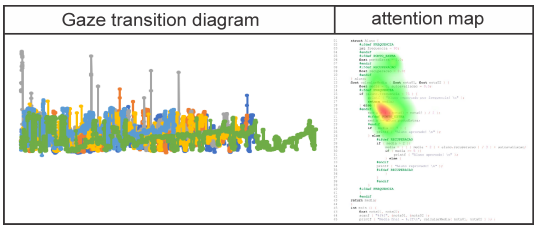


(a) Program 1 with 2 dependent variables

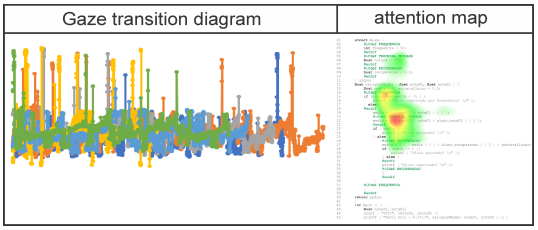


(b) Program 2 with 4 dependent variables

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



(a) Program 3 with 2 dependent variables



(b) Program 4 with 4 dependent variables

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

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 2b), which has four dependent variables, than on Program 1 (Figure 2a) which has two dependent variables. Program 1 and Program 2 are programs developed in the same domain 1. It is possible to observe in Figure 2b a higher number of transitions toward the top of the source code than in Figure 2a. 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 2a) reveals that participants' attention was directed to the usages of `finalGrade` variable. We hypothesize that this occurs because `finalGrade` variable concentrated most of dependencies in this program generating more cognitive effort. The attention map of Program 2 (Figure 2b) reveals that participants executed more transitions closed to 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.

For programs in Domain 2, the gaze transitions diagrams reveal that participants executed more transitions on Program 4 (Figure 3b), which has four dependent variables, than Program 3 (Figure 3a), which has two dependent variables. This scenario is the same as Domain 1. We observed in Figure 3b a higher number of transitions toward the top of the source code than in Figure 3a. 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 3a) reveals that participants' attention was directed to the usages of `serviceFee` variable. The attention maps show the two red regions in the dependent variable. We hypothesize that this occurs because `serviceFee` variable concentrated most of dependencies in this program attracting more attention to this area.

In summary, programs with more dependent variables force the developer to direct their attention to more regions, causing a more spread distribution of attention and transition over distinct parts of source code.

2.9 Heart-related biometrics

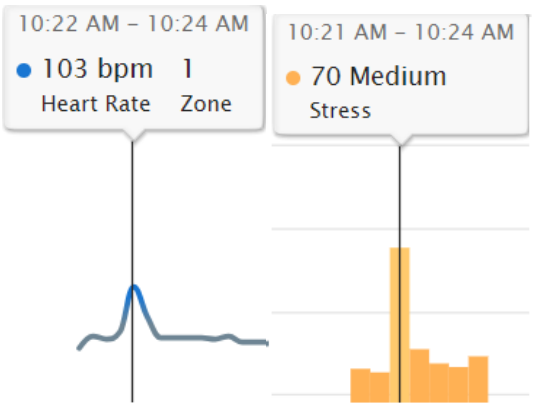
Previous research has shown that heart-related biometrics can be linked to difficulty in comprehending small code snippets (Walter and Porges, 1976; Fritz et al., 2014; Nakagawa et al., 2014; Müller and Fritz, 2015). The general concept behind these studies is that heart-related biometrics can be used to determine the 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). A Heart Rate Variation (HRV) measures the variation in time between heartbeats (Walter and Porges, 1976). In our study, we identified the total number of stressful moments experienced by the participants. To identify participants' stressful moments, we monitored participants' heartbeats during task execution and recorded the moments when there were variations in the heart rate.

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

$H_0 h$: There is no significant difference in the number of stressful moments to specify the output of the programs when comparing programs with different numbers of dependent variables.

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

Figure 4 shows an example of data captured by the smartwatch. Visually, in Figure 4a we detected a variation (stressful moments) in the heart rate of one of the participants. Between 10:22 AM and 10:24 AM the participant registered



(a) HRV **(b)** Stress Level

Figure 4. Stressful moments collected by smartwatch

Table 6. Total number of stressful moments

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

103 heartbeats per minute. Then, we check if this value is above the mean and standard deviation.

All stressful moments 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.

Table 6 shows the sum of number of stressful moments of all participants during the tasks and the total number of different participants accounted for these stressful moments. Four participants had 4 stressful moments for programs with 2 dependent variables and six participants had 8 stressful moments for programs with 4 dependent variables. Our data revealed no significant difference between the number of stressful moments needed to specify the output correctly in programs with 2 and 4 dependent variables. The p-value 0.5381. Based on this, we cannot reject our null hypothesis $H_0 h$.

Result 4: We could not find significant difference in the number of stressful moments when comparing programs with two and four dependent variables.

2.10 Gaze Movements Analysis

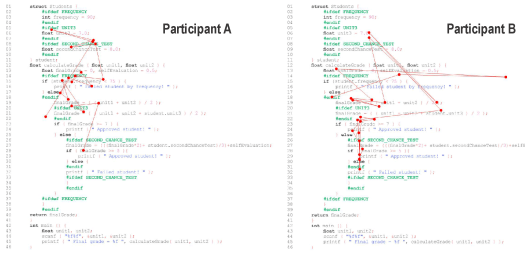
Our Result 4 shows that there was no significant difference in terms of the number of stressful moments while participants analyzed programs with different numbers of dependent variables. Despite this, we decided to investigate what was happening in terms of gaze movements during stressful moments.

An scan path is 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 4 shows an example of stressful moments of one of the participants. Based on Figure 4, we tried to identify what could have caused a variation in the participant's heart rate during source code analysis. For that, we used a scan path to identify where he or she was looking at the moment of the stressful moments and why it might have generated cognitive effort.

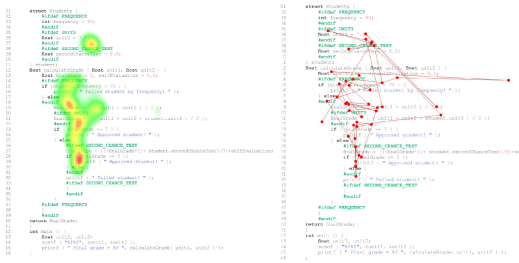
Table 6 shows that 2 participants had 2 stressful moments analyzing Program 1 (2 dependent variables). We generated two scan paths, each one for each moment when each of the two stressful moments happened. In both cases, the participants were executing tasks with all features enabled. Figure 5a shows the two scan paths. 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 5b we superimposed the two individual scan paths corresponding to stressful moments. We also generated attention maps corresponding to the participants' stressful moments. Figure 5b shows the gaze movements of participants were concentrated in the region of variable definitions and variable usages when stressful moments 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 6a represents scan paths of participants analyzing Program 3, another program with 2 dependent variables. Two participants register stressful moments. In the first scan path, the participant was considering 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 stressful moments. In the second scan path, the participant was considering the `TAX` feature disabled. The participant navigated along the source code trying to resolve the task. Figure 6b is composed by the overlapping of the two individual scan paths and attention maps during stressful moments. The red region on `COMISSION` feature indicates that participants' attention was directed to the definition and usages of the variables of the enabled feature. We hypothesize the stressful moments 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.

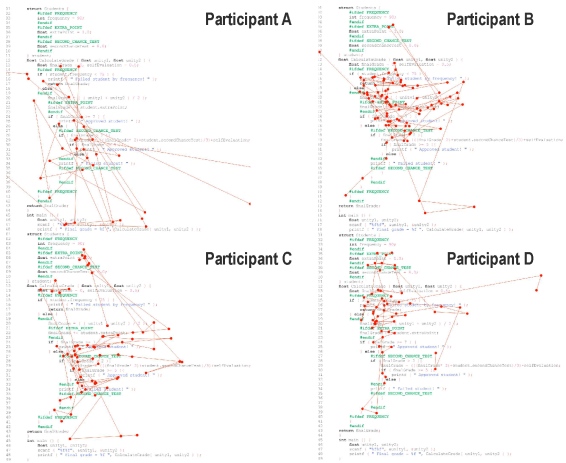
Figure 5c shows four scan paths participants performed in Program 2 (4 dependent variables) during stressful moments. The participants performed long saccades and regressions between variable definitions and variable usages. In Figure 5d, we have the overlapping of the four individual scan paths and attention maps corresponding to participants' gaze movements during the four stressful moments occurrences. Figure 5d confirms that gaze movements concentrated in the region of variable definitions and variable usages when stressful moments occurred. The red region indicates that participants' attention was directed to the usage of `finalGrade` variable. Figure 5d shows only one red region in programs with 4 dependent variables, and Figure 5b 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



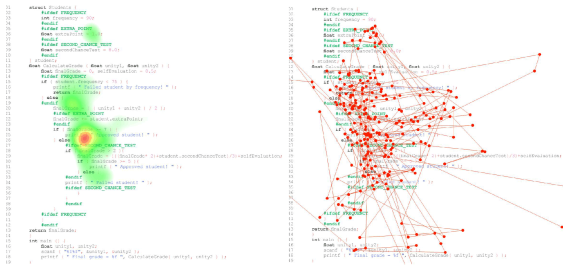
(a) Individual scan path of participants analyzing Program 1.



(b) Gaze movements of participants analyzing Program 1.



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



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

Figure 5. Gaze movements of Program 1 and 2 at the moment of stressful moments

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 6c and 6d.

In summary, participants seem to have performed long saccades and regressions during stressful moments. Also, stressful moments 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 of only few participants.

2.11 Threats to Validity

2.11.1 External validity

Programs. Due to limitations of the eye-tracking device, 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 50 lines of code and more than three features, there may be additional effects that we have not observed.

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. This is different from a usual real work environment. However, we had this limitation because the source code should fit on the screen to properly work with our eye-tracking device.

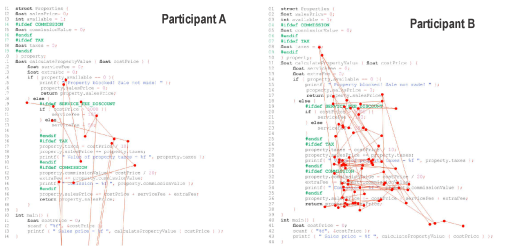
Participants. This study had a relatively small number of participants. However, we took steps to minimize threats to validity and ensure reliable results. First, we selected participants with programming experience. We collected data from professional developers and professors of programming languages. Additionally, participants were evaluated under controlled conditions. This helped to minimize unexpected effects and ensure that the results were accurate.

Experimental context. Heart rate data were collected in a simulated programming environment, where participants worked on small-scale programming tasks. These tasks were constrained in terms of the number of lines of code to comply with the requirements of the eye-tracking device, which imposed limitations on session duration. Consequently, it is possible that these tasks did not generate sufficient cognitive load or emotional engagement to elicit measurable variations in heart rate. This limitation may affect the generalizability of our findings to real-world programming scenarios, where tasks are typically more complex and prolonged.

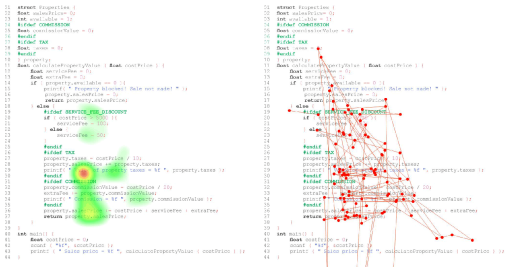
2.11.2 Internal validity

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.

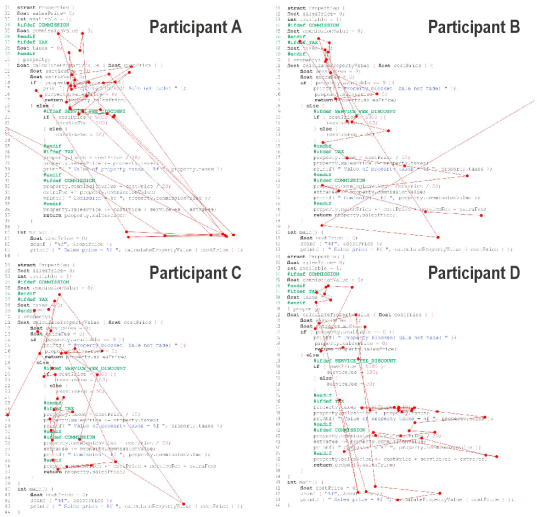
Participants' experience. We controlled confounding factors via the Latin square design and randomization. We selected 12 participants experts in language C. They are programming language professors and developers from industry.



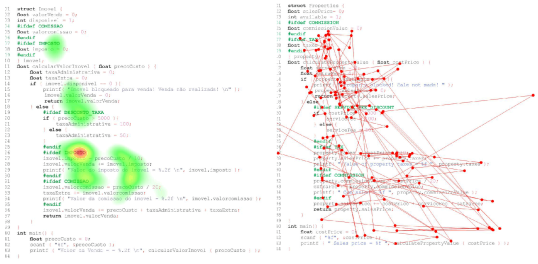
(a) Individual scan path of participants analyzing Program 3.



(b) Gaze movements of participants analyzing Program 3.



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



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

Figure 6. Gaze movements of Program 3 and 4 at the moment of stressful moments

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 one of the authors.

2.11.3 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.

3 Study 2 - How degrees of variability affect program comprehensibility

To answer our research question two (RQ2), we carried out a controlled experiment with 12 developers, who analyzed programs trying to specify their output.

3.1 Experiment Settings

We performed this experiment with the same participants of Study 1 (Section 2). We took advantage of the availability of the participants and also executed the tasks of this experiment just after the tasks of Study 1. So, we controlled confounding factors using the same experimental design and procedures of Study 1 (Section 2.1).

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 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. We used programs with a maximum of 6 feature expressions due to display limitations of the code on the screen.

We quantified participant comprehension effort by means of: (i) time to analyze each program, (ii) the 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: sale of products (Domain 1) and game of hit the target (Domain 2). We fixed the number of feature dependencies (6 dependencies) and the number of dependent variables (3 dependent variables) as the same for all programs. Then, we implemented programs with different degrees of variability. Two programs were implemented with less variability and the other two were implemented with more variability. In summary, for each domain, we had two programs with different degrees of variability.

We designed our experiment as a standard Latin Square. Study 1 discusses Latin Square design (Section 2.1). Figure

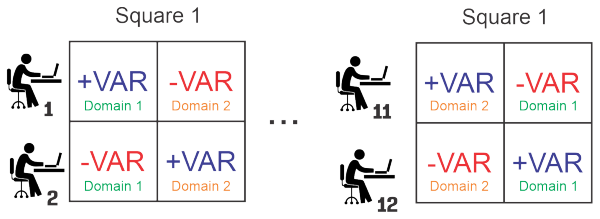


Figure 7. Latin Square design (2x2).

7 explains our 2x2 Latin Square. In its cells, we have the treatment, in this case, the degree of variability. The acronyms in the cells represent the program characteristics: +VAR represents a program with more variability and -VAR represents a program with less variability. The lines represent developers. Developer 1 firstly analyzed a program with more variability, and, afterwards, a program with less variability. Developer 2 also analyzed both programs but in reverse order.

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

```
1  struct Products {
2      int totalProductsForSale = 10;
3      #ifdef PRODUCT_CONTROL
4      int totalProductsPurchased = 20;
5      int minimumNumberProducts = 5;
6      #endif
7      int totalProductsSold = 0;
8  } product;
9      float sellProduct ( int numberOfProducts )
           {
10         float unitaryValue = 5.0;
11         #ifdef PRODUCT_CONTROL
12         if ( product.totalProductsForSale -
               numberOfProducts < 0 ) {
13             printf( "Insufficient number of
                   products!" );
14             return 0;
15         } else {
16             if ( product.totalProductsForSale <
                    product.minimumNumberProducts
                   ) {
17                 printf ( "Last units!
                       Readjusted price" );
18                 unitaryValue = unitaryValue +
                       2.0;
19                 product.totalProductsForSale +=
                        product.
                       totalProductsPurchased;
20             }
21             #endif
22             printf( "Product sold." );
23             product.totalProductsForSale -=
                   numberOfProducts;
24             product.totalProductsSold +=
                   numberOfProducts;
25         #ifdef PRODUCT_CONTROL
26         }
27         #endif
28         return numberOfProducts * unitaryValue;
29     }
30 int main() {
31     int numberOfProducts = 0;
32     scanf ( "%d", &numberOfProducts );
33     printf ( "Sale price = %f", sellProduct (
           numberOfProducts ) );
34 }
```

To avoid learning effects due repetition of domains we distributed the domains along our Latin square column as in our Study 1 (Section 2.1). Thus, each developer analyzed two different programs, each one in a domain different to the other. In total 12 participants ran our experiment, the same individuals who participated in Study 1 (Section 2.1).

3.2 Programs

Within the same domain, the programs are similar in terms of number of lines of code (LOC) (Lanza and Marinescu, 2007), McCabe cyclomatic complexity (CC) (McCabe, 1976), number of dependent variables (NDV) and and number of feature dependencies (NFD). All programs have three dependent variables and six feature dependencies, as shown in Table 7. In this case, we were particularly concerned with the variation in the number of feature expressions (NFE) (Liebig et al., 2010) and number of features (NOFC) (Liebig et al., 2010).

Table 7. Metrics

Domain	Program	LOC	CC	NFE	NOFC	NDV	NFD
Domain 1	**Program 1**	34	6	3	1	3	6
	Program 2	40	6	6	3	3	6
Domain 2	**Program 3**	40	7	3	1	3	6
	Program 4	44	8	6	3	3	6

Listing 4: Program 2: Sale of products domain with 6 feature expressions and 3 feature constant

```
1  struct Products {
2      int totalProductsForSale = 10;
3      #ifdef BUY_PRODUCT
4      int totalProductsPurchased = 20;
5      #endif
6      #ifdef MINIMUN_NUMBER_PRODUCTS
7      int minimumNumberProducts = 5;
8      #endif
9      int totalProductsSold = 0;
10 } product;
11 float sellProduct ( int numberOfProducts ) {
12     float unitaryValue = 5.0;
13     #ifdef PRODUCT_CONTROL
14     if ( product.totalProductsForSale -
           numberOfProducts < 0 ) {
15         printf ( "Insufficient number of
               products!" );
16         return 0;
17     } else {
18     #endif
19         product.totalProductsForSale -=
               numberOfProducts;
20         #ifdef MINIMUN_NUMBER_PRODUCTS
21         if ( product.totalProductsForSale <
               product.minimumNumberProducts ) {
22             printf ( "Last units! Readjusted
                   price" );
23             unitaryValue = unitaryValue + 2.0;
24             #ifdef BUY_PRODUCT
25             product.totalProductsForSale +=
                   product.totalProductsPurchased;
26             #endif
27         }
28         #endif
29         printf ( "Product sold." );
30         product.totalProductsSold +=
               numberOfProducts;
31         #ifdef PRODUCT_CONTROL
32     }
33         #endif
34     return numberOfProducts * unitaryValue;
```

```
35 }
36 int main() {
37     int numberOfProducts = 0;
38     scanf ( "%d", &numberOfProducts );
39     printf ( "Sale Price = %f", sellProduct (
           numberOfProducts));
40 }
```

We implemented the programs for this experiment inspired by our previous study (Santos and Sant'Anna, 2019) and by common programming tasks. In the following, we describe each program.

Program 1: sale of products with less variability. Listing 3 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.

Program 2: sale of products with more variability. Listing 4 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.

Program 3 and Program 4: We replicated the same characteristics of Programs 1 and 2 in Programs 3 and 4. They are available at our research share website.[5] Calculate a player's score is the domain of these programs.

3.3 EXPERIMENT PROCEDURES

The procedures of this study are the same of Study 1, described in Section 2.3.

3.4 Tasks

This experiment was carried out together with the experiment of Study 1 (Section 2). 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 three tasks for this study 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 programs depended on the Latin Square.

The programs with one feature constant 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 variables values, (iii) Task 03: all features disabled.

We explained the proposed scenarios and initial values of each task to each participant before he or she started the task. The participant could also find similar 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 similar. The difference is only the names of features enabled and disabled. And, according to 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 research share website.[6] In the following, we present the results regarding each metric.

3.5 Experimental Results

3.5.1 Time to provide the correct answer

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.

Table 8. 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

Rows in Table 8 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 8 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.

[5] https://zenodo.org/records/13308778

[6] https://zenodo.org/records/13308778

3.6 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 the number of attempts needed until the correct answer when comparing programs with different degrees of variability.

Table 9. 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

Table 9 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 and 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.

3.7 Visual effort

Regarding visual effort, as in study 1 (Section 2), we quantified the total number of fixations executed by the participants. Also, we analyzed their gaze transitions and attention maps.

Total number of fixations

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

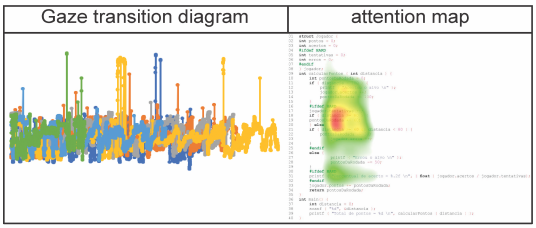
H_0f: There is no significant difference in the number of fixations to specify the output when comparing programs with different degrees of variability.

Table 10. Number of fixations

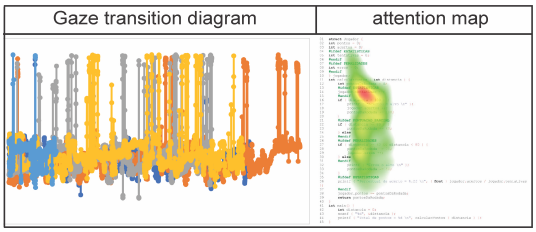
With less variability			With more variability		
Program 1	**Program 3**	**all programs**	**Program 2**	**Program 4**	**all programs**
195	151	346	218	219	437

Table 10 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 8. Gaze transitions diagram and attention map of programs 3 and 4.

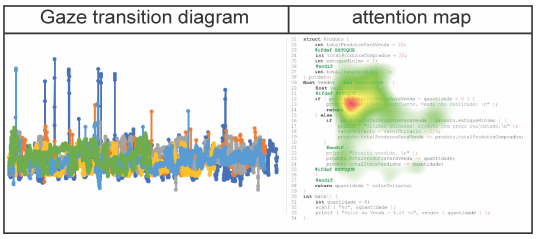
Gaze transitions and attention map

Figure 8a and Figure 8b show our gaze transitions diagrams and attention maps related to programs 1 and 2 respectively. Figure 9a and Figure 9b 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.

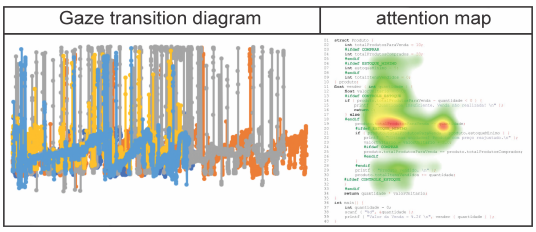
The gaze transitions diagrams reveal that participants executed more transitions on Program 2 (Figure 8b), which has more variability, than Program 1 (Figure 8a), which has more variability. Program 1 and Program 2 are programs developed in the same domain 1. It is possible to observe in Figure 8b a greater distribution of transitions toward the top of the source code than in Figure 8a. 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 8a) reveals that participants' attention was directed to the usages of `roundPoints` variable. 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 8b) reveals that participants, when analyzing programs with more variability, needed to navigate overall source code, and, as a consequence, the attentions are more distributed along the source code. Programs with more variability force the participant to simulate different configurations of enabled/disabled features.

For programs in domain 2, the gaze transitions diagram reveals that participants executed more transitions on Program 4 (Figure 9b), which has more variability, than Program 3 (Figure 9a) which has less variability. This scenario is the same as domain 1. We observed in Figure 9b a greater distribution of transitions toward the top of the source code than transitions in Figure 9a. When analyzing programs with more variability, participants navigated between feature expression regions, and, as a consequence, the gaze transitions



(a) Program 3 with less variability



(b) Program 4 with more variability

Figure 9. Gaze transitions diagram and attention map of programs 3 and 4.

were more distributed along the source code. In these cases, participants needed to realize long transitions to feature expression regions and variable definitions regions.

The attention map of Program 3 (Figure 9a) reveals that participants' attention was directed to the usages of `totalProductsForSale` variable. 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 3b), also shows that participants' attention was directed to the usages of `totalProductsForSale` variable. The attention maps show a more widespread distribution of attention overall source codes. This leads to the following result.

In summary, programs with more variability forced developers to direct their attention to more regions, causing a more widespread distribution of attention and transition over distinct parts of source code.

3.8 Heart-related biometrics

We counted the number of stressful moments for programs with different degrees of variability, as in study 1 (Section 2). Our null hypothesis about this metric is:

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

Table 11. Total number of stressful moments

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

Table 11 shows the total number of stressful moments of participants during the tasks. Four participants had 5 stressful moments for programs with less variability and six participants had 8 stressful moments for programs with more variability. The χ^2 test revealed no significant difference between the number of stressful moments 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 4: There was no significant difference in the number of stressful moments until giving the correct answer when comparing programs with different degrees of variability.

3.9 GAZE MOVEMENTS ANALYSIS

Our Result 4 shows that there was no significant difference in terms of stressful moments while partici-pants analyzed programs with different degrees of variability. However, as in study 1 (Section 2), we decided to investigate what was happening in terms of gaze movements during stressful moments.

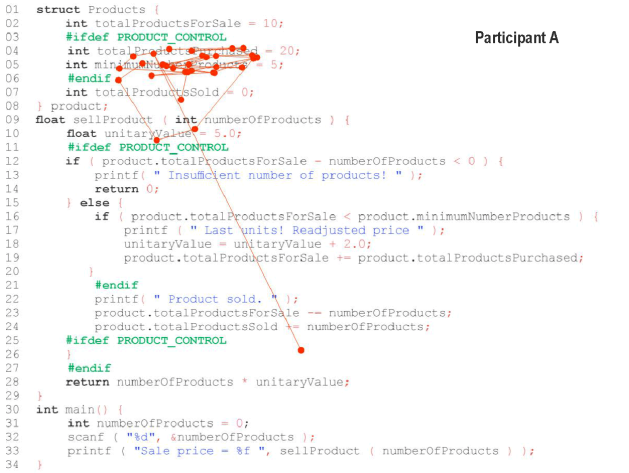
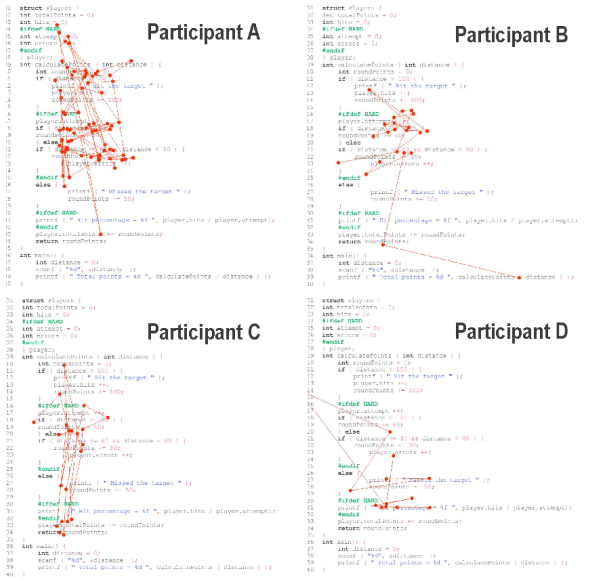


Figure 10. Individual scan path of participants from program 1 (less variability).

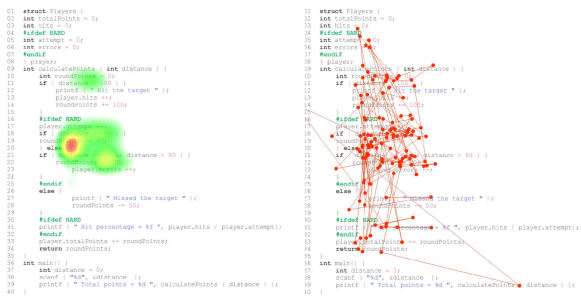
Figure 11. Gaze movements diagrams of Program 1 (less variability).

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

Figure 10 shows a scan path of the only participant who had stressful moments while analyzing Program 1. We generated a scan path at the moment the stressful moments happened. This heart variation happened while he/she answered the third task, which had the program with all features disabled. In Figure 11 we have the scan path image and the attention map during the stressful moment. Figure 11 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 stressful moments happened when the participant perceived that he or she was analyzing a piece of code that contained a disabled feature. Analyzing programs

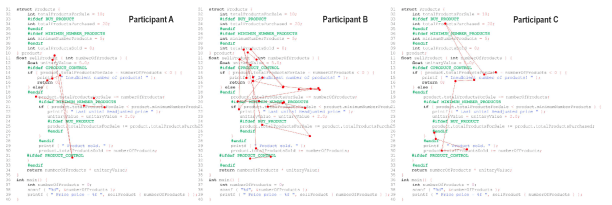


(a) Individual scan path of participants from program 3 (less variability).

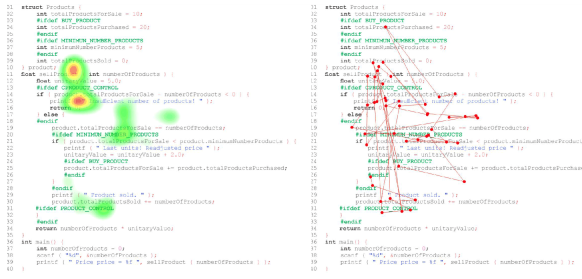


(b) Gaze movements diagram of Program 3 (less variability).

Figure 12. Gaze movements of Program 3 in the moment of stressful moments



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

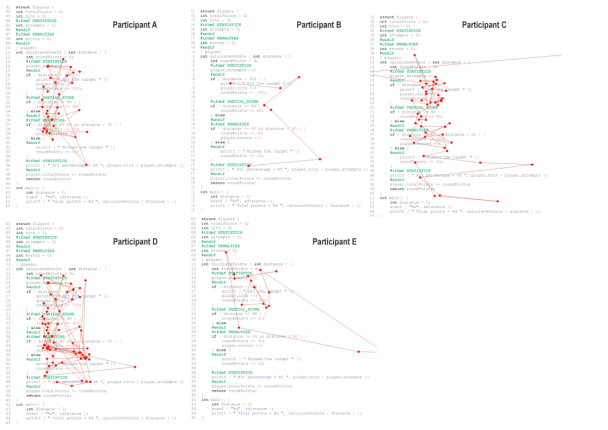


(b) Gaze movements diagram of Program 2 (more variability).

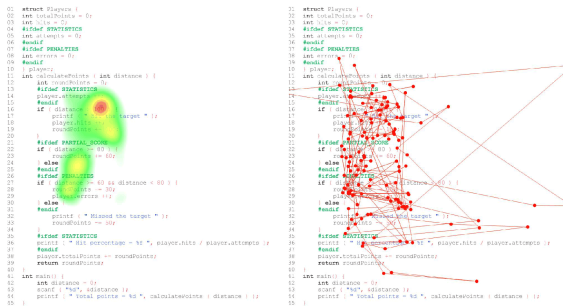
Figure 13. Gaze movements of Program 2 in the moment of stressful moments

with disabled features is not a trivial task. The participant concentrated their gaze movements, during the stressful moments, in a disabled feature.

Figure 12a represents scan paths of participants that analyzed Program 3, another program with less variability. Four



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



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

Figure 14. Gaze movements of Program 4 in the moment of stressful moments

participants registered stressful moments. Similarly to what we found in Study 1 (Section 2.5), most of the moments that participants had a stressful moment they were executing long gaze transitions. In this case, Figure 12a shows participants B, C, and D executing long gaze transitions. Two of the four participants had stressful moments while they were answering the third task, which has all features disabled. Figure 12b 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 13a 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 13b we had the overlapping of the four individual scan paths and attention maps that occurred during stressful moments. Figure 13b 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 14a and 14b. Again, most of participants were answering the third task, which has all features disabled while having a stressful moment.

In summary, participants seem to have performed long gaze transitions during stressful moments 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 stressful moments occurrence. However, it is important to highlight again that these findings were based on limited observations and information.

3.10 Threats to Validity

Study 1 and 2 procedures were the same, described in section 2 and 3. The programs and tasks were also similar, thus, the threats to validity of these studies are the same. In addition, Study 2 includes further threats to validity, which are described below.

3.10.1 Internal validity

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.

Fatigue. All participants began Study 2 immediately after completing Study 1, meaning any potential fatigue would have affected all participants equally. In Study 1, each participant performed two tasks, with an average duration of 1.5 minutes per task and a 2-minute break between them. These short sessions were designed to minimize the risk of fatigue. Additionally, in Study 2, we counterbalanced the order of tasks to further reduce the potential effects of fatigue.

Learning. Participants may have gained experience while performing the tasks in Study 1, potentially making the tasks in Study 2 easier to complete. To mitigate this effect, we selected participants who had prior experience with `#ifdef`. Additionally, we implemented a 2-minute distraction task between activities, where participants were instructed to watch a video of fish in an aquarium. This task served to help participants relax and allowed us to record their baseline heart rate, while also minimizing the immediate transfer of learning between tasks. Additionally, we counterbalanced the tasks in Study 2, varying their order among participants to further reduce learning effects.

4 Related Work

Previous research has indicated that `#ifdef` is considered harmful to the comprehensibility of configurable system source code (Spencer and Collyer, 1992; Feigenspan et al., 2013; Medeiros et al., 2017). In this context, numerous studies have focused on identifying possible causes and effects of this relation (Santos and Sant'Anna, 2019; Fenske et al., 2020).

A group of researchers investigated the types of errors and bugs in the source code of configurable systems. Medeiros *et al.* found and identified syntax errors in releases and commits of configurable systems (Medeiros et al., 2013). 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 variability bugs, i.e. bugs caused by the use of `#ifdefs` (Abal et al., 2014).

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., 2017). 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` (da Costa et al., 2021).

Other previous studies performed empirical studies related to comprehensibility and maintainability of configurable systems. Melo *et al.* used an eye-tracking device to evaluate the impact of `#ifdefs` in the comprehensibility of configurable systems (Melo et al., 2017). Another previous study showed that feature dependencies impacted the comprehensibility of programs with `#ifdefs` (Santos and Sant'Anna, 2019). Additionally, they showed that different types of feature dependencies may impact comprehensibility in different degrees (Santos and Sant'Anna, 2019). 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 et al., 2017).

Despite providing important contributions, none of these studies analyzed their data taking dependent variables into account. This is the main difference from our study, which explicitly analyzed in detail how the number of dependent variables affects the comprehensibility of configurable systems.

5 Conclusion

Here we answer our research questions.

RQ1 - How do different numbers of dependent variables affect the comprehensibility of configurable system source code?

Programs with more dependent variables were more difficult to understand. Our results 1 and 3 show that programs with more dependent variables demanded more comprehension effort from participants. Developers spent more time and more fixations when analyzing the programs with four dependent variables. We hypothesize 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.

In practice, our results indicate that comprehensibility is more negatively affected when the program increases the

number of dependent variables and when these variables are defined at a point far from the points where they are used. We encourage researchers and practitioners, whenever possible, to use fewer dependent variables in their code. For future work, we suggest replicating this study with more participants and including real environments and source codes.

RQ2 - 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 4 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 1 (Sections 2) in which comprehensibility was hindered 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 task was debugging, and the programs were not implemented with `#ifdef`.

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 confirms previous studies that showed that most participants answer the tasks correctly in programs with `#ifdef` (Melo et al., 2016, 2017; Santos and Sant'Anna, 2019).

Based on our findings and observations during the experiments, we see potential strategies to address feature dependencies in configurable systems that may impact research and practice and, thus, deserve to be further investigated.

From a research perspective, complexity metrics based on the quantity of dependent variables should be defined and studied. Furthermore, these metrics can be integrated into software development tools to help developers identify functions or features with a high number of dependent variables. Such tools could highlight sections of code that require careful testing, including test cases designed to examine how different features interact with or change the contents of dependent variables. This identification could help developers prioritize their efforts and improve system reliability and maintainability. Tools or IDE plugins may also provide developers with visualizations of feature dependencies and dependence variables without the need for excessive code navigation.

Developers can also try to minimize the number of dependent variables in their code. This can involve refactoring source code to reduce unnecessary dependencies or define dependent variables closer to their usage points. Such practices may reduce the cognitive load required to understand the dependent variables. By combining those strategies with further empirical studies, researchers and practitioners can improve their understanding about how useful it is to be aware of the existence of feature dependencies and dependent variables in configurable systems.

In summary, we concluded that feature dependency may affect the comprehensibility of configurable system source code when the source code contains a high number of dependent variables. We hypothesize that this happens because developers need to direct more attention to dependent variables. 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 from 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.

Acknowledgements

This study was supported by the Brazilian National Council for Scientific and Technological Development (CNPq) and Foundation to the state of Bahia (FAPESB).

References

Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 variability bugs in the linux kernel: a qualitative analysis. In *29th ACM/IEEE international conference on Automated software engineering*.

Bailey, R. A. (2008). *Design of comparative experiments*, volume 25. Cambridge University Press.

Baniassad, E. and Murphy, G. (1998). Conceptual module querying for software reengineering. In *20th International Conference on Software Engineering*.

Camilli, G. and Hopkins, K. D. (1978). Applicability of chi-square to 2×2 contingency tables with small expected cell frequencies. *Psychological Bulletin*.

da Costa, J. A. S., Gheyi, R., Ribeiro, M., Apel, S., Alves, V., Fonseca, B., Medeiros, F., and Garcia, A. (2021). Evaluating refactorings for disciplining# ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*.

Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., and Saake, G. (2013). Do background colors improve program comprehension in the# ifdef hell? *Empirical Software Engineering*.

Fenske, W., Krüger, J., Kanyshkova, M., and Schulze, S. (2020). # ifdef directives and program comprehension: The dilemma between correctness and preference. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Fenske, W., Schulze, S., and Saake, G. (2017). How preprocessor annotations (do not) affect maintainability: a case study on change-proneness. *ACM SIGPLAN Notices*.

Fritz, T., Begel, A., Müller, S. C., Yigit-Elliott, S., and Züger, M. (2014). Using psycho-physiological measures to assess task difficulty in software development. In *36th International Conference on Software Engineering (ICSE)*.

Garvin, B. J. and Cohen, M. B. (2011). Feature interaction faults revisited: An exploratory study. In *22nd Interna-*

tional Symposium on Software Reliability Engineering (ISSRE).

Hijazi, H., Couceiro, R., Castelhano, J., De Carvalho, P., Castelo-Branco, M., and Madeira, H. (2021). Intelligent biofeedback augmented content comprehension (tellback). *IEEE Access*.

Lanza, M. and Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE)*.

Malaquias, R., Ribeiro, M., Bonifácio, R., Monteiro, E., Medeiros, F., Garcia, A., and Gheyi, R. (2017). The discipline of preprocessor-based annotations-does# ifdef tag n't# endif matter. In *25th International Conference on Program Comprehension (ICPC)*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*.

Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating preprocessor-based syntax errors. In *ACM SIGPLAN Notices*.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kästner, C., Ferreira, B., Carvalho, L., and Fonseca, B. (2017). Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*.

Medeiros, F., Ribeiro, M., Gheyi, R., Braz, L., Kästner, C., Apel, S., and Santos, K. (2020). An empirical study on configuration-related code weaknesses. In *XXXIV Brazilian Symposium on Software Engineering*.

Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., and Gheyi, R. (2015). An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. *ACM SIGPLAN Notices*.

Melo, J., Brabrand, C., and Wasowski, A. (2016). How does the degree of variability affect bug finding? In *38th International Conference on Software Engineering (ICSE)*.

Melo, J., Narcizo, F. B., Hansen, D. W., Brabrand, C., and Wasowski, A. (2017). Variability through the eyes of the programmer. In *25th International Conference on Program Comprehension (ICPC)*.

Müller, S. C. and Fritz, T. (2015). Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In *37th IEEE International Conference on Software Engineering (ICSE)*.

Nakagawa, T., Kamei, Y., Uwano, H., Monden, A., Matsumoto, K., and German, D. M. (2014). Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In *36th international conference on software engineering*.

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*.

Rayner, K. (2009). Eye movements and attention in reading, scene perception, and visual search. *The quarterly journal of experimental psychology*.

Ribeiro, M., Borba, P., and Kästner, C. (2014). Feature maintenance with emergent interfaces. In *36th International Conference on Software Engineering (ICSE)*.

Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent feature modularization. In *International conference companion on Object-oriented programming systems languages and applications companion*.

Rodrigues, I., Ribeiro, M., Medeiros, F., Borba, P., Fonseca, B., and Gheyi, R. (2016). Assessing fine-grained feature dependencies. *Information and Software Technology*.

Santos, D. and Sant'Anna, C. (2019). How does feature dependency affect configurable system comprehensibility? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*.

Santos, D., SantAnna, C., and Ribeiro, M. (2023). An experiment on how feature dependent variables affect configurable system comprehensibility. In *17th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 61–70.

Schulze, S., Liebig, J., Siegmund, J., and Apel, S. (2013). Does the discipline of preprocessor annotations matter? a controlled experiment. In *12th international conference on Generative programming: concepts & experiences*. Association for Computing Machinery.

Spencer, H. and Collyer, G. (1992). # ifdef considered harmful, or portability experience with c news. *Usenix Summer 1992 Technical Conf.*, pages 185–197.

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., and Lohmann, D. (2014). Static analysis of variability in system software: The 90,000# ifdefs issue. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*.

Voßkühler, A., Nordmeier, V., Kuchinke, L., and Jacobs, A. M. (2008). Ogama (open gaze and mouse analyzer): open-source software designed to analyze eye and mouse movements in slideshow study designs. *Behavior research methods*.

Walter, G. F. and Porges, S. W. (1976). 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*.