UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

PÓS-GRADUAÇÃO EM INFORMÁTICA

ROMERO BEZERRA DE SOUZA MALAQUIAS

A DISCIPLINARIDADE DE ANOTAÇÕES CONDICIONAIS DE PRÉ-PROCESSAMENTO #IFDEF TAG NÃO #ENDIF IMPORTA

MACEIÓ, AL

OUTUBRO - 2017

ROMERO BEZERRA DE SOUZA MALAQUIAS

A DISCIPLINARIDADE DE ANOTAÇÕES CONDICIONAIS DE PRÉ-PROCESSAMENTO #IFDEF TAG NÃO #ENDIF IMPORTA

Dissertação apresentada ao programa de Pós-Graduação em Informática da Universidade Federal de Alagoas, como requisito parcial para obtenção do grau de mestre em Informática.

Orientador: Profº Dr. Márcio de Medeiros Ribeiro

Maceió, Al

Outubro - 2017

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Helena Cristina Pimentel do Vale

M237d Malaquias, Romero Bezerra de Souza.
A disciplinaridade das anotações condicionais de pré-processamento # ifdef TAG Não #endif importa / Romero Bezerra de Souza Malaquias. – 2018.
79 f. : il.

Orientador: Márcio de Medeiros Ribeiro.
Dissertação (mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Programa de Pós-Graduação em Informática Maceió, 2017.

Bibliografia: f. 74-79.

1. Engenharia de software. 2. Disciplinaridade das anotações. 3. Ifdefs. 4. Condicionais de compilação de códigos. 5. Compiladores (Programas de computador). I. Título.

CDU: 004.41

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PpgI
Instituto de Computação
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Membros da Comissão Julgadora da Dissertação de Romero Bezerra de Souza Malaquias, intitulada: “A Disciplinaridade de Anotações Condicionais de Pré-Processamento #ifdef TAG Não #endif Importa”, apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 09 de outubro de 2017, às 14h, na Sala de Reuniões, do Instituto de Computação da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Orientador

Prof. Dr. Ig Ibert Bittencourt Santana Pinto
UFAL – Instituto de Computação
Examinador

Prof. Dr. Rohit Gheyi
Universidade Federal de Campina Grande
Examinador

Agradecimentos

Agradeço a minha família pelo apoio a todos os meus sonhos e projetos, por estarem presentes e me concederem todo o suporte que eu preciso para os meus objetivos.

A minha namorada Sarah Batinga, pelo apoio e por me acompanhar nessa jornada.

Ao meu orientador, Márcio de Medeiros Ribeiro, pela dedicação, conselhos e apoio à minha formação. Sem sua orientação, este trabalho não teria sido possível. Obrigado por sua paciência e encorajamento durante todo o percurso.

Aos orientadores que a pesquisa me trouxe, em especial, ao Professor Alessandro Garcia e ao Professor Rodrigo Bonifácio, os quais, forneceram embasamento e conselhos durante a execução do experimento, além da hospitalidade com a qual me receberam em suas Instituições.

À todos os colegas professores Rodrigo Paes, Rohit Gheyi, Ig Bittencurt, Baldoino Fonseca que , de alguma forma, incrementaram conhecimento a minha pesquisa.

Aos amigos de Maceió que me apoiaram nessa jornada (Marcio Augusto, Fernanda Andrade, Regina Cansanção, dentre outros), os amigos do Rio de Janeiro (Davy Baía, Rafael "Rafalsky"Henrique, Andrey Rodrigues, Bruno Cafeo, ...) e os de Brasília (Rafael Vespasiano, Leomar Camargo, ...).

Às pessoas que direta e indiretamente contribuíram para a realização deste trabalho, peço desculpas àqueles que os nomes não estão aqui.

Ao pessoal do EASY pelo apoio e discussões calorosas acerca dos direcionamentos do trabalho.

Aos docentes, funcionários e discentes da Universidade Federal de Alagoas - UFAL, Pontifícia Universidade Católica do Rio de Janeiro - PUC/RJ e da Universidade de Brasília - UnB.

E por fim, à CAPES pelo apoio e suporte financeiro fornecidos a este trabalho.

Resumo

O pré-processador do C é uma ferramenta simples, efetiva e independente de linguagem. Na prática o pré-processador é visto como uma solução para problemas de portabilidade e variabilidade. Para utilizá-lo, basta anotar trechos do código com diretivas condicionais. Estas definem se aquele trecho de código será incluído ou excluído. Apesar de ser largamente utilizado, o pré-processador é amplamente criticado. Seu uso é relacionado a impactos negativos na manutenibilidade e legibilidade do código. Em particular, esses problemas podem ser piores quando se utiliza anotações não disciplinadas. Contudo, um experimento controlado realizado anteriormente indicou que a disciplinaridade não afeta a manutenibilidade e a legibilidade do código. Esse resultado contradiz as críticas recebidas e alguns guias para desenvolvimento de código (como por exemplo as do Linux, que explicitamente pedem para que os desenvolvedores evitem anotações não disciplinadas). Um exemplo de anotação não disciplinada ocorre quando são anotadas apenas partes de unidades sintáticas do C. Dado este cenário, onde há uma divergência entre a prática e a teoria, existe a necessidade de se compreender melhor a importância dada por desenvolvedores à disciplinaridade das anotações condicionais e se ela de fato causa algum impacto no desenvolvimento de sistemas. Nesse trabalho conduziremos um método misto de pesquisa que envolve três estudos. O primeiro consiste em sugerir transformações de código não disciplinado em disciplinado para sistemas *open source*. Nesse contexto, identificamos 110 anotações não disciplinadas em sistemas C/C++ de código aberto no *GitHub*. Estes sistemas têm diferentes domínios, tamanhos e popularidades em métricas do *GitHub*. A partir disso, nós refatoramos as anotações identificadas para que se tornassem disciplinadas. Por fim, submetemos nossas sugestões através de *pull requests*. A maioria das refatorações foram aceitas e agora integram o código do sistemas. Foi conseguida uma taxa de aceitação de 71%. O segundo estudo consiste em minerar repositórios *open-source* buscando *commits* que contenham transformações de anotações não disciplinadas em disciplinadas. Foram encontradas 90 transformações. Estas, foram classificadas em refatoração ou não. Adicionalmente, entramos em contato com o desenvolvedor a fim de entender o que o motivou a realizar tal alteração. No terceiro estudo, foi conduzido um experimento controlado. Este tem várias diferenças em relação ao executado anteriormente, como o bloqueio a algumas variáveis de confusão e mais réplicas. De acordo com os resultados, temos evidências de que realizar manutenção em código com anotações não disciplinadas é uma tarefa com um custo maior de tempo e é mais propensa a erros que as mesmas tarefas na forma disciplinada. Essas evidências se contrapõem ao resultado encontrado no experimento anterior. De forma geral, nesse trabalho, foi concluído que a disciplinaridade das anotações não deve ser negligenciada.

Palavras-chave: Disciplinaridade das anotações; Ifdefs; Compilação condicional.

Abstract

The C preprocessor is a simple, effective, and language-independent tool. Developers use the preprocessor in practice to deal with portability and variability issues. Despite the widespread usage, the C preprocessor suffers from severe criticism, such as negative effects on code understandability and maintainability. In particular, these problems may get worse when using undisciplined annotations, i.e., when a preprocessor directive encompasses only parts of C syntactical units. Nevertheless, despite the criticism and guidelines found in systems like Linux to avoid undisciplined annotations, the results of a previous controlled experiment indicated that the discipline of annotations has no influence on program comprehension and maintenance. To better understand whether developers care about the discipline of preprocessor-based annotations and whether they can really influence on maintenance tasks, in this work we conduct a mixed-method research involving three studies. In the first one we identify undisciplined annotations in 110 open source C/C++ systems of different domains, sizes, and popularity GitHub metrics. We then refactor the identified undisciplined annotations to make them disciplined. Right away, we submit pull requests with our suggestions. The majority of our pull requests have been accepted and are now merged. We reach a total acceptance rate of 71%. In the second study, we mined open-source repositories to find commits where the developer transforms undisciplined codes into disciplined ones. In that context, we found 90 transformations. Those transformations were classified in refactoring or not, and we contacted the responsible to ask why he did that code change. In the third study, we conduct a controlled experiment. We have several differences with respect to the aforementioned one, such as blocking of cofounding effects and more replicas. We have evidences that maintaining undisciplined annotations is more time consuming and error prone, representing a different result when compared to the previous experiment. Overall, we conclude that undisciplined annotations should not be neglected.

Keywords: Discipline of annotations; Ifdefs; Conditional Compilation.

Lista de ilustrações

Figura 1 – Exemplo de um código na sua forma não disciplinada (acima) e disciplinada (abaixo). Código extraído do *VIM*. 10
Figura 2 – Exemplo de portabilidade para diferentes Sistemas Operacionais. Código extraído do *VIM*. 16
Figura 3 – Exemplo de um código com anotações condicionais que pode causar um vazamento de memória. Exemplo extraído de Medeiros et. al. [1]. . . . 17
Figura 4 – Exemplo de um código em sua forma não disciplinada (acima) e disciplinada (abaixo). Código extraído do *Libelektra*. 17
Figura 5 – Desenvolvedor do *Nginx* evitando o uso de anotações não disciplinadas. 18
Figura 6 – Disciplinando anotações com duplicação de código. 21
Figura 7 – Nossa estratégia para a criação de pull requests. 22
Figura 8 – Exemplos de *pull requests* submetidos. Abaixo de cada transformação, nós ilustramos as mensagens que recebemos dos desenvolvedores. . . . 23
Figura 9 – Resultados dos nossos *pull requests*. 24
Figura 10 – *Word cloud* feita a partir dos comentários obtidos nos *pull requests*. . . 25
Figura 11 – Processo de identificação de transformações e contato com desenvolvedores. 29
Figura 12 – Exemplo de refatoração com correção de erro (não perfectiva). 30
Figura 13 – Exemplo de refatoração visando a melhoria da qualidade do código (perfectiva). 31
Figura 14 – Gráfico representando a classificação das transformações encontradas. . 32
Figura 15 – Exemplo de refatoração com correção de erro (não perfectiva). 33
Figura 16 – Projeto do experimento utilizando quadrados latinos. 39
Figura 17 – Estrutura da distribuição de tarefas em termos de unidades de experimento. 40
Figura 18 – Trecho de código do *VIM* que inspirou a tarefa de corrigir um erro de sintaxe. 41
Figura 19 – Trecho de código do *Linux Kernel* que inspirou a tarefa de adicionar nova funcionalidade. 42
Figura 20 – Trecho de código do *Linux Kernel* que inspirou a tarefa de corrigir um erro de comportamento. 43
Figura 21 – *Plugin* para Eclipse. Note os botões do nosso *plugin* na esquerda (topo). 44
Figura 22 – Tempo necessário para concluir tarefas do Tipo 1. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 45
Figura 23 – Gráfico de densidade de tempo necessário para tarefas do Tipo 1. . . . 46
Figura 24 – Tentativas necessárias para concluir tarefas do Tipo 1. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 47

Figura 25 – Gráfico de tempo necessário para concluir tarefas do Tipo 2. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 48
Figura 26 – Gráfico de densidade de tempo necessário para tarefas do Tipo 2. . . . 49
Figura 27 – Tentativas necessárias para concluir tarefas do Tipo 2. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 50
Figura 28 – Gráfico de quantidade de tempo necessário para conclusão de tarefas do Tipo 3. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 51
Figura 29 – Gráfico de densidade de tempo necessário para tarefas do Tipo 3. . . . 52
Figura 30 – Tentativas necessárias para concluir tarefas do Tipo 3. Onde “D” é relativo a disciplinado e “U” a não disciplinado. 53
Figura 31 – Tempo total para concluir todas as tarefas. 53
Figura 32 – Gráfico de densidade considerando a diferença de tempo entre $(T_{nd} - T_d)$, onde T_{nd} é o tempo gasto por um participante para resolver tarefas na forma não disciplinada e T_d é o tempo gasto pelo mesmo participante para responder na forma disciplinada. 54
Figura 33 – Número de tentativas para concluir todas as tarefas. 55
Figura 34 – Exemplo de transformação de um código não disciplinado (esquerda) em um disciplinado (direita) feito pelo P-Cpp. Exemplo extraído de Garrido et al [2] 63
Figura 35 – Exemplo de transformação de um código não disciplinado (esquerda) em um disciplinado (direita) sugerido pelo catálogo de refatorações. . . 63
Figura 36 – Versão disciplinada (esquerda) e não disciplinada (direita) para a tarefa de corrigir um erro de sintaxe no *Libxml2*. 70
Figura 37 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de adicionar nova funcionalidade no *Libxml2*. 71
Figura 38 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de corrigir um erro de comportamento no *Libxml2*. 71
Figura 39 – Versão disciplinada (esquerda) e não disciplinada (direita) para a tarefa de corrigir um erro de sintaxe no *VIM*. 72
Figura 40 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de adicionar nova funcionalidade no *VIM*. 72
Figura 41 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de corrigir um erro de comportamento no *VIM*. 73

Sumário

1 INTRODUÇÃO 10
1.1 Definição dos Estudos Realizados 11
1.2 Contribuições 14
1.3 Organização 14

2 EMBASAMENTO TEÓRICO 15
2.1 Pré-processador do C 15
2.2 Disciplinado x Não Disciplinado 16
2.3 Sistemas Configuráveis 19

3 CONTATO COM OS DESENVOLVEDORES: SUGESTÕES PARA DISCIPLINAR ANOTAÇÕES 20
3.1 Configuração 20
3.2 Resultados e Discussão 22
3.3 Ameaças à Validade 26

4 CONTATO COM OS DESENVOLVEDORES: MINERANDO REPOSITÓRIOS 28
4.1 Configuração 28
4.2 Resultados e Discussão 30
4.3 Ameaças à Validade 34

5 EXPERIMENTO CONTROLADO 36
5.1 Configuração 36
5.2 Unidades Experimentais 39
5.3 Execução e Procedimentos de Análise de Dados 41
5.4 Resultados e Discussão 44
5.5 Primeira análise: Análise por tipo de tarefa 45
5.5.1 Tipo 1: Corrigir erro de sintaxe 45
5.5.2 Tipo 2: Adicionar uma nova funcionalidade 47
5.5.3 Tipo 3: Corrigir um erro de comportamento 49
5.6 Segunda análise: Análise de um dia de trabalho. 51
5.7 Ameaças à Validade 55

6 TRABALHOS RELACIONADOS 57
6.1 Pré-processador do C 57
6.2 Disciplinaridade das Anotações 59

6.3 Refatoração em Código com Anotações Condicionais 62

7 CONCLUSÃO 64
7.1 Revisão das Contribuições 65
7.2 Implicações 66
7.3 Trabalhos Futuros 66

A APÊNDICE 68
A.1 Lista de Sistemas que Foram Submetidos *Pull Requests* 68
A.2 Tarefas do Experimento 70

REFERÊNCIAS 74

1 Introdução

O pré-processador do C é uma ferramenta amplamente utilizada para implementar variabilidade no código e resolver problemas de portabilidade em sistemas configuráveis [3–11]. Contudo, seu uso está relacionado a um aumento na complexidade do código [12–14]. Para utilizar o pré-processador os desenvolvedores anotam o código usando diretivas como **`#ifdef`**, **`#elif`**, **`#else`**, e **`#endif`**. Estudos existentes [15–17] classificam o uso dessas anotações como disciplinado e não disciplinado. O uso não disciplinado ocorre quando a diretiva não envolve inteiramente a subárvore da árvore sintática abstrata correspondente. Nesse sentido, um exemplo acontece quando se anota somente parte da declaração de uma variável. Na Figura 1 tem-se um exemplo de código extraído do *VIM*.[1] O código encontra-se anotado na forma não disciplinada (acima) e disciplinada (abaixo). Note que o código acima não é disciplinado porque somente uma parte da expressão está anotada com **`#ifdef`**. Em outras palavras, a declaração da variável está incompleta. Já no código abaixo da Figura 1, temos que as anotações envolvem completamente a declaração da variável `prev_ptr` completamente, sendo chamada de uma anotação disciplinada.

```
char_u * prev_ptr = ptr - (
#ifdef FEAT_MBYTE
   has_mdyte ? mb_1 :
#endif
1);
```

```
#ifdef FEAT_MBYTE
    char_u * prev_ptr = ptr - (has_mdyte ? mb_1 : 1);
#else
    char_u * prev_ptr = ptr - (1);
#endif
```

Figura 1 – Exemplo de um código na sua forma não disciplinada (acima) e disciplinada (abaixo). Código extraído do *VIM*.

Em um estudo anterior [17], foi conduzido o primeiro experimento controlado para testar a hipótese de que *a disciplinaridade das anotações influencia na compreensão e manutenção do código anotado.* Porém, seus resultados não sustentaram a hipótese [17]. Concluiu-se que a disciplinaridade *não* influencia nos quesitos de compreensibilidade e manutenibilidade nas tarefas realizadas no experimento. Essa conclusão é baseada em medidas de tempo e de corretude da resposta para conclusão das tarefas. As tarefas consistiam em rotinas de manutenção e compreensão de código com anotações condicionais. Por corretude, uma resposta poderia ser considerada como “correta”, “quase correta” ou

[1] `https://github.com/vim/vim`

"errada". Esse resultado contradiz as críticas que o uso de anotações não disciplinadas vem recebendo. Seu uso é associado a uma diminuição da legibilidade, manutenibilidade e compreensibilidade [4, 18–24]. Somadas a essas críticas, existem diretrizes de codificação que pedem que os desenvolvedores evitem o uso de anotações não disciplinadas. Um exemplo pode ser visto nas diretrizes do *Kernel do Linux*, que diz: "*Dê preferência a compilar funções inteiras, ao invés de partes de funções ou expressões.*"[2]

Diante deste cenário, onde a prática e a teoria parecem se contradizer, nosso trabalho tem por objetivo verificar se esses dois mundos realmente divergem. Nós buscamos entender a relação da disciplinaridade das anotações condicionais com o desenvolvimento e o programador de código. Em outras palavras, tencionamos entender se os desenvolvedores preocupam-se com a disciplinaridade das anotações de pré-processamento e se a disciplinaridade destas realmente impactam na realização das tarefas de manutenção e compreensão do código. Nossa pesquisa é inspirada em trabalhos anteriores que utilizam repositórios de código, como o *GitHub*, para entrar em contato com os desenvolvedores [5, 16, 25] e também pelo experimento mencionado anteriormente [17]. Acreditamos que, ao ampliar o entendimento do impacto da disciplinaridade de anotações de pré-processamento no desenvolvimento de código, será possível guiar o surgimento de novas ferramentas para auxiliar os desenvolvedores na remoção de anotações não disciplinadas e aprimorar os guias de boas práticas de programação.

1.1 Definição dos Estudos Realizados

Esse estudo tenciona entender melhor o relacionamento entre desenvolvedor e disciplinaridade do código. Isso implica em entender se o desenvolvedor prefere código disciplinado (1), se ele disciplina o código (2) e se o uso de código disciplinado traz benefícios em relação ao não disciplinado (3). Para alcançar esse objetivo será conduzido um método de pesquisa misto [26] que consiste em três estudos. A Tabela 1 traz um sumário das hipóteses, abordagens e questões de pesquisa para cada estudo.

O primeiro estudo tem como questão de pesquisa: *Os desenvolvedores aceitam sugestões para a remoção de anotações não disciplinadas?* Para responder essa questão nós encontramos anotações não disciplinadas em 110 sistemas de código aberto C/C++ de diferentes domínios, tamanhos e popularidades em repositórios no *GitHub*. A partir disso, refatoramos manualmente o código, transformando as anotações em disciplinadas. Apenas uma anotação por projeto foi refatorada. Em seguida foram submetidos *pull requests* com essas sugestões. Adicionalmente, o *GitHub* também foi utilizado para a submissão de comentários e perguntas visando entender o que os desenvolvedores pensam sobre a alteração vista no *pull request*. Nós acreditamos que a taxa de aceitação dos *pull-requests*

[2] https://01.org/linuxgraphics/gfx-docs/drm/process/coding-style.html#conditional-compilation

– Estudo 1 –	– Estudo 2 –
Q1: Os desenvolvedores preferem código disciplinado?	**Q2**: Os desenvolvedores disciplinam anotações condicionais? Como?
A1 Sugerir *pull-requests* disciplinando o código.	**A2** Minerar repositórios em busca de *commits* onde essas refatorações ocorreram e entrar em contato com os desenvolvedores responsáveis.
H1: Teremos uma taxa de aceitação significativa, indicando que os desenvolvedores se importam com a disciplinaridade das anotações condicionais.	**H2**: Iremos encontrar *commits* que transformam código não disciplinado em código disciplinado.

– Estudo 3 –
Q3: A disciplinaridade das anotações afeta a realização de tarefas de manutenção de código nos quesitos de tempo e propensão a erros?
A3 Realizaremos um experimento controlado com alunos de cursos de graduação a fim de medir o impacto da disciplinaridade.
H3: As tarefas não disciplinadas vão consumir mais tempo e mais tentativas para serem concluídas.

Tabela 1 – Visão geral das questões de pesquisa, abordagens utilizadas e hipóteses.

pode ser utilizada como um indicador capaz de inferir que os desenvolvedores se importam com a disciplinaridade. Adicionalmente fizemos uma análise qualitativa das respostas. Para isso classificamos as respostas em categorias e criamos uma *word cloud* com elas.

No segundo estudo, utilizamos as seguintes questões de pesquisa: *(1) Os desenvolvedores disciplinam o código? (2) Qual a motivação para realizar tais refatorações? (3) Como ocorrem tais refatorações?* Para responder essas perguntas, nós mineramos repositórios de código *open-source* em busca de *commits* onde o desenvolvedor disciplinou uma anotação não disciplinada. Os *commits* encontrados foram classificados em refatoração ou não. Um *commit* considerado refatoração é uma alteração de código que não altera o comportamento do mesmo [27]. Com isso esperamos ter indícios de que o processo de disciplinar não ocorre de forma oportuna. Por oportuna, entende-se uma refatoração que foi feita em conjunto com uma correção. Para cada *commit* encontrado entramos em contato via *email* com o desenvolvedor a fim de descobrir sua motivação. Por fim, nós classificamos e geramos uma *word cloud* com as respostas obtidas. Acreditamos que essa abordagem também poderá nos dar indícios que os desenvolvedores se importam com a disciplinaridade do código a ponto de remover anotações não disciplinadas.

No terceiro estudo, focaremos nas seguintes questões de pesquisa: *(1) O uso de anotações não disciplinadas faz com que o tempo para resolver tarefas de manutenção aumente? (2) O uso de anotações não disciplinadas faz com que o desenvolvedor cometa mais erros ao resolver uma tarefa de manutenção?* Para responder essas perguntas, conduzimos um experimento controlado com 64 participantes (graduandos da Universidade de Brasília, UnB). Esse experimento utiliza tarefas de manutenção para testar as mesmas variáveis

dependentes do experimento anterior (tempo e corretude) [17]. Contudo, nosso experimento possui diferenças significativas em relação ao anterior [17]. Primeiramente, nosso experimento foi modelado a fim de (a) bloquear mais fontes de elementos de confusão (como experiência, envolvimento dos participantes e o conjunto de tarefas que eles devem resolver) e (b) aumentamos a quantidade de réplicas, que nos permite obter mais precisamente a estimativa do fator/efeito de interação [28]. Em segundo lugar, o experimento realizado anteriormente utilizou uma técnica baseada no trabalho de Liebig et. al [15] para fazer a versão disciplinada das suas tarefas. Essa técnica consiste em clonar código para disciplinar as anotações. Isso acarretou em uma grande diferença no número de linhas de código entre as tarefas com código disciplinado e não disciplinado. Alguns estudos correlacionam o tamanho do código a diferentes métricas [29–35] e sugerem que códigos maiores são mais propensos a erros. Para evitar esse possível ruído, em nosso experimento obtivemos um número de linhas de código bem próximo entre as tarefas, i.e., as versões disciplinadas das tarefas têm quase o mesmo número de linhas das não disciplinadas. Outra diferença é que não seguimos a noção de "quase correto" no nosso experimento. Somente consideramos as tarefas que foram corretamente concluídas. Todos os dados de participantes que não conseguiram concluir corretamente todas as tarefas foram descartados. Uma outra diferença é que nós não fazemos comparações entre tarefas que não são correspondentes. Por exemplo, nós não comparamos uma tarefa de corrigir um erro em um código disciplinado com uma tarefa de identificar um erro em um código não disciplinado. No nosso entendimento, essas tarefas são diferentes e, consequentemente, não devem ser comparadas. Por outro lado, nosso experimento tem apenas três tipos de tarefa de manutenção: corrigir um erro de sintaxe, adicionar uma nova funcionalidade e corrigir um erro de comportamento. O experimento anterior [17] tinha sete tipos diferentes de tarefas, dos quais escolhemos três tipos para utilizarmos no nosso experimento. Estes tipos serviram de base para criarmos nossas tarefas. Acreditamos que a quantidade de tentativas e o tempo necessário que o participante levou para realizar as tarefas serão importantes para melhor entender o impacto da disciplinaridade das anotações no desenvolvimento de sistemas.

Os resultados obtidos sugerem que a disciplinaridade faz diferença na prática. Em relação ao primeiro estudo, foram submetidos 110 *pull requests*, onde 99 foram respondidos. Nestes, quase dois terços foram aceitos. Em relação aos rejeitados, segundo o *feedback* dos desenvolvedores, alguns poderiam ter sido aceitos se fossem submetidos em outros trechos de código. Aconteceu que, por falta de experiência com os projetos, acabamos sugerindo alterações em trechos de código depreciados ou em bibliotecas de terceiros. Para estas rejeições, perguntamos se os desenvolvedores aceitariam os *pull requests* caso tivéssemos submetido as sugestões para um código que não fosse depreciado ou de um terceiro. Considerando que alguns desenvolvedores responderam positivamente (aceitariam), alcançamos uma taxa de aceitação de 71%. Já em relação à abordagem de minerar os repositórios a fim de encontrar transformações ou remoções de anotações não disciplinadas, foram encon-

tradas 90 transformações. Apenas 19 desenvolvedores, responsáveis pelas transformações, responderam à nossa entrevista, onde 52% afirmou ter realizado essa transformação para melhorar a qualidade do código. A respeito do experimento controlado, nossas observações trazem evidências de que realizar manutenção em código não disciplinado é uma tarefa mais susceptível a erros e que consome mais tempo do que em código disciplinado. Esses resultados contrariam os do experimento anterior [17].

1.2 Contribuições

As principais contribuições deste trabalho são:

- Um *quasi-experimento* baseado em *pull requests* para um melhor entendimento sobre como os desenvolvedores se relacionam com a disciplinaridade das anotações condicionais;
- Um estudo empírico baseado em minerar *commits* para um melhor entendimento sobre se os desenvolvedores refatoram as anotações condicionais;
- Um experimento controlado, que investiga o efeito da disciplinaridade das anotações condicionais, em termos de tempo de resposta e corretude em tarefas de manutenção.

1.3 Organização

No Capítulo 2 iremos introduzir o embasamento teórico relacionado ao uso do pré-processador do C. No Capítulo 3 iremos apresentar o estudo relativo ao contato com os desenvolvedores via sugestões para disciplinar anotações. Será descrita a configuração, os resultados, uma discussão sobre os resultados e as ameaças à validade do estudo. No Capítulo 4 iremos apresentar o estudo relativo à mineração de repositórios. Será descrita a configuração, os resultados, uma discussão sobre os resultados e as ameaças à validade do estudo. No Capítulo 5 iremos apresentar o experimento que realizamos. Esse experimento consiste em um quadrado latino para verificar o impacto da disciplinaridade das anotações. Será descrita a configuração, as unidades experimentais, a execução e procedimentos de análise de dados, os resultados, uma discussão sobre os resultados e por fim as ameaças à validade. No Capítulo 6 iremos apresentar os trabalhos relacionados. No Capítulo 7 serão apresentadas as considerações finais deste trabalho.

2 Embasamento Teórico

Neste capítulo iremos introduzir alguns termos utilizados no estudo. Na Seção 2.1 focaremos em explicar o pré-processador do C. Na Seção 2.2 mostraremos duas formas de anotar código com o pré-processador C: a forma disciplinada e a forma não disciplinada. Por fim, na Seção 2.3 iremos introduzir o conceito de sistemas configuráveis.

2.1 Pré-processador do C

O pré-processador do C é uma ferramenta simples e efetiva. Essa ferramenta é amplamente utilizada por desenvolvedores para tratar problemas de variabilidade e portabilidade. Ela permite realizar processamento de texto em códigos fonte. Com isso é possível criar variações no código. Para criar essas variações, ou variantes, o desenvolvedor precisa anotar o código com diretivas condicionais **`#ifdef`**, **`#elif`**, **`#else`**, e **`#endif`**. Quando a condição de uma diretiva é atendida, o pré-processador irá adicionar aquele trecho de código anotado ao código final. O conjunto que define quais as diretivas que terão suas condições atendidas é chamado de configuração. Essencialmente o pré-processador é utilizado em projetos C/C++, incluindo vários servidores web, bancos de dados e sistemas operacionais conhecidos [5, 15, 17]. Na Figura 2 é possível ver um exemplo de portabilidade, onde o código de um editor de texto (*VIM*)[1]' é reutilizado para diferentes sistemas operacionais. Para gerar uma versão do *VIM* para um sistema operacional *UNIX*, por exemplo, a variável **`UNIX`** da Figura 2 precisa estar definida. Se ela estiver definida, as variáveis **`pid`**, **`st_dev`** e **`st_ino`** serão compiladas e estarão presentes no executável do sistema. Caso contrário, elas serão removidas completamente do processo de compilação. Caso a variável **`UNIX`** não esteja definida e a variável **`WIN32`** esteja definida, o código será portado para o sistema *Windows*. Nesse caso, as variáveis **`pid`** (do tipo **`DWORD`**), **`hproc`**, **`nVolume`**, **`nIndexHigh`** e **`nIndexLow`** serão consideradas. Assim sendo, o código tem variantes para *Unix* e *Windows*. Note que temos dois exemplos de configurações (uma **`UNIX`** e outra **`WIN32`**). Uma configuração pode ser definida através do uso de comandos **`#define`** que podem ser colocados no código. Por exemplo, `#define UNIX`. Outra forma de se escolher uma configuração é definir diretamente para o compilador.

Apesar do seu amplo uso, o pré-processador do C recebe várias críticas, como falta de separação das preocupações (*Separation of concerns*) [3, 4, 10, 15, 18, 19, 36, 37], susceptibilidade à introdução de erros (como erros de sintaxe, identificadores não declarados, vazamentos de memória, etc) [3, 15, 36, 37, 37–43] e obscurecimento do código [19, 44, 45]. Por exemplo, a Figura 3 apresenta um trecho de código retirado do F Virtual Widow Manager

[1] https://github.com/vim/vim

```c
typedef struct csi {
    char * fname;
    char * ppath;
    char * flags;
    #if defined(UNIX)
        pid_t pid;
        dev_t st_dev;
        ino_t st_ino;
    #else
        # if defined(WIN32)
            DWORD pid;
            HANDLE hProc;
            DWORD nVolume;
            DWORD nIndexHigh;
            DWORD nIndexLow;
        # endif
    #endif
    FILE * fr_fp;
    FILE * to_fp;
} csinfo_T;
```

Figura 2 – Exemplo de portabilidade para diferentes Sistemas Operacionais. Código extraído do *VIM*.

(FVWM).[2] Nesse exemplo o uso de anotações condicionais pode causar um vazamento de memória (*memory leak*). Se `RE_ENABLE_I18N` estiver definido e a variável `sbcset` for nula, a execução não irá alcançar a função que libera a memória, i.e., `re_free (mbcset)`, o que vai manter o espaço de memória alocado mas sem uso.

Todos esses problemas tornam as tarefas de manutenção e compreensão de código com diretivas de pré-processamento mais difíceis. [3, 46]. Os problemas citados na seção anterior pioram com o uso de anotações não disciplinadas [5].

2.2 Disciplinado x Não Disciplinado

Em relação a disciplina das anotações temos que: "*Anotações disciplinadas se alinham à estrutura do código fonte, elas tem como alvo fragmentos de código que pertencem a subárvores inteiras da árvore sintática abstrata correspondente*" [17]. A partir dessa definição, um exemplo de uma anotação disciplinada é um **`#ifdef`** contendo inteiramente uma declaração de um **`if`**. Em contraste, anotar somente parte da condição do comando **`if`** representa um exemplo de anotação não disciplinada, como pode ser visto na Figura 4. Essa figura ilustra trechos de código extraídos do *VIM*.[3] O código que se encontra na parte de cima tem a diretiva **`#if`** dividindo a expressão condicional do comando **`if`** em partes.

[2] http://fvwm.org/
[3] https://github.com/vim/vim

```
static bin_tree_t * parse_bracket_exp (){
   // code here..
  #ifdef RE_ENABLE_I18N
      mbcset = (re_charset_t *) calloc (sizeof (re_charset_t), 1);
   #endif

  #ifdef RE_ENABLE_I18N
      if (sbcset == NULL || mbcset == NULL)
   #else
      if (sbcset == NULL)
   #endif
   {
      *err = REG_ESPACE;
      return NULL;
   }
   // code here..
   #ifdef RE_ENABLE_I18N
       re_free (mbcset);
   #endif

   return NULL;
}
```

Figura 3 – Exemplo de um código com anotações condicionais que pode causar um vazamento de memória. Exemplo extraído de Medeiros et. al. [1].

Note que somente a parte `&& stat != yajl_status_insufficient_data` da condição está anotada com a diretiva `#if`. Essa divisão caracteriza uma anotação não disciplinada. Já no código que se encontra na parte inferior, a diretiva **`#if`** não divide a atribuição da variável. Como não houve divisão no comando, tem-se uma anotação disciplinada.

```
if (stat != yajl_status_ok
    #if YAJL_MAJOR == 1
        && stat != yajl_status_insufficient_data
    #endif
)
    {
         ...
     }
```

```
int test_status = (stat != yajl_status_ok);
#if YAJL_MAJOR == 1
    test_status = test_status && (stat != yajl_status_insufficient_data);
#endif
if (test_status)
    {
         ...
     }
```

Figura 4 – Exemplo de um código em sua forma não disciplinada (acima) e disciplinada (abaixo). Código extraído do *Libelektra*.

Outro indício que aponta o uso de anotações não disciplinadas como uma má prática pode ser visto em guias de desenvolvimento (*Code guidelines*). Esses guias representam manuais que devem ser seguidos pelos desenvolvedores do sistema e geralmente servem para pedir que os desenvolvedores evitem más práticas de implementação [5]. Em alguns desses guias, é possível encontrar afirmativas que pedem para que os desenvolvedores evitem o uso de anotações não disciplinadas, como pode ser visto no guia de desenvolvimento do Linux: "*Prefira compilar funções inteiras ao invés de partes de funções ou expressões. Ao invés de colocar um ifdef em uma expressão, fatore parte ou ela toda em uma função separada e aplique os condicionais àquela função.*"[4] Porém, mesmo estas regras podendo ser verificadas automaticamente [5, 15, 47], estudos anteriores [15, 19] mostram que elas nem sempre são seguidas.

Os guias de desenvolvimento não são os únicos a proporem formas de evitar o uso das anotações não disciplinadas. Estudos anteriores [1, 2, 15, 48] contêm diferentes abordagens de como disciplinar anotações. A Figura 5 ilustra um trecho de código do servidor *web Nginx Web Server*[5] onde o desenvolvedor disciplinou a anotação.[6] O lado esquerdo contém anotações não disciplinadas, pois apenas pedaços da condição do **if** estão anotados. Note que as anotações do lado direito contêm fragmentos de código de subárvores inteiras. Isto é, atribuições completas da variável **ready** estão englobadas pelas diretivas de pré-processamento.

```
#if (WIN32)
if ((ready = select(0, NULL, tp))
#else
if ((ready = select(max_fd + 1, NULL, tp))
#endif
        == -1) {
    ngx_log_error(NGX_LOG_ALERT);
    return NGX_ERROR;
    if (ready == -1) {
    err = ngx_socket_errno;
    } else {
    err = 0;
    }
}
```

```
#if (WIN32)
ready = select(0, NULL, tp);
#else
ready = select(max_fd + 1, NULL, tp);
#endif
if (ready == -1) {
  ngx_log_error(NGX_LOG_ALERT);
  return NGX_ERROR;
  if (ready == -1) {
    err = ngx_socket_errno;
  } else {
    err = 0;
  }
}
```

Figura 5 – Desenvolvedor do *Nginx* evitando o uso de anotações não disciplinadas.

O primeiro experimento controlado realizado para medir os efeitos da disciplinaridade no desenvolvimento do código concluiu que a disciplinaridade das anotações não tem efeito observável na legibilidade e manutenibilidade do código anotado [17]. Este resultado contradiz os obtidos em outros estudos que afirmam que anotações não disciplinadas são mais difíceis de ler e entender [4, 5, 18, 19]. Nesse trabalho nós tentaremos compreender

[4] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/coding-style.rst#n1004

[5] http://nginx.org

[6] https://github.com/git-mirror/nginx/commit/8292037f7c233f8351b4baf90c84656550cb12cf#diff-69dc91a92312b78403fe301f07b53992

melhor se os desenvolvedores se importam com a disciplinaridade e se ela realmente faz diferença em tarefas de manutenção de sistemas. Para isso faremos três estudos. O primeiro foca em verificar se os desenvolvedores aceitam sugestões para disciplinar o código anotado. O segundo busca entender se os desenvolvedores disciplinam código. Por fim, realizaremos um experimento controlado inspirado no realizado anteriormente [17].

2.3 Sistemas Configuráveis

Sistemas configuráveis são sistemas de *software* com funcionalidades que podem ser ativadas ou desativadas [49]. Um exemplo poder ser visto no *Windows*.[7] O *Windows* possui um mesmo código que pode ser adaptado (com diferentes funcionalidades) para diferentes versões (*Basic, Professional, dentre outras*). Sistemas configuráveis constituem uma importante classe de *software*. Exemplos de sistemas configuráveis vão de produtos personalizáveis pelo usuário, como o *Firefox*, *Eclipse IDE*, *Google Chrome*, até sistemas altamente configuráveis, como o *Linux*, *Apache*, *Postgreesql*, *Sqlite*,*Libxml2*, *Cherokee*, *VIM*, dentre outros. O uso de pré-processamento é a forma mais comum de implementar sistemas configuráveis.

[7] https://www.microsoft.com/pt-br/windows/

3 Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações

O primeiro estudo que iremos reportar nesse trabalho se refere à submissão de sugestões para disciplinar anotações através do uso de *pull requests* em sistemas *open source*. Nós supomos que a taxa de aceitação dos *pull requests* irá servir como indicador que esse tipo de refatoração é relevante para os desenvolvedores. A Tabela 2 resume a questão de pesquisa, abordagem e hipótese do estudo. Adicionalmente, iremos realizar uma análise qualitativa das respostas obtidas. A seguir, iremos apresentar as configurações desse estudo. Depois os resultados serão apresentados e discutidos.

– Estudo 1 –	– Estudo 2 –
Q1: Os desenvolvedores preferem código disciplinado?	**Q2**: Os desenvolvedores disciplinam anotações condicionais? Como?
A1 Sugerir *pull-requests* disciplinando o código.	**A2** Minerar repositórios em busca de *commits* onde essas refatorações ocorreram e entrar em contato com os desenvolvedores responsáveis.
H1: Teremos uma taxa de aceitação significativa, indicando que os desenvolvedores se importam com a disciplinaridade das anotações condicionais.	**H2**: Iremos encontrar *commits* que transformam código não disciplinado em código disciplinado.

– Estudo 3 –
Q3: A disciplinaridade das anotações afeta a realização de tarefas de manutenção de código nos quesitos de tempo e propensão a erros?
A3 Realizaremos um experimento controlado com alunos de cursos de graduação a fim de medir o impacto da disciplinaridade.
H3: As tarefas não disciplinadas vão consumir mais tempo e mais tentativas para serem concluídas.

Tabela 2 – Visão geral da questão de pesquisa, abordagem utilizada e hipótese para o Estudo 1.

3.1 Configuração

Nós utilizaremos os *pull requests* para responder a seguinte questão: *Os desenvolvedores aceitam sugestões para remover anotações não disciplinadas?* Responder essa pergunta é importante para verificar se os desenvolvedores se sentem mais confortáveis em trabalhar com anotações disciplinadas. Adicionalmente os comentários dos desenvolvedores, em relação aos *pull requests*, podem trazer novas perspectivas acerca dos benefícios e consequências do processo de disciplinar anotações de pré-processamento.

Antes de começar o trabalho de escolha dos projetos e alterar os códigos, nós precisávamos entender como disciplinar anotações. Nesse contexto estudos anteriores propõem formas de transformar anotações não disciplinadas em disciplinadas. Alguns desses estudos usam duplicação de código [2,15,48], o que poderia nos levar a um resultado indesejável. Acreditamos que a duplicação do código pode ofuscar os benefícios que o uso disciplinado oferece. A Figura 6 ilustra como essa transformação ocorre na prática. Na versão não disciplinada (código na parte de cima), as condições do comando `if` foram quebradas com as diretivas `#if` e `#else`. Note que, ao disciplinar (código na parte de baixo), o comando `if` foi duplicado.

```
if ((ready = select(
#if (WIN32)
0
#else
max_fd + 1
#endif
    , NULL, tp)) == -1) {
    ...
}
```

```
#if (WIN32)
if ((ready = select(0, NULL, tp)) == -1) {
    ...
}
#else
if ((ready = select(max_fd + 1, NULL, tp)) == -1) {
    ...
}
#endif
```

Figura 6 – Disciplinando anotações com duplicação de código.

Resolvemos optar por uma abordagem que não duplica código. Ela consiste em seguir o catálogo de refatorações publicado por Medeiros et. al [1,16]. O próximo passo foi selecionar quais projetos iriam ser refatorados. Para isso, escolhemos 110 projetos *open-source* que utilizam C/C++ como linguagem e com diferentes quantidades de linha de código e popularidade no *GitHub*. Esse número e variedade de projetos tenta evitar o ruído de não ter aleatoriedade na escolha dos mesmo. Alguns desses projetos são *Openvpn, Libpng, Libxml2, Cherokee, Sonyxperiadev, SQLite3, Arduino e TcpDump*. A lista completa pode ser vista no Apêndice A.1.

A Figura 7 ilustra a estratégia utilizada para a criação dos *pull requests*. Em cada sistema, nós utilizamos uma ferramenta que encontra anotações não disciplinadas (Passo 1). Essa ferramenta é capaz de encontrar oportunidades de aplicar o catálogo de refatorações [1], como por exemplo o lado esquerdo Figura 5. A ferramenta foi utilizada para analisar o código fonte inteiro de cada sistema. Com isso foi possível encontrar várias anotações não disciplinadas. Optamos por escolher, manualmente e de forma aleatória, apenas uma

transformação por sistema. Essa foi uma decisão tomada para evitar ruídos, como por exemplo um mesmo desenvolvedor emitir sua opinião (aceitando ou rejeitando *pull requests*) várias vezes. Então, para cada anotação, nós aplicamos a refatoração correspondente no catálogo e submetemos o *pull request* (Passo 2). Note que nós não estamos corrigindo erros ou submetendo novos casos de teste. Essas submissões contêm apenas a versão disciplinada de uma anotação. Ocasionalmente, a pedido dos desenvolvedores, foram submetidos (via *GitHub*) comentários também. Estes visam esclarecer a motivação das alterações (Passo 3). Essa é uma abordagem típica para que os *pull requests* possam ser aceitos em sistemas *open-source*.

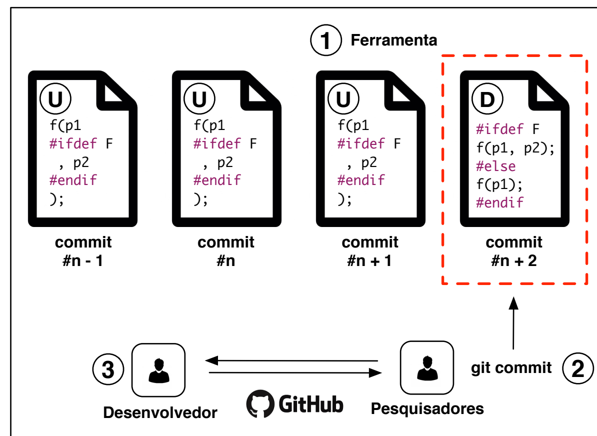

Figura 7 – Nossa estratégia para a criação de pull requests.

3.2 Resultados e Discussão

Foram submetidos 110 *pull requests*, sendo um *pull request* por sistema. Isso significa que nesse estudo consideramos 110 sistemas baseados em pré-processamento. A lista de todos os sistemas que foram submetidos pode ser vista no Apêndice A.1. A Figura 8 ilustra quatro exemplos de *pull requests* que foram submetidos. Apesar do catálogo de refatorações ser mais amplo, nós utilizamos apenas três: R2, R4 e R6. Estudos apontam [16] que a aplicabilidade de tais refatorações pode ser bastante comum em sistemas *open-source*. A Figura 8 ilustra os *pull requests* submetidos e refatorações utilizadas. No topo da Figura 8, no lado esquerdo, nós mostramos um *pull request* que foi aceito (R6). No lado direito, temos um que foi rejeitado (R4). Já na parte inferior esquerda temos um *pull-request* para código de terceiros (R2) e na direita um para código depreciado (R6). A Figura 8 também contém comentários dos desenvolvedores para cada *pull request*. A figura apresenta também os tipos de refatorações aplicadas.

A categoria dos aceitos é direta, significando que o desenvolvedor aceitou o *pull request* e realizou um *merge* com as alterações sugeridas. Em contraste, para os que foram rejeitados

(*pull requests* que não tiveram *merge*), nós tentamos entender o que motivou a rejeição. Para isto, nós consideramos duas categorias adicionais além dos rejeitados em si: depreciados e código de terceiros. A categoria dos depreciados também é direta, significando que submetemos uma alteração para um código *deprecated*, ou seja, um código que não é utilizado. Código de terceiros significa que a mudança foi feita em um código que não foi escrito pelos desenvolvedores do sistema (por exemplo um código de uma biblioteca), que de alguma forma está junto ao código do sistema. Nesse contexto, a maioria dos desenvolvedores não permitiam alterações para aquele código. Assim sendo, eles sugeriram que nós criássemos um *pull request* diretamente para o repositório do proprietário (da biblioteca, por exemplo). Note que para essas duas novas categorias, depreciados e código de terceiros, o que motivou a rejeição não foi necessariamente a alteração sugerida, mas sim, **onde** ela foi sugerida. Na parte de baixo da Figura 8, nós ilustramos essas duas categorias. Nós apresentamos também os comentários dos desenvolvedores.

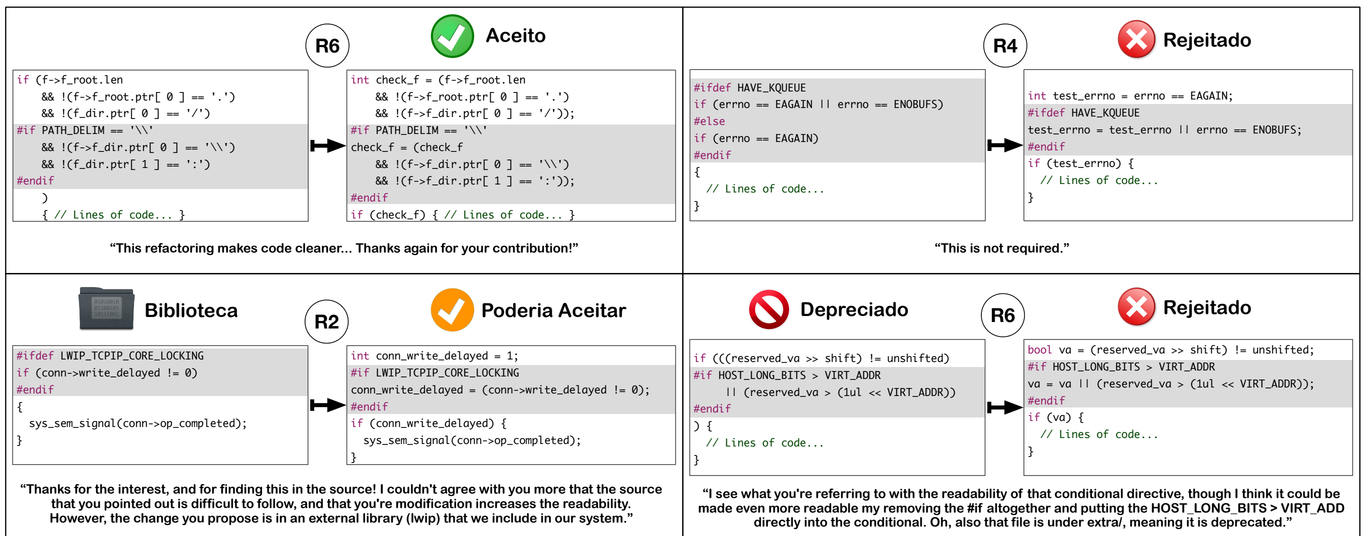

Figura 8 – Exemplos de *pull requests* submetidos. Abaixo de cada transformação, nós ilustramos as mensagens que recebemos dos desenvolvedores.

O gráfico em barras da Figura 9 ilustra a taxa de aceitação dos nossos *pull requests*. Até agora, 11 dos 110 *pull requests* estão sem resposta. Logo a Figura 9 desconsidera esses 11, i.e., estamos apresentando os resultados de 99 *pull requests*. Note que, considerando os que têm resposta, quase dois terços (63%, 62 *pull requests*) foram aceitos.

Como mencionado, nós não implementamos nenhum teste adicional para verificar nossos *pull requests*. Mesmo assim, os desenvolvedores concordaram em aceitar nossas alterações. De fato, nenhum *pull request* foi recusado por falta de casos de teste. Ainda nos aceitos, 43 (69%) dos 62 foram aceitos sem a necessidade de qualquer tipo de alteração. Por outro lado, 19 (31%) precisaram de algumas alterações, como seguir os guias de desenvolvimento de cada sistema (por exemplo, nomes de variáveis, espaços em branco,

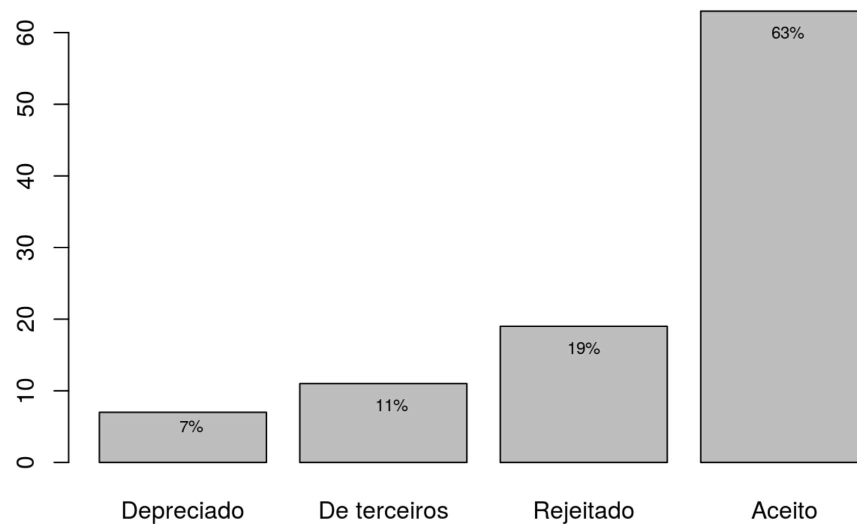

Figura 9 – Resultados dos nossos *pull requests*.

identação dos **`#ifdef`** etc). A seguir apresentaremos alguns comentários de desenvolvedores que aceitaram *pull requests*.

> "*That's much better.*"
> "*Better implementation.*"
> "*Easier to read.*"
> "*Ok, thanks for the fix. Further improvements are welcome.*"

Um desenvolvedor mencionou inclusive que refatorar anotações não disciplinadas está incluído na lista de *TODO* (a fazer), do seu projeto. "*I agree. In fact, it's in the libpng17 TODO list: Refactor preprocessor conditionals to compile entire statements.*" Agora iremos apresentar comentários de desenvolvedores que rejeitaram pull requests.

> "*We generally don't do stylistic-only changes.*"
> "*Not only is the code you wrote wrong, I also think it's not needed and probably even harder to understand.*"
> "*I'd call this code harder to read.*" "*I don't find it more readable either.*"
> "*I'm not going to randomly apply some rules from random style guides to random stb libraries.*"

Figura 10 – *Word cloud* feita a partir dos comentários obtidos nos *pull requests*.

Para facilitar a análise dos comentários obtidos, foi criada uma *word cloud* (Figura 10). Nela os comentários foram classificados em 16 categorias. Para a criação das categorias, nós buscamos agrupar respostas semelhantes. Por exemplo, "*Keep our style*", "*We generally don't do stylistic only changes*", "*Isn't worth the gain for purely stylistic changes*", "*Stylistic preference alone is insufficient*", "*Otherwise need to audit where else changes would be required*" foram agrupadas em *Stylistic changes*. Note que um comentário pode pertencer a uma ou mais categorias. Por exemplo, um desenvolvedor afirmou que o código não ficou mais fácil de entender e que os nomes das variáveis adicionadas foram ruins. Nesse exemplo o comentário contaria para as categorias *Not easier to read or understand* e *Bad variable name*. A *word cloud* representa a frequência das categorias através do tamanho da fonte, logo, *Thanks, merged* foi a resposta mais obtida em relação às outras.

Como pode ser visto na Figura 10 apenas quatro das 16 categorias representam comentários positivos em relação às refatorações. Isso ocorre porque os desenvolvedores nem sempre sentiam necessidade de comentar o motivo que os levou a aceitarem os *pull requests*.

A partir da análise da Figura 10 nós temos que a segunda categoria que mais se destaca: "*The change is an improvement*". Essa resposta que nos dá indícios de que as refatorações de fato trouxeram uma melhoria para a qualidade do código. Contudo, note que a terceira categoria com maior destaque, "*Not easier to read or understand*", contradiz a segunda "*The change is a improvement*". Essa contradição já era esperada visto que existem critérios subjetivos na avaliação da qualidade de um código. Algumas categorias complementam a "*Not easier to read or understand*" e representam parte desses critérios, como por exemplo "*Bad variable name*" ou "*Prefer to save lines of code*". Apesar dessa contradição encontrada, a frequência obtida para a categoria que considera a refatoração uma melhoria foi maior. Portanto, nós temos indícios de que os desenvolvedores se preocupam com a disciplinaridade do código.

É possível notar que alguns desenvolvedores consideram que disciplinar anotações é

unicamente uma questão de estilo, representados por "*Stylistcs changes*". Nesse caso, eles afirmam que preferem trabalhar em assuntos mais sérios, como erros, a perder tempo com questões de estilo. Por esse motivo, rejeitaram as submissões. Ainda sobre a questão de estilo, nós tivemos algumas rejeições motivadas pelo fato que o resto do código não segue o "estilo" que sugerimos (disciplinado). Além do mais, uma vez que nós não somos especialistas nos projetos que submetemos as alterações, nós acidentalmente acabamos introduzindo erros em alguns *pull requests*. Por motivos óbvios, os mesmos foram rejeitados.

Nós investigamos também os *pull requests* que foram submetidos para códigos depreciados e de terceiros. Perguntamos aos desenvolvedores se eles aceitariam o código, caso ele não tivesse sido submetido a um trecho depreciado ou de terceiros. Nesse sentido, nove (50%) dos 18 desenvolvedores responderam que teriam aceitado. Se considerarmos esses casos como potenciais aceitos, nós alcançamos uma taxa de aceitação de 71%. A seguir, apresentaremos respostas positivas e negativas para essas duas categorias.

> "*Sure, but I would love to see them on the core project.*"
> "*Oh yes, I would say it is a better implementation.*"
> "*The time involved in coding, review, merging, testing, etc., and the possibility of unintended consequences, isn't worth the gain for purely stylistic changes.*"
> "*Sorry, we have to be practical here - so many more serious bugs and issues.*"
> "*readability is subjective, personally I don't find this patch an improvement.*"
> "*I am not sure I see the advantage to this PR.*"

Baseado nos nossos resultados, quase dois terços dos desenvolvedores aceitaram nossas sugestões para remover anotações não disciplinadas. De forma geral, esses resultados sugerem que a disciplinaridade das anotações é importante e não deve ser negligenciada.

3.3 Ameaças à Validade

Submeter um *pull request* por sistema é uma ameaça à validade externa. Essa decisão foi tomada para evitar duas ou mais respostas vindas do mesmo desenvolvedor. Nós acreditamos que essa ameaça é minimizada devido ao fato que submetemos um número alto de *pull requests* para uma grande variedade de sistemas. Outra ameaça à validade externa é o fato que utilizamos um pequeno número de refatorações do catálogo para disciplinar anotações [1]. Apenas três tipos de refatorações foram utilizados (como pode ser visto na Figura 8). Nesse sentido, nós não podemos generalizar esses resultados para os outros tipos de refatorações. Não obstante, nós utilizamos essas três refatorações por conta da alta frequência de oportunidades que temos de aplicá-las [15]. Em outras palavras, o

lado esquerdo das refatorações vistas na Figura 8 são comuns em códigos de sistemas *open-source*. A respeito dos *pull requests* rejeitados e que foram submetidos a códigos de terceiros e depreciados, nove (50%) dos 18 desenvolvedores responderam que aceitariam a alteração, caso ela tivesse sido feita em um código em uso ou proprietário. Nós adicionamos esses casos à taxa total de aceitação (71%). Essa decisão representa uma ameaça, visto que ela depende de uma questão de interpretação, pois a alteração não recebeu um *merge* no projeto. Em alguns casos, desenvolvedores perguntaram porque eles deveriam aceitar nossas sugestões para remover anotações não disciplinadas. Nesses casos nós arguíamos a favor das anotações disciplinadas. Mencionávamos, por exemplo, que o código ficaria mais fácil de compreender. Nesse sentido, nossos comentários podem ter induzido a aceitação de alguns *pull requests*. Portanto, esses comentários representam também uma ameaça. Porém, minimizamos essa ameaça pois essa etapa de arguição faz parte do processo de submissão de *pull requests*. Em outras palavras, um desenvolvedor que submete uma alteração a um projeto acredita que a mesma irá trazer algum benefício ao projeto e, portanto, fará comentários a favor dela. Uma outra ameça está na escolha dos sistemas. Nesse sentido nós não utilizamos nenhum tipo técnica como aleatorização para a escolha. Contudo, essa ameaça é reduzida pela variedade de sistemas utilizados.

4 Contato com os Desenvolvedores: Minerando repositórios

Neste estudo nós mineramos código de repositórios *open source* buscando refatorações onde os desenvolvedores disciplinam anotações não disciplinadas. Com este estudo pretendemos entender como o processo de disciplinar anotações ocorre na prática. A Tabela 3 resume a questão de pesquisa, abordagem e hipótese do estudo. Neste capítulo iremos apresentar a configuração, a ferramenta desenvolvida e os resultados.

– Estudo 1 –	– Estudo 2 –
Q1: Os desenvolvedores preferem código disciplinado?	**Q2**: Os desenvolvedores disciplinam anotações condicionais? Como?
A1 Sugerir *pull-requests* disciplinando o código.	**A2** Minerar repositórios em busca de *commits* onde essas refatorações ocorreram e entrar em contato com os desenvolvedores responsáveis.
H1: Teremos uma taxa de aceitação significativa, indicando que os desenvolvedores se importam com a disciplinaridade das anotações condicionais.	**H2**: Iremos encontrar *commits* que transformam código não disciplinado em código disciplinado.

– Estudo 3 –
Q3: A disciplinaridade das anotações afeta a realização de tarefas de manutenção de código nos quesitos de tempo e propensão a erros?
A3 Realizaremos um experimento controlado com alunos de cursos de graduação a fim de medir o impacto da disciplinaridade.
H3: As tarefas não disciplinadas vão consumir mais tempo e mais tentativas para serem concluídas.

Tabela 3 – Visão geral da questão de pesquisa, abordagem utilizada e hipótese para o Estudo 2.

4.1 Configuração

Nossa investigação tem foco em entender se os desenvolvedores realizam refatorações de código a fim de disciplinar anotações não disciplinadas. Com essa investigação esperamos responder as seguintes questões de pesquisa: (1) Os desenvolvedores disciplinam o código? (2) Qual a motivação para realizar tais refatorações? (3) Como ocorrem as refatorações?

Para responder a primeira questão nós mineramos códigos de repositórios *open source* para encontrar transformações. Uma transformação consiste em transformar código não disciplinado em disciplinado, podendo ser uma refatoração ou não. Adicionalmente, para as transformações encontradas, entramos em contato com o desenvolvedor via *email*. Esse

contato busca responder a segunda questão. Por fim, para responder a terceira questão, nós buscamos classificar as transformações encontradas.

No contexto de entender como ocorrem as transformações, tencionamos distinguir se elas ocorrem de forma oportuna ou se são frutos de uma tentativa de melhoria do código. As melhorias no código se referem a uma alteração feita pelo desenvolvedor apenas com intenção de melhorar a qualidade do código (refatoração) e foram classificadas como "**perfectivas**" [50]. Com alteração oportuna, nos referimos à uma alteração que foi feita em conjunto com outras, por exemplo, o desenvolvedor corrigiu um erro de comportamento e também melhorou o código, estas foram classificadas como "**não perfectivas**". Acreditamos que os comentários do *commit* e uma análise do código alterado são suficientes para realizar essa classificação.

Responder essas questões nos trará indícios de que, em um cenário real, existe a preocupação com a disciplinaridade do código anotado. Adicionalmente, o contato com esses desenvolvedores, através de *emails*, irá trazer um maior entendimento de como e porquê o processo de disciplinar código ocorre.

Para realizar o processo de mineração de repositórios, nós desenvolvemos uma ferramenta. A Figura 11 ilustra o processo de identificação das transformações e contato com os desenvolvedores. Para cada *commit*(n + 1) é feita uma comparação com o anterior (n). Nesta comparação é verificado se, dentro do código alterado, houve um aumento do número de anotações disciplinadas e uma diminuição do número de anotações não disciplinadas em relação ao anterior (Passo 1). Caso ocorra, é emitido um alerta e esse *commit* é verificado manualmente (Passo 2). Caso a análise manual confirme que o código foi disciplinado e o *email* do desenvolvedor que o realizou esteja disponível, nós enviamos um *email* perguntando a motivação para realizar tal alteração (Passo 3).

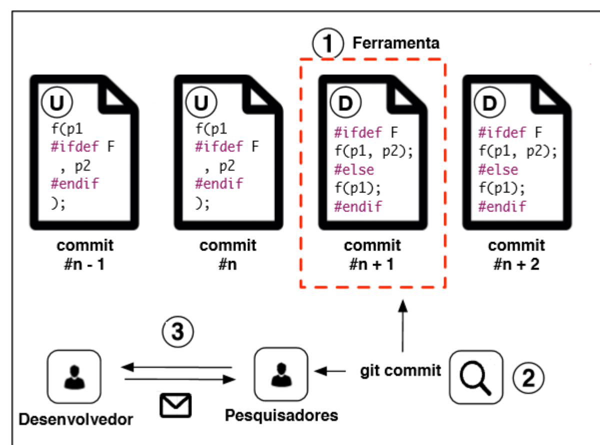

Figura 11 – Processo de identificação de transformações e contato com desenvolvedores.

A ferramenta foi utilizada em 25 projetos e foi possível encontrar 90 *commits* com

transformações de código não disciplinado para disciplinado. Ao afirmar que um *commit* contém uma transformação, nós consideramos que a lista de alterações daquele *commmit* contém pelo menos um arquivo onde uma anotação foi disciplinada. As transformações que foram realizadas junto com mudanças de comportamento do código ou correção de erros, como pode ser visto na Figura 12, são consideradas **não perfectivas**. Foram encontrados, também *commits* que continham apenas refatorações, estes, são considerados **perfectivos**. Um exemplo pode ser visto na Figura 13.

```
...
-   #ifdef PNG_HANDLE_AS_UNKNOWN_SUPPORTED
else
     {
     if (png_ptr->push_length + 4 > png_ptr->buffer_size)
     {
      png_push_save_buffer(png_ptr);
      return;
     }
      png_handle_unknown(png_ptr, info_ptr, png_ptr->push_length,PNG_HANDLE_CHUNK_AS_DEFAULT);
-    }
     #endif
```

```
...
else
     {
     if (png_ptr->push_length + 4 > png_ptr->buffer_size)
     {
       png_push_save_buffer(png_ptr);
     return;
     }
+    #ifdef PNG_HANDLE_AS_UNKNOWN_SUPPORTED
      png_handle_unknown(png_ptr, info_ptr, png_ptr->push_length,PNG_HANDLE_CHUNK_AS_DEFAULT);
     #endif
+    }
```

Figura 12 – Exemplo de refatoração com correção de erro (não perfectiva).

No total foram enviados 40 *emails* onde 19 foram respondidos. Essas respostas foram categorizadas para a criação de uma *word cloud*. No total, as respostas foram divididas em 12 categorias. As mensagens enviadas seguiam um *Template* 1 que foi produzido a fim de evitar que a resposta do participante seja influenciada. Para isto, foi criada uma mensagem aberta que evita revelar objetivo da entrevista.

4.2 Resultados e Discussão

Foram encontradas 90 transformações em 25 projetos. Ao contrário do estudo visto no Capítulo 3, nós estamos considerando uma ou mais transformações por projeto e desenvolvedor, pois, desta vez, estamos estudando se elas ocorrem. As transformações encontradas foram separadas em duas classificações. A perfectiva, que representa as transformações que são refatorações (melhorias de código). E a não perfectiva, representando as melhorias

```
...
-     that->pm->limit += pow(
    #if PNG_MAX_GAMMA_8 < 14
-     (that->this.bit_depth == 16 ? 8. :
-     6. + (1<<(15-PNG_MAX_GAMMA_8)))
    # else
-     8.
    # endif
-     /65535, data.gamma);
```

```
...
     #if PNG_MAX_GAMMA_8 < 14
+     that->pm->limit += pow((that->this.bit_depth == 16 ? 8. :
+     6. + (1<<(15-PNG_MAX_GAMMA_8)))/65535, data.gamma);
    # else
+     that->pm->limit += pow((that->this.bit_depth == 16 ? 8. :
+     8. + (1<<(15-PNG_MAX_GAMMA_8)))/65535, data.gamma);
    # endif
```

Figura 13 – Exemplo de refatoração visando a melhoria da qualidade do código (perfectiva).

Dear <developer_name>,
As part of a research team with members from Federal University of Alagoas and Federal University of Campina Grande, Brazil, we are investigating the use of the C Preprocessor in open-source projects. We are analyzing software repositories to understand certain types of preprocessor conditional directives.
We found that you are an active developer in the <project_name> project and we would like to ask you one question about a specific patch you have submitted. In commit <id_commit_with_link>, you have made the following changes:
<code>
Your changes in this specific part of the source code seem to either not add new functionalities or modify the code behavior. We are interested in understanding the reasons of such changes. May you explain these reasons?
Thanks in advance.

Template 1: *Template* para envio de mensagens aos desenvolvedores.

de código que vieram de forma oportuna. Das 90 transformações encontradas, 60 foram classificadas como perfectivas. A Figura 14 ilustra essa categorização. O número razoável de transformações encontradas sugere que os desenvolvedores removem anotações não disciplinadas do código (Questão 1).

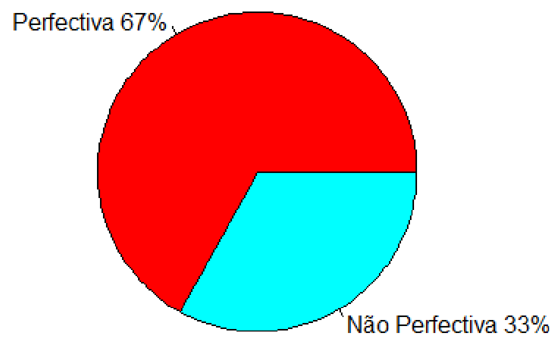

Figura 14 – Gráfico representando a classificação das transformações encontradas.

> *Baseado no fato que encontramos um número razoável de transformações, podemos inferir que os desenvolvedores removem anotações não disciplinadas do código.*

Dessas 90 transformações que foram identificadas, contatamos, via *email*, 40 desenvolvedores. Não foi possível contatar todos os desenvolvedores responsáveis, pois, nem sempre os contatos dos mesmos estava disponíveis. Este contato foi feito a fim de entender por que eles fizeram essas transformações. Desses 40 desenvolvedores apenas 19 responderam. As respostas obtidas foram categorizadas e apresentadas na Figura 15. Alguns exemplos de resposta obtidos podem ser vistos a seguir:

> "*...The change clearly removes useless preprocessor directives and overall makes the code clearer to read. Clarifying code...*"
> "*...The discussed change is part of a bugfix, and since both parts of the ifdef are identical at the time, this was obviously meant as a temporary change...*"
> "*...Someone felt splitting the strcspn() arguments across #ifdef blocks was too complex, and it would just be cleaner to call the function twice in separate blocks...*"
> "*Because it fixes the syntax of the code, else it was not valid C in all cases.*"

Para facilitar a análise das respostas obtidas, foi criada uma *word cloud*. Nela as respostas foram classificadas em 12 categorias. Para criação das categorias, nos buscamos agrupar respostas que tenham o mesmo sentido. Como por exemplo: "*Fix Syntax*", "*Bug*

fix” e “*Correct logical error*” foram agrupadas em “*Bug fix*”. Note que o comentário pode pertencer a mais de uma categoria. Por exemplo, numa situação de transformação não perfectiva, pode acontecer do desenvolvedor ter a resposta classificada como “*Bug fix*” e “*Improve the code quality*”.

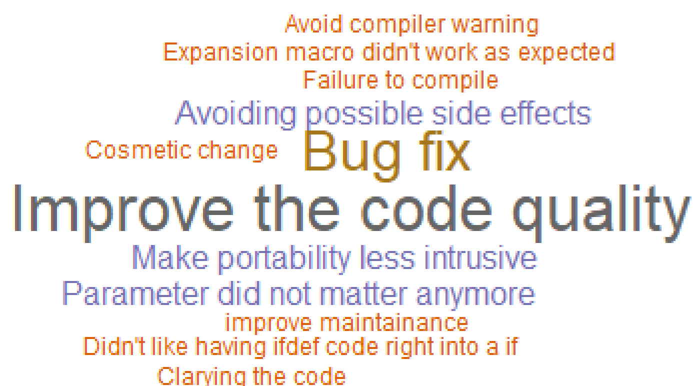

Figura 15 – Exemplo de refatoração com correção de erro (não perfectiva).

Como pode ser visto na Figura 15, a resposta com maior destaque se refere a melhorar a qualidade do código. A segunda categoria “*Bug fix*” pode correlacionar o surgimento de *bugs* ao uso de anotações não disciplinadas. Contudo, essa afirmação não pode ser comprovada apenas com essa análise. Uma categoria interessante é a de “*Avoiding possible side effects*”, que está relacionada ao fato que alguns desenvolvedores temem que aquela anotação não disciplinada pode gerar um resultado não esperado. Essa categoria geralmente está associada ao “*Failure to compile*”. A análise da *word cloud* nos permite concluir que na maioria das vezes a remoção de uma anotação não disciplinada é motivada por uma melhoria no código ou correção de um *bug* (Questão 2). Adicionalmente, ao considerar as categorias “*Improve the code quality*”, “*Clarifying the code*”, “*Make the portability less intrusive*” e “*Cosmetic change*” como melhoria de código. Em conjunto com a categorização das transformações (67 % dos casos foram considerados refatorações), podemos inferir que as transformações ocorrem, na maioria dos casos, com foco em melhorar o código em vez de ser apenas uma alteração oportuna (Questão 3).

Através do número razoável de transformações encontradas, concluímos que os desenvolvedores refatoram código não disciplinado. Adicionalmente, baseado na categorização das transformações encontradas em conjunto com a análise do word cloud*, podemos sugerir que as transformações ocorrem na maioria dos casos com a intenção de melhorar a qualidade do código.*

4.3 Ameaças à Validade

Uma ameaça a esse estudo é o fato de que uma das questões envolve perguntar o que motivou o desenvolvedor a realizar a transformação. Ao executar a refatoração, fica implícito que o desenvolvedor a vê como uma melhoria. Contudo, a principal intenção desse estudo está em entender como o processo de disciplinar ocorre e se de fato ele ocorre. Entender por exemplo, o que o desenvolvedor considera que foi melhorado ao refatorar o código. Portanto, essa ameaça pode ser minimizada. Esse estudo tenciona destacar a relevância do uso de anotações disciplinadas e não o contrastar com o uso das não disciplinadas.

Existem algumas ameaças em relação à ferramenta desenvolvida. A primeira é em relação a confiabilidade dela. Ela foi implementada a fim de indicar suspeitas de transformações e, portanto, não é capaz de classificar de forma precisa se ocorreu realmente uma refatoração. Essa ameaça foi minimizada pelo fato de que para cada suspeita de transformação nós executamos uma avaliação manual. Portanto, nenhuma das refatorações encontradas deixou de passar por uma avaliação mais precisa. Ainda em relação à ferramenta, ela não foi desenvolvida a fim de identificar todas as possíveis ocorrências de transformações. Com ela, pretendíamos encontrar um grupo mínimo de ocorrências. Para minimizar essa ameaça, optamos por usar uma quantidade grande de projetos.

Outra ameaça à validade está relacionada aos projetos utilizados. Existe a possibilidade de alguma *code guideline* obrigar o desenvolvedor a realizar esse tipo de refatoração. Essa ameaça é reduzida, pois, escolhemos uma grande quantidade de projetos de diferentes tamanhos e popularidades. Uma outra ameça está na escolha dos sistemas, nós não utilizamos nenhum tipo técnica como aleatorização para a escolha. Contudo, essa ameaça é reduzida pela variedade de sistemas utilizados.

Outra ameaça está relacionada às transformações encontradas. Muitas delas ocorreram há muito tempo (algumas com mais de 10 anos). Isso dificulta o contato com os desenvolvedores, pois, em alguns casos, eles podem já não estar mais associados ao projeto ou até mesmo terem dificuldades em lembrar o que motivou aquela alteração. Para minimizar esse efeito, nós enviamos o trecho de código e o *link* do *commit* na mensagem do *email*. A mensagem do *email* também pode representar uma ameaça. É possível que ela influencie a resposta do desenvolvedor. Para reduzir essa ameaça, nós tentamos manter a pergunta mais aberta possível: "...notamos que você fez uma alteração que não parece alterar o comportamento do código. Nós gostaríamos de entender os seus motivos. Você poderia explicar?" Outra ameaça em relação à pergunta é que por ela ser aberta, algumas respostas acabaram sendo abertas também, e consequentemente, fugiram ao foco da pesquisa. Ainda em relação aos *emails*, apesar de 47% dos *emails* terem sido respondidos, apenas 40 desenvolvedores foram contactados. Essa ameaça é reduzida pelo fato de que a ferramenta desenvolvida não é capaz de achar todos os tipos de refatorações, além de que, apenas

16% das anotações condicionais são não disciplinadas [15], portanto se espera um número baixo de ocorrências de transformações. Adicionalmente, algumas das transformações encontradas são muito antigas e o desenvolvedor pode não estar mais associado ao projeto, consequentemente, é difícil entrar em contato com o desenvolvedor responsável. Por fim, a classificação das transformações e a categorização das respostas dos *emails* podem ser consideradas uma ameaça, pois, nem sempre essas caracterizações partem de critérios objetivos. Para diminuir essa ameaça nós utilizamos uma estratégia de *triple check* onde três pessoas individualmente classificavam as transformações.

5 Experimento Controlado

Neste capítulo iremos apresentar o nosso experimento controlado. Neste experimento iremos mitigar duas limitações do anterior [17]: a falta de replicação e bloqueio. Iremos apresentar a configuração, as unidades do experimento e os resultados A Tabela 4 resume a questão de pesquisa, abordagem e hipótese do estudo.

– Estudo 1 –	– Estudo 2 –
Q1: Os desenvolvedores preferem código disciplinado?	**Q2**: Os desenvolvedores disciplinam anotações condicionais? Como?
A1 Sugerir *pull-requests* disciplinando o código.	**A2** Minerar repositórios em busca de *commits* onde essas refatorações ocorreram e entrar em contato com os desenvolvedores responsáveis.
H1: Teremos uma taxa de aceitação significativa, indicando que os desenvolvedores se importam com a disciplinaridade das anotações condicionais.	**H2**: Iremos encontrar *commits* que transformam código não disciplinado em código disciplinado.
– Estudo 3 –	
Q3: A disciplinaridade das anotações afeta a realização de tarefas de manutenção de código nos quesitos de tempo e propensão a erros?	
A3 Realizaremos um experimento controlado com alunos de cursos de graduação a fim de medir o impacto da disciplinaridade.	
H3: As tarefas não disciplinadas vão consumir mais tempo e mais tentativas para serem concluídas.	

Tabela 4 – Visão geral da questão de pesquisa, abordagem utilizada e hipótese para o Estudo 3.

5.1 Configuração

Nossa investigação foca em tentar entender o efeito da disciplinaridade das anotações condicionais em tarefas de manutenção de software. Para isso, iremos tentar responder as seguintes questões de pesquisa: *(1) O uso de anotações não disciplinadas faz com que o tempo para resolver tarefas de manutenção aumente? (2) O uso de anotações não disciplinadas faz com que o desenvolvedor cometa mais erros ao resolver uma tarefa de manutenção?* Para responder essas perguntas, nós iremos executar um experimento controlado. Nesse experimento os participantes irão executar atividades com alguns parâmetros definidos. Essas atividades foram baseadas em códigos reais encontrados em repositórios como *LIBXML2, VIM, Linux*. No total serão seis atividades divididas em dois grupos. Cada grupo de atividades foi criado a partir de um sistema diferente, *LIBXML2* e *VIM* respectivamente. Todas as atividades têm duas versões, uma disciplinada e não disciplinada.

Esses dois grupos de atividades possuem pares de atividades semelhantes, ou tipos. Por exemplo a primeira atividade de cada grupo se trata de corrigir um erro sintático. Logo, a primeira atividade de cada grupo representa o tipo "correção de erro sintático." Cada participante irá receber um grupo de atividades na versão disciplinada e outro na versão não disciplinada. A execução das atividades será medida. Nós iremos contar quanto tempo cada atividade levou para ser finalizada. Por atividade finalizada entende-se uma atividade que foi concluída dentro dos parâmetros definidos. Iremos medir também a quantidade de erros ocorridos (tentativas) até a conclusão da mesma. Por fim iremos executar testes de hipótese para saber se houve realmente uma diferença significativa entre as execuções de diferentes disciplinaridades. Nesse sentido, estabelecemos as seguintes hipóteses nulas:

- $H1_0$: Não há diferença significativa no tempo necessário para se fazer manutenção em um código disciplinado (tratamento de controle) em relação a um não disciplinado (tratamento sobre investigação).
- $H2_0$: Não há diferença significativa entre o número de erros cometidos por um desenvolvedor quando a manutenção é feita em um código disciplinado (tratamento de controle) em relação a um não disciplinado (tratamento sobre investigação).

Uma vez que estamos comparando dois tratamentos, no caso da hipótese nula ser rejeitada, basta comparar o valor médio entre as observações de controle e tratamento para estimar o efeito das anotações não disciplinadas.

A análise será feita de duas formas. Na primeira forma iremos estudar o impacto da disciplinaridade nas atividades por tipo. Na segunda forma, iremos considerar os resultados conjunto dos tipos.

Nós optamos por utilizar um *design* diferente do estudo anterior. Em Schulze et al. [17], os participantes foram separados em dois grupos que realizaram tarefas para cada tratamento. Um grupo realizou tarefas com anotações disciplinadas e o outro com anotações não disciplinadas. A variável de confusão relacionada à experiência dos participantes foi tratada com o uso de um questionário para medir a mesma. Uma vez que essa experiência foi medida, eles foram divididos em grupos nivelados. Nós acreditamos que esse tipo de abordagem gera ruídos desnecessários, pois não há como garantir que o nivelamento foi justo. Adicionalmente, essa abordagem impede a aleatorização dos participantes. Identificamos, também que não houve um tratamento para o aprendizado do participante, o que representa outra fonte de ruídos. No experimento anterior, os alunos são divididos em grupos que fazem atividades relacionados a um só tratamento. Uma outra fonte de ruído não tratada foi o engajamento dos participantes. Por exemplo, se os participantes que fazem parte do grupo disciplinado são mais engajados, é esperado que tenham resultados melhores que o grupo não disciplinado. Logo, o que estaria gerando a diferença seria o desnível entre

grupos e não a técnica. Por fim, os grupos de diferentes tratamentos não tiveram o mesmo conjunto de tarefas e foram comparados como se estivessem fazendo as mesmas tarefas. Por exemplo, na sétima tarefa o grupo responsável pelo tratamento disciplinado deveria identificar e corrigir um erro, enquanto o não disciplinado deveria apenas identificar.

Para bloquear essas variáveis citadas do trabalho de Schulze et al. [17], nós optamos por utilizar como *design* o quadrado latino [51]. Com esse *design* é possível dividir, de forma aleatória, os participantes em pares. Essa divisão trata o ruído da falta de aleatoriedade. Adicionalmente, cada participante irá fazer dois conjuntos de tarefas, onde cada conjunto recebe um tratamento diferente. Com isso, estamos tratando o ruído da experiência e do engajamento, visto que a experiência e engajamento de cada participante vai contar para cada tratamento. Ainda sobre as tarefas, cada conjunto de tarefas foi montado a partir de códigos de sistemas diferentes e cada sistema tem sua versão para os dois tratamentos. Assim, diminuímos o ruído do aprendizado, pois cada tratamento será visto em códigos distintos. Por fim, as tarefas de cada conjunto, mesmo em códigos diferentes, são correspondentes. Com isso garantimos que a comparação dos resultados seja feita a partir de um mesmo conjunto de tarefas. Diante disto, consideramos o quadrado latino de segunda ordem como um modelo adequado para uma situação que envolve comparar dois tratamentos diferentes.

A Figura 16 ilustra a formação dos quadrados latinos. Em primeiro lugar, os participantes são separados em pares de forma aleatória. Cada réplica do quadrado compreende um par de participantes. Após a formação dos quadrados, é sorteado para um dos participantes do quadrado qual tratamento ele receberá no seu primeiro conjunto de tarefas. Por exemplo, no "Quadrado 1" temos que o participante 1 recebeu o formato não disciplinado para seu Conjunto de Tarefas 1 (CT_1). Uma vez que um dos participantes tem seu tratamento de conjunto de tarefas definido, os outros tratamentos do quadrado são definidos automaticamente. Ainda sobre o "Quadrado 1", como o CT_1 do participante 1 foi não disciplinado, seu Conjunto de Tarefas 2 (CT_2) será disciplinado. Uma vez que o participante 1 tem seus tratamentos definidos, o participante 2 também os tem. O participante 2 irá receber tratamentos diferentes do 1. Como pode ser visto, ele recebeu o tratamento disciplinado no primeiro conjunto de tarefas e não disciplinado no segundo. Todos os quadrados seguem esta mesma regra: é sorteado o tratamento do primeiro conjunto de tarefas de um participante e os outros são definidos automaticamente. Uma vez que temos uma quantidade razoável de participantes, nossa análise vai ser sobre vários quadrados. Esse modelo também nos leva a ter uma réplica por quadrado latino (aumentando o número de graus de liberdade, *degrees of freedom*) [52].

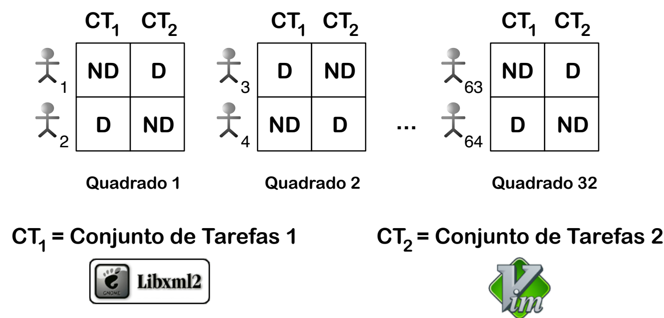

Figura 16 – Projeto do experimento utilizando quadrados latinos.

5.2 Unidades Experimentais

Os participantes do nosso experimento consistem em 64 alunos de diferentes cursos da Universidade de Brasília (UnB): Ciências da computação, Engenharia da computação e Mecatrônica. Todos os participantes estão matriculados em um curso de técnicas avançadas de programação e têm de três a cinco semestres de experiência em programação (em particular, usando linguagens como C/C++ e Java). Eles também têm uma experiência prévia usando o pré-processador do C. Os participantes foram escolhidos aleatoriamente e separados em um conjunto de 32 quadrados latinos. Nós realizamos um treino de 50 minutos. Nesse treino foi ensinado os conceitos básicos de sistemas configuráveis usando o pré-processador do C. A fim de evitar o surgimento de ruídos, no material de treino haviam exemplos de anotações disciplinadas e não disciplinadas. Em nenhum momento do treino os participantes foram requisitados ou treinados a distinguir entre as duas disciplinaridades. Nós mostramos apenas que diferentes granularidades de código podem ser anotadas. Antes do experimento em si, os participantes tiveram que realizar três tarefas de aquecimento compostas de anotações disciplinadas e não disciplinadas. Note que para esse aquecimento, ambos os participantes dos quadrados realizaram tarefas usando os mesmos tratamentos. Acreditamos que o treino realizado não foi capaz de revelar o objetivo do experimento, visto que não foi formado conhecimento prévio em relação a disciplinaridade de anotações.

Foram preparadas duas máquinas virtuais para a execução do experimento. Cada máquina tinha seis instâncias pré-configuradas do IDE Eclipse.[1] Essas instâncias continham um *plugin* [53] para coleta de dados de uso, relativos a tempo e número de tentativas. Cada tarefa tinha sua própria instância do IDE Eclipse. A Máquina Virtual 1 estava configurada para os participantes que iriam responder o primeiro conjunto de tarefas (CT_1) na forma disciplinada e o segundo conjunto de tarefas (CT_2) na forma não disciplinada. Já para a Máquina Virtual 2, nós invertemos nossos tratamentos seguindo o modelo do quadrado latino, conforme ilustrado na Figura 17.

[1] https://eclipse.org/

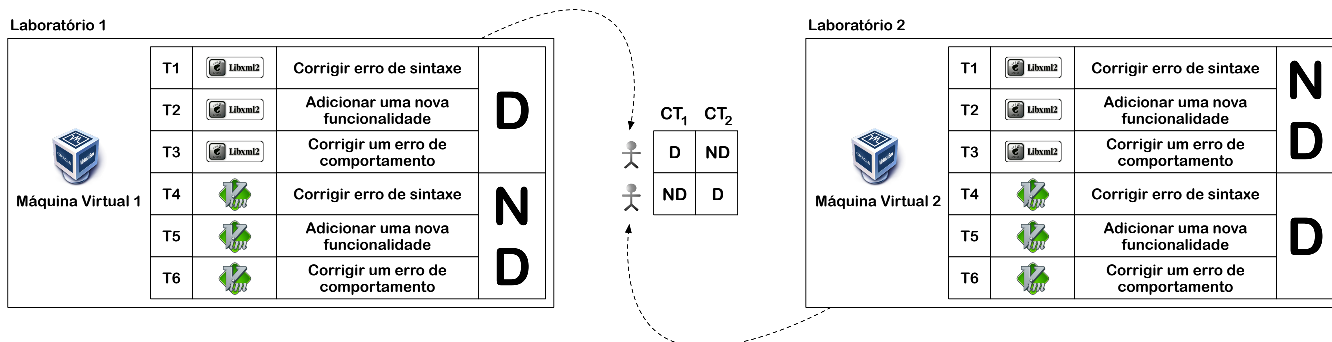

Figura 17 – Estrutura da distribuição de tarefas em termos de unidades de experimento.

Nossas tarefas foram baseadas em códigos existentes de dois sistemas altamente configuráveis, *LIBXML2* e *VIM*, os quais foram utilizados em outros estudos [15, 17, 37, 46]. Nós adaptamos o código para simplificar a execução das tarefas, i.e., removemos trechos de código que não estavam relacionados à tarefa. Nossas tarefas foram inspiradas em códigos do *Linux*[2] e *VIM*.[3] As Figuras 18, 19 e 20 ilustram códigos nos quais nos baseamos para criar as tarefas do nosso experimento. Os códigos das tarefas podem ser vistos no Apêndice A.2. A simplificação foi feita também para que os participantes pudessem terminar as tarefas em até 150 minutos. Foram criados três tipos de tarefa para cada sistema. No primeiro tipo, os participantes devem corrigir um erro de sintaxe, identificando e removendo um parêntese adicional em uma configuração específica. Na segunda tarefa, existem duas funções alternativas, anotadas com diretivas **`#ifdef`** e **`#else`**. A tarefa consiste em adicionar uma nova chamada de função. Todas as chamadas, incluindo a chamada a ser adicionada, têm parâmetros anotados com diretivas de pré-processamento. Por fim, na terceira tarefa, os participantes devem corrigir um erro de comportamento. O código contém diferentes variações de expressões com variáveis. Algumas partes dessas expressões são opcionais e somente uma configuração retorna um valor incorreto. Os participantes devem corrigir somente essa configuração. No total, foram preparados 12 cenários de manutenção: seis com anotações disciplinadas e seis com não disciplinadas. Para simplificar a distribuição dos participantes, foram configurados dois laboratórios do departamento de Ciência da Computação da UnB ("*Laboratório 1*" e "*Laboratório 2*" na Figura 17).

[2] http://www.linux.org/
[3] http://www.vim.org/

```
…
#ifdef FEAT_XCLIPBOARD
      xterm_Shell != 0
#if defined(USE_XSMP) ||
defined(FEAT_MZSCHEME)
      ||
#endif
#endif
#ifdef USE_XSMP
      xsmp_icefd != -1
#ifdef FEAT_MZSCHEME
      ||
#endif
#endif
#ifdef FEAT_MZSCHEME
  (mzthreads_allowed() && p_mzq > 0)
#endif
      ))
```

Figura 18 – Trecho de código do *VIM* que inspirou a tarefa de corrigir um erro de sintaxe.

5.3 Execução e Procedimentos de Análise de Dados

Primeiramente nós separamos aleatoriamente os 64 participantes em 32 réplicas de quadrados latinos. Então, nós escolhemos os tratamentos aleatoriamente (Disciplinado x Não Disciplinado) para as células de cada quadrado. Na verdade, após escolher o tratamento da primeira célula de um quadrado latino de segunda ordem, os tratamento para as outras células são definidas automaticamente. Após isso, conduzimos os participantes para o Laboratório 1, caso começassem com o tratamento disciplinado, ou o Laboratório 2, caso contrário. Quando os participantes chegaram nos laboratórios receberam instruções sobre os procedimentos a serem seguidos e as tarefas a serem realizadas. Cada participante recebeu um documento impresso, contendo as descrições das tarefas e como deveriam ser executadas. Os participantes foram instruídos a abrir o Eclipse correspondente à primeira tarefa e, após ler e entender o enunciado, ligar o cronômetro (botão de *Iniciar*). Este botão faz parte de um *plugin* para o Eclipse que desenvolvemos para este experimento. O *plugin* também tem um botão para parar o tempo, caso o participante precisasse tirar uma dúvida, por exemplo. Adicionalmente, o *plugin* tem um botão para compilar o código para possíveis testes que o participante desejasse realizar. Por fim, havia um botão para finalizar a tarefa. Ao clicar neste botão, o *plugin* executava vários casos de teste para checar se a tarefa foi concluída com êxito ou não (por exemplo, se os erros de sintaxe e comportamento foram corretamente corrigidos). Caso os testes passem, o participante é instruído a fazer a próxima tarefa no próximo Eclipse. Isso ocorreu sucessivamente até concluir as seis tarefas. Caso os testes não passem, o cronômetro não é parado e é contabilizado um erro (uma tentativa por parte do participante). Os participantes devem tentar até conseguirem finalizar cada tarefa. A Figura 21 ilustra a tela do *plugin*.

```
return 3 * nla_total_size(0) /* CTA_TUPLE_ORIG|REPL|MASTER */
return sprintf(page,"%4sBursts: TX"
#if !defined(CONFIG_ATM_ENI_BURST_TX_16W) && \
    !defined(CONFIG_ATM_ENI_BURST_TX_8W) && \
    !defined(CONFIG_ATM_ENI_BURST_TX_4W) && \
    !defined(CONFIG_ATM_ENI_BURST_TX_2W)
            " none"
#endif
#ifdef CONFIG_ATM_ENI_BURST_TX_16W
            " 16W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_TX_8W
            " 8W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_TX_4W
            " 4W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_TX_2W
            " 2W"
#endif
            ", RX"
#if !defined(CONFIG_ATM_ENI_BURST_RX_16W) && \
    !defined(CONFIG_ATM_ENI_BURST_RX_8W) && \
    !defined(CONFIG_ATM_ENI_BURST_RX_4W) && \
    !defined(CONFIG_ATM_ENI_BURST_RX_2W)
            " none"
#endif
#ifdef CONFIG_ATM_ENI_BURST_RX_16W
            " 16W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_RX_8W
            " 8W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_RX_4W
            " 4W"
#endif
#ifdef CONFIG_ATM_ENI_BURST_RX_2W
            " 2W"
#endif
#ifndef CONFIG_ATM_ENI_TUNE_BURST
            " (default)"
#endif
            "\n","");
```

Figura 19 – Trecho de código do *Linux Kernel* que inspirou a tarefa de adicionar nova funcionalidade.

Talvez pela natureza das tarefas, apenas 30 participantes concluíram as tarefas dentro do prazo de 150 minutos. Diferentemente do experimento realizado anteriormente [17], nós não consideramos tarefas "quase corretas". Nós optamos por uma abordagem conservadora, e assim todos os dados de participantes que não conseguiram concluir todas as tarefas foram descartados. Em virtude disso, algumas das réplicas de quadrados latinos ficaram desfalcadas. Então, para minimizar o número de réplicas perdidas, resolvemos rearranjar de forma aleatória os quadrados incompletos em novos quadrados. Para isso, nós dividimos os participantes que sobraram, ou que a sua dupla de quadrado não concluiu as tarefas, em dois grupos. O primeiro grupo corresponde aos participantes que começaram com as tarefas no tratamento disciplinado e o segundo aos que começaram no tratamento não disciplinado. Após essa divisão, nós sorteamos novamente, de forma aleatória, as duplas

```
return 3 * nla_total_size(0) /* CTA_TUPLE_ORIG|REPL|MASTER */
          + 3 * nla_total_size(0) /* CTA_TUPLE_IP */
          + 3 * nla_total_size(0) /* CTA_TUPLE_PROTO */
          + 3 * nla_total_size(sizeof(u_int8_t)) /* CTA_PROTO_NUM */
          + nla_total_size(sizeof(u_int32_t)) /* CTA_ID */
          + nla_total_size(sizeof(u_int32_t)) /* CTA_STATUS */
          + nla_total_size(sizeof(u_int32_t)) /* CTA_TIMEOUT */
          + nla_total_size(0) /* CTA_PROTOINFO */
          + nla_total_size(0) /* CTA_HELP */
          + nla_total_size(NF_CT_HELPER_NAME_LEN) /* CTA_HELP_NAME */
          + ctnetlink_secctx_size(ct)
#ifdef CONFIG_NF_NAT_NEEDED
          + 2 * nla_total_size(0) /* CTA_NAT_SEQ_ADJ_ORIG|REPL */
          + 6 * nla_total_size(sizeof(u_int32_t)) /* CTA_NAT_SEQ_OFFSET */
#endif
#ifdef CONFIG_NF_CONNTRACK_MARK
          + nla_total_size(sizeof(u_int32_t)) /* CTA_MARK */
#endif
#ifdef CONFIG_NF_CONNTRACK_ZONES
          + nla_total_size(sizeof(u_int16_t)) /* CTA_ZONE */
#endif
          + ctnetlink_proto_size(ct)
          ;
```

Figura 20 – Trecho de código do *Linux Kernel* que inspirou a tarefa de corrigir um erro de comportamento.

entre esses grupos.

Considerando apenas os quadrados que sobreviveram e os que foram rearranjados, encontramos algumas observações que achamos suspeitas, com tempos que pareciam ser baixos demais. Talvez o participante esqueceu de ativar o cronômetro na hora certa. Para evitar ruídos, resolvemos que todos os valores que fossem próximos ao primeiro quartil, no caso quatro minutos, seriam imputados. Para isso, nós utilizamos a biblioteca *MICE* do R[4]. Nós escolhemos como método de imputação o PMM (*Predictive Mean Matching*). O PMM é um método que garante que os valores imputados sejam plausíveis [54]. Conforme mencionado, essa é uma abordagem conservadora que seguimos para reduzir os ruídos. Note que a imputação só foi realizada para o tempo de execução das tarefas. A contagem de erros não foi alterada. Acreditamos que o número de erros não foi afetado por possíveis falhas de procedimento, visto que essa contagem é independente de quando o tempo foi iniciado. Em resumo, de um total de 180 observações (seis tarefas, 30 participantes), nós imputamos 27 observações em tarefas disciplinadas e 21 em tarefas não disciplinadas. Após a imputação, nós realizamos uma análise dos dados, tanto por tipo de tarefa, quanto para os tipos de forma geral. Nós testamos as hipóteses introduzidas usando o método de análise de variância (*ANOVA*) [55], como será detalhado na Seção 5.4. Antes do teste de hipótese, nós verificamos se os dados satisfazem os pressupostos da ANOVA. Para isto, validamos os modelos usando a *Validação Global de Suposições de Modelos Lineares* [56], que se trata de um teste para suposições que podem restringir o uso da ANOVA: Linearidade, Homocedasticidade, Não Correlacionamento e Normalidade. Para testar essas premissas, nós usamos a biblioteca GVLMA do R[5] e encontramos que os modelos para a análise geral

[4] https://cran.r-project.org/web/packages/mice/index.html
[5] https://cran.r-project.org/web/packages/gvlma/index.html

das tarefas são aceitáveis. Porém, o quesito de normalidade falhou para a análise individual dos tempos de execução das tarefas. Diante disto, optamos por aplicar uma normalização utilizando $log_2(n+1)$ para a análise do tempo de execução das tarefas individuais. Como método para o teste de hipóteses, escolhemos o *Latin Square ANOVA*, visto que o modelos de análise geral e o de individual, após a normalização, satisfazem os pressupostos e que iremos testar a influência de apenas um fator por vez (tempo ou número de tentativas). Adicionalmente, calculamos o intervalo de confiança e o tamanho do efeito nas ocasiões em que a hipótese nula foi rejeitada. Optamos por calcular o intervalo de confiança utilizando o *Tukey (Tukey Honest Significant Test)*. O tamanho do efeito foi calculado utilizando *etha-squared* seguindo as recomendações vistas em Miles et al. [57]

Em resumo, a análise dos dados foi dividida em duas frentes: na primeira, nós analisamos o impacto da disciplinaridade por tipo de atividade; na segunda, nós consideramos o conjunto. Ao considerar o conjunto, estamos simulando o impacto da disciplinaridade em um dia de trabalho. Um dia de trabalho se refere a um dia em que o desenvolvedor tem que realizar um conjunto de tarefas. Ainda sobre o tratamento dos dados, ao separar os dados por tipo de atividade, eles pararam de se apresentar em uma distribuição normal. Nesse caso, para fazer o uso do ANOVA nós utilizamos uma normalização de dados baseada em $log_2(n+1)$

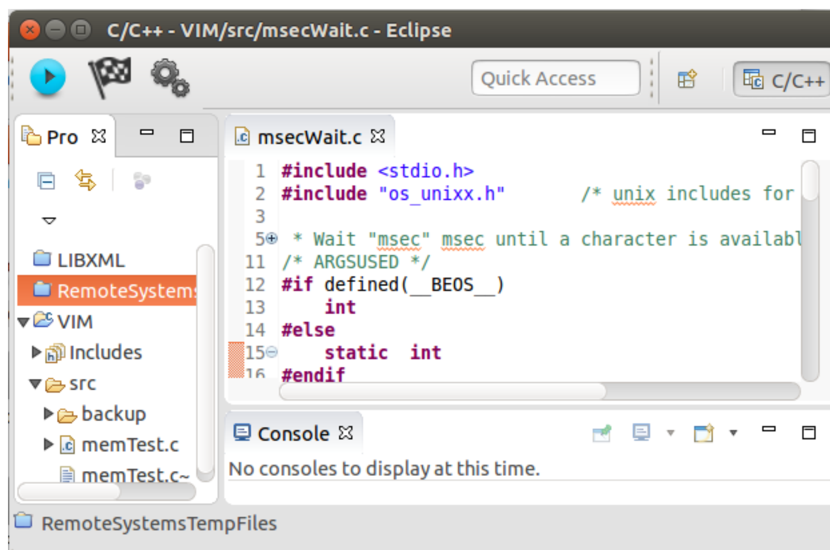

Figura 21 – *Plugin* para Eclipse. Note os botões do nosso *plugin* na esquerda (topo).

5.4 Resultados e Discussão

A análise dos resultados foi dividida em duas partes. A primeira parte consiste em verificar o impacto da disciplinaridade por tipo de tarefa. A segunda análise simula o impacto da

disciplinaridade em um dia de trabalho, portanto, considera para a análise o impacto da disciplinaridade os dados do conjunto de tarefas.

5.5 Primeira análise: Análise por tipo de tarefa

Para a análise individual de cada tipo de tarefa nós seguimos o mesmo modelo. Este consiste gerar *boxplots* comparando os resultados obtidos (tempo e número de erros) entre as diferentes disciplinaridades, um gráfico de densidade de diferenças de tempo e por fim a execução do teste de hipótese. Caso a hipótese nula seja rejeitada, nós calculamos o intervalo de confiança e o tamanho do efeito.

5.5.1 Tipo 1: Corrigir erro de sintaxe

O gráfico da Figura 22 mostra estatísticas descritivas em relação ao tempo total de execução da tarefa do Tipo 1 (Corrigir um erro de sintaxe). Como pode ser visto há sobreposição entre os dois *boxplots*. Além da sobreposição podemos notar que as medianas têm valores bem próximos (29 minutos para o tratamento disciplinado e 31 para o não disciplinado). Essa análise nos dá indícios de que o tratamento disciplinado tem um custo menor de tempo, porém essa diferença de tempo não aparenta ser significativa.

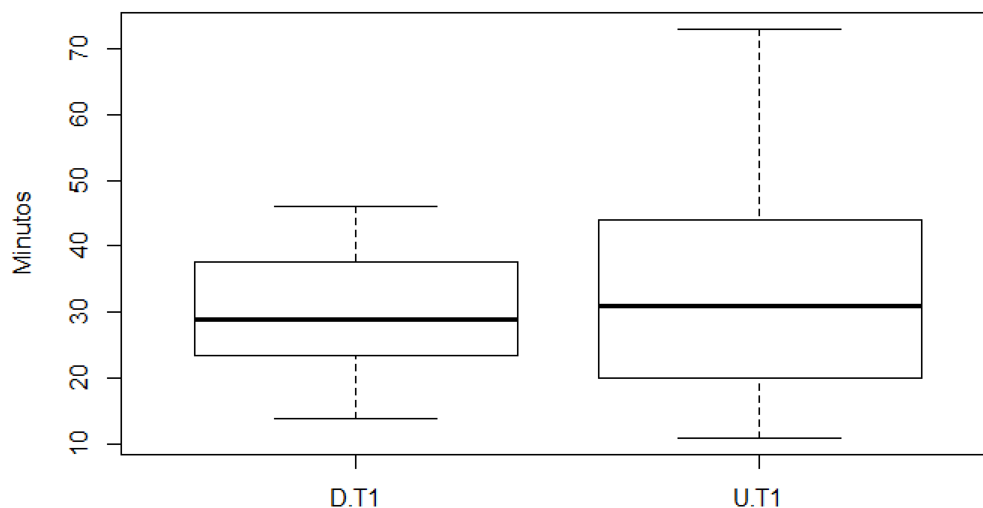

Figura 22 – Tempo necessário para concluir tarefas do Tipo 1. Onde “D” é relativo a disciplinado e “U” a não disciplinado.

A Figura 23 mostra um gráfico de densidade das diferenças entre o tempo das tarefas do Tipo 1 disciplinadas e não disciplinadas para cada participante ($T_{nd} - T_d$). Em outras

palavras, a área maior que zero representa os participantes que levaram mais tempo para responder as atividades no tratamento não disciplinado, enquanto os valores menores que zero apontam os que levaram mais tempo no disciplinado. Note que as áreas para ambos os lados são bastante próximas. Portanto, não parece que a disciplinaridade das anotações causa impacto na resolução de tarefas do Tipo 1.

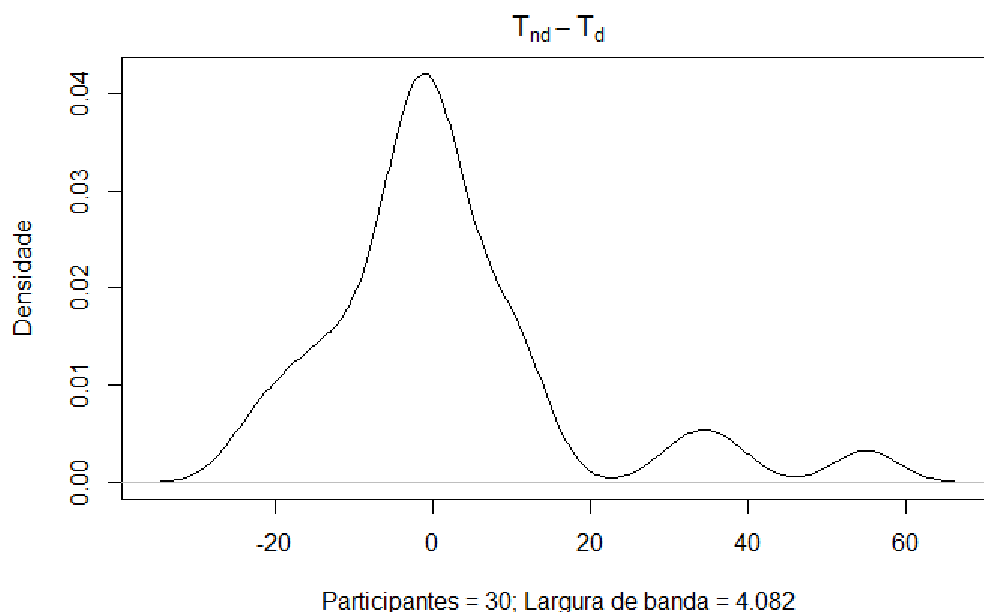

Figura 23 – Gráfico de densidade de tempo necessário para tarefas do Tipo 1.

Esse comportamento se repete quanto ao número de erros cometidos, como pode ser visto na Figura 27. Os dois *boxplots* também se sobrepõem e têm medianas próximas (quatro tentativas para o disciplinado e três para o não disciplinado). Contudo, dessa vez a mediana não disciplinada é ligeiramente inferior à disciplinada. Essa análise nos dá indícios de que não há diferenças significativas na quantidade de erros cometidos durante a execução de tarefas do Tipo 1, quando se usa os diferentes tipos de disciplinaridade.

Ao executar o ANOVA para cada hipótese ($H1_0$ e $H2_0$), não foram encontradas evidências para rejeitá-las. Este resultado está de acordo com a análise exploratória onde suspeitamos que não havia diferença significativa na execução entre os dois tratamentos.

Esse resultado contraria a expectativa inicial, de que haveria diferenças entre os dois tratamentos. Contudo, acreditamos que a motivação para a proximidade entre os tempos e números de erros foi a simplicidade da tarefa. Ao planejar a tarefa, nós procuramos encontrar um erro sintático que não exigisse muito conhecimento da linguagem C. Essa restrição é necessária, pois não temos a intenção de medir os conhecimentos do participante sobre a linguagem e sim o impacto dos dois tratamentos. Outro requisito é que a dificuldade fosse semelhante entre os dois tratamentos. Esse requisito diminui bastante o nosso escopo

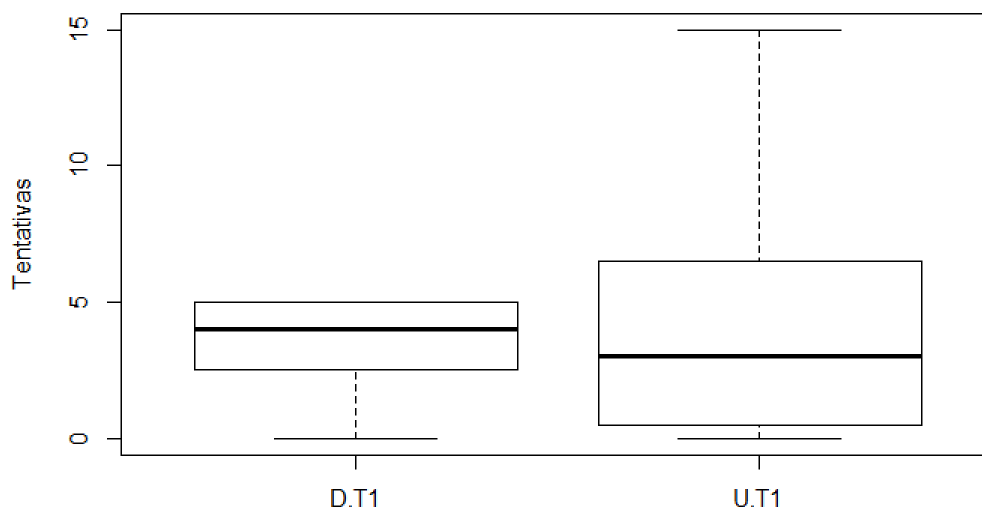

Figura 24 – Tentativas necessárias para concluir tarefas do Tipo 1. Onde “D” é relativo a disciplinado e “U” a não disciplinado.

de possíveis tarefas, visto que precisávamos escolher um erro sintático que servisse tanto para a forma disciplinada quanto para não disciplinada. A forma disciplinada possui um escopo menor de erros sintáticos que a forma não disciplinada. Um indício disso é que ela possui ferramentas automáticas que indicam erros de sintaxe enquanto as ferramentas para indicar erros na forma não disciplinada podem dar resultados incorretos [19]. Para cumprir esses dois requisitos, nós escolhemos um problema de balanceamento de parênteses. Com isso a complexidade da tarefa foi muito menor em relação às outras.

> *Baseado em uma análise exploratória dos dados e em um teste de hipótese, concluímos que a disciplinaridade das anotações em tarefas do Tipo 1 (corrigir erro de sintaxe) não torna a finalização da tarefa mais custosa.*

5.5.2 Tipo 2: Adicionar uma nova funcionalidade

O gráfico da Figura 25 mostra estatísticas descritivas em relação ao tempo total dos participantes para concluir as tarefas do Tipo 2. Algumas informações relevantes podem ser tiradas desse gráfico. Nele é possível visualizar uma leve sobreposição entre os *boxplots*. Esse gráfico nos dá indícios que o uso não disciplinado tem custo de tempo maior que o disciplinado. Note também que a mediana do tratamento não disciplinado é cerca de 20% maior (34 minutos) que o disciplinado (26 minutos). Adicionalmente, a maior parte do *boxplot* não disciplinado se encontra acima da mediana do disciplinado.

A Figura 26 nos mostra o gráfico de densidade para o tempo de execução das tarefas do

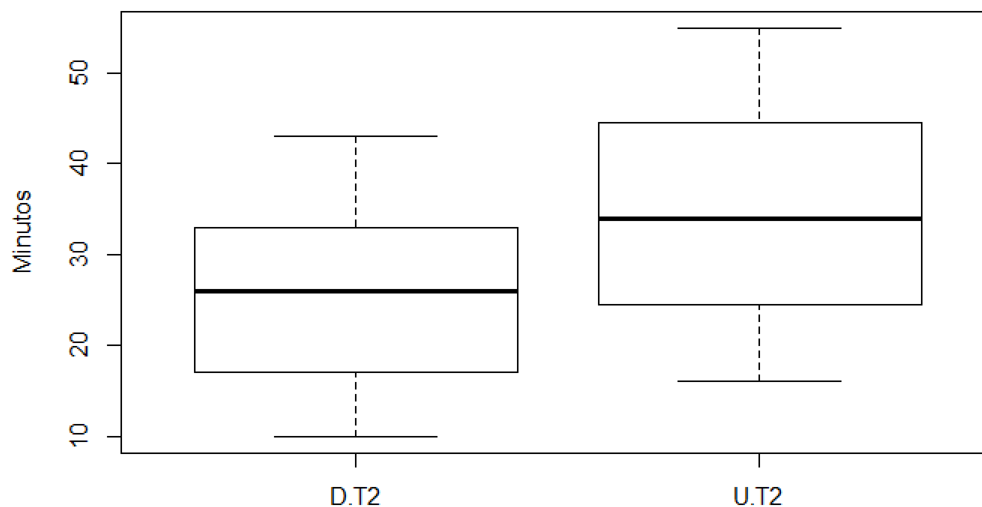

Figura 25 – Gráfico de tempo necessário para concluir tarefas do Tipo 2. Onde “D” é relativo a disciplinado e “U” a não disciplinado.

Tipo 2. Ele nos dá indícios que de os participantes realizaram as tarefas disciplinadas em um tempo menor (a área com valores maiores que zero é maior). Nele é possível observar também a presença de alguns *outliers* que realizaram tarefas não disciplinadas em um tempo muito menor que as disciplinadas. Contudo, mesmo com esses gráficos apontando que o uso de anotações disciplinadas consome menos tempo para tarefas do Tipo 2, $H1_0$ não foi rejeitada. Portanto, não foi possível encontrar diferenças significativas de tempo para a finalização de tarefas do Tipo 2 nas diferentes disciplinaridades.

A Figura 27 representa o número de tentativas para a finalização das tarefas do Tipo 2. Nela é possível notar que as medianas entre o *boxplot* disciplinada e a não disciplinada estão muito distantes, cerca de 110% (quatro para a disciplinada e nove para a não disciplinada). Adicionalmente existe pouca sobreposição entre os *boxplots*, o que nos traz indícios que utilizar o tratamento disciplinado diminuiu a quantidade de erros em relação ao outro.

Ao executar o teste de hipótese (ANOVA) encontramos evidências para rejeitar a hipótese $H2_0$ (p$-value = 0,0184 < 0,05 = \alpha$). Em relação ao tamanho do efeito, temos um valor moderadamente grande ($0,06 < \eta^2 = 0,092 < 0,14$). Já em relação ao intervalo de confiança, para 95% de confiança, obtivemos um p$-value = 0,0184113$ confirmando que a variação é significativa.

Um fato interessante que ocorreu na execução das tarefas do Tipo 2 é que oito dos 30 participantes resolveram disciplinar a tarefa antes de prosseguir com a realização da tarefa. Também houve um participante que transformou de disciplinada para não disciplinada.

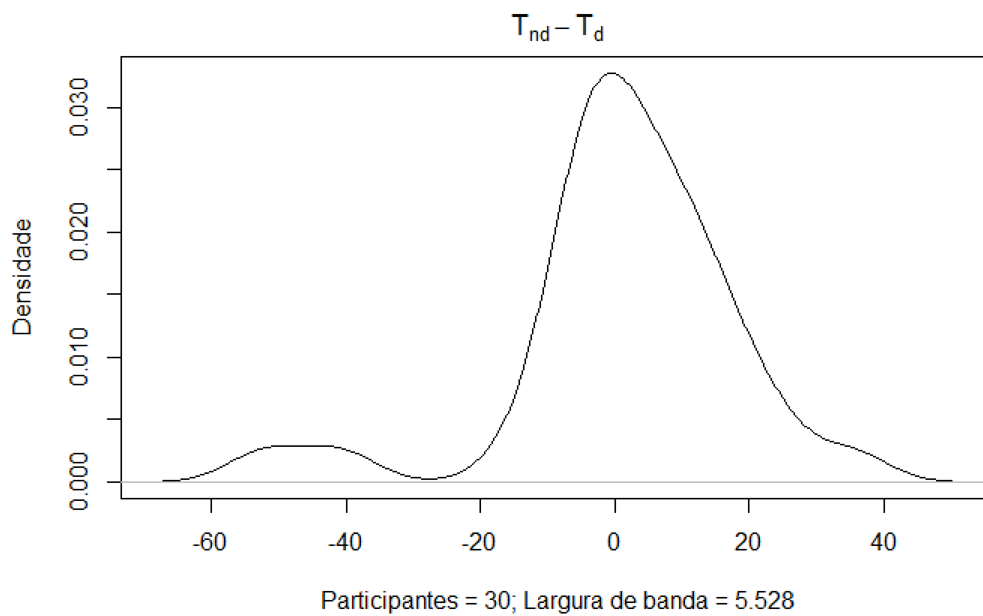

Figura 26 – Gráfico de densidade de tempo necessário para tarefas do Tipo 2.

Infelizmente não é possível estimar o impacto dessa estratégia utilizada pelos participantes, pois, não conseguimos inferir a partir de que momento o código foi refatorado. Contudo, os participantes que disciplinaram seu código, tiveram tempos semelhantes e algumas vezes inferiores aos que receberam o tratamento disciplinado desde o início.

Baseado em uma análise exploratória dos dados e em um teste de hipótese, concluímos que o uso de anotações não disciplinadas, ao realizar tarefas do Tipo 2 (adicionar uma nova funcionalidade), torna o código mais susceptível a erros.

5.5.3 Tipo 3: Corrigir um erro de comportamento

O gráfico da Figura 28 mostra estatísticas descritivas em relação ao tempo total dos participantes para concluir as tarefas do Tipo 3. Algumas informações importantes podem ser tiradas desse gráfico. Por exemplo, existe uma diferença expressiva entre as medianas dos dois tratamentos, a mediana para a resolução de tarefas não disciplinadas corresponde a mais que o dobro da disciplinada (23 minutos para a disciplinada e 62 minutos para a não disciplinada). Adicionalmente, não existe sobreposição entre os dois *boxplots*. Esse comportamento nos dá fortes indícios de que realizar a tarefas do Tipo 3 no tratamento não disciplinado é mais custoso que no outro.

O gráfico de densidade, que pode ser visto na Figura 29, nos dá indícios também de que a forma não disciplinada foi mais custosa. Note que boa parte da área se encontra a entre zero e 40.

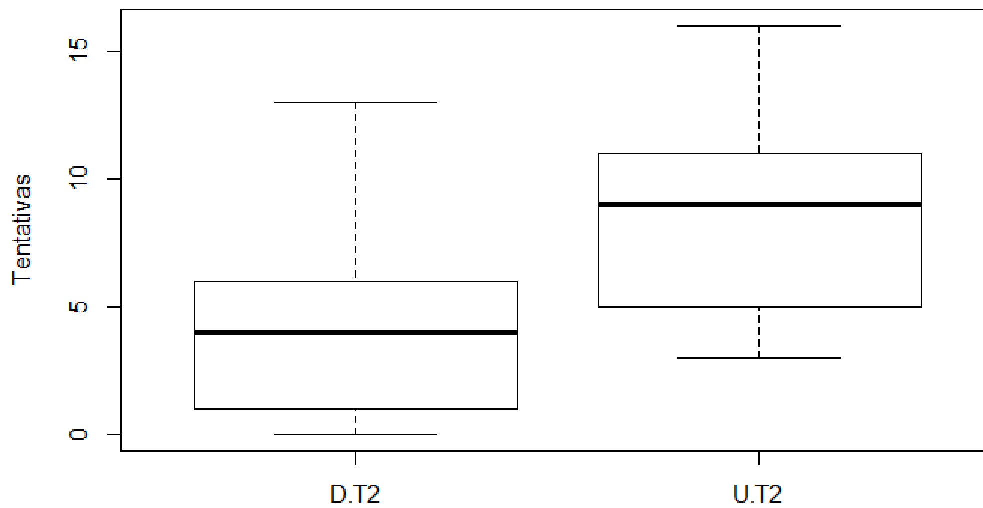

Figura 27 – Tentativas necessárias para concluir tarefas do Tipo 2. Onde “D” é relativo a disciplinado e “U” a não disciplinado.

O teste de hipótese rejeitou $H1_0$ (p$-value = 1.59e-05 < 0,001 = \alpha$). Em relação ao tamanho do efeito, temos um valor grande ($\eta^2 = 0,277 > 0,14$). Já em relação ao intervalo de confiança, para 95% de confiança, obtivemos um p$-value = 1,59e-05$, isso nos leva a concluir que tarefas do Tipo 3 são mais custosas na forma não disciplinada.

A Figura 30 mostra estatísticas descritivas em relação ao número de tentativas para finalização das tarefas do Tipo 3. Nela é possível notar um comportamento semelhante ao anterior (Figura 28), i.e. os *boxplots* não se sobrescrevem e existe uma diferença significativa entre as medianas (três e 13 tentativas).

Ao testar a hipótese $H2_0$, nós encontramos evidências para rejeitar a hipótese nula (p$-value = 9,84e-05 < 0,001 = \alpha$). Em relação ao tamanho do efeito, temos um valor grande ($\eta^2 = 0,232 > 0,14$). Já em relação ao intervalo de confiança, para 95% de confiança, obtivemos um p$-value = 0,0184113$, isso nos leva a concluir que executar tarefas do Tipo 3 na forma não disciplinada deixa o participante mais susceptível ao erro.

> *Baseado em uma análise exploratória dos dados e em dois testes de hipótese, concluímos que o uso de anotações não disciplinadas, ao realizar tarefas do Tipo 3 (corrigir erro de comportamento), faz com que as tarefas custem mais tempo e tornam o código mais susceptível a erros.*

As tarefas do Tipo 3 foram consideradas as mais difíceis. Talvez por conta dessa natureza foi possível encontrar as maiores diferenças de tempo e tentativas entre os dois

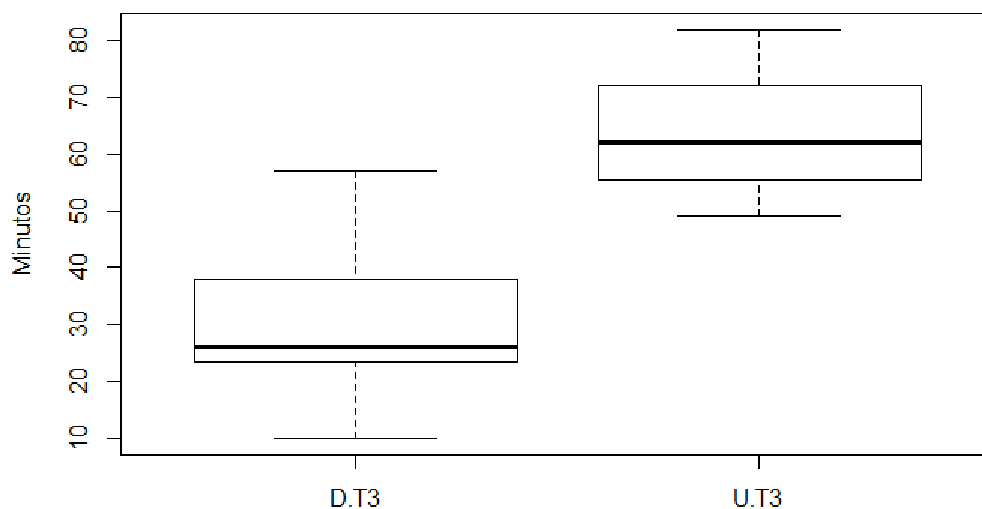

Figura 28 – Gráfico de quantidade de tempo necessário para conclusão de tarefas do Tipo 3. Onde "D" é relativo a disciplinado e "U" a não disciplinado.

tratamentos. A partir desse resultado, nós acreditamos que o impacto da disciplinaridade se correlaciona à dificuldade de execução da tarefa que está sendo executada. A partir dos dados que temos, não é possível provar alguma relação causal.

5.6 Segunda análise: Análise de um dia de trabalho.

O gráfico da Figura 31 mostra estatísticas descritivas em relação ao tempo total dos participantes para concluir todas as tarefas. Algumas informações relevantes podem ser tiradas dessa figura. Por exemplo, a mediana do total quando se usa anotações não disciplinadas (63 minutos) é quase 90% maior que quando se usa anotações disciplinadas (35 minutos). A maioria das observações relacionadas ao tempo de concluir todas as tarefas na forma disciplinada estão entre 30 e 52,75 minutos (o primeiro e terceiro quartis, respectivamente); em contraste, a maioria dos casos não disciplinados está entre 49 e 72.50 minutos. Quase não há sobreposição entre os *boxplots*, o que nos dá algumas evidências de que realizar tarefas de manutenção na forma não disciplinada consome mais tempo.

A Figura 32 mostra um gráfico de densidade das diferenças entre o tempo das tarefas no tratamento disciplinado e o não disciplinado ($T_{nd} - T_d$). Em outras palavras, a figura mostra a diferença de tempo gasto em tarefas disciplinadas e não disciplinadas para cada participante. Note que a maioria dos participantes gastaram mais tempo resolvendo as tarefas no modo não disciplinado. Isso pode ser notado pois a maioria das diferenças resultaram em um valor positivo ($T_{nd} > T_d$). Contudo, alguns participantes, sete (23%)

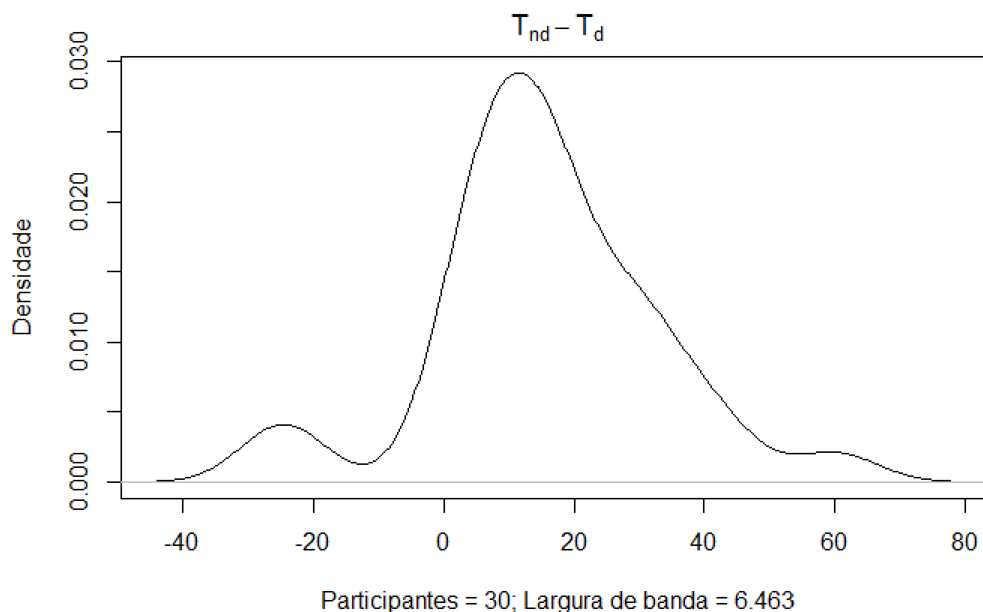

Figura 29 – Gráfico de densidade de tempo necessário para tarefas do Tipo 3.

dos 30, concluíram as tarefas não disciplinadas em menos tempo que as disciplinadas. A mediana da diferença para esses casos é de 17 minutos. Outro ponto interessante é que um participante levou 43 minutos a menos para responder utilizando anotações não disciplinadas.

Além da analise exploratória, nós testamos a primeira hipótese ($H1_0$) usando a técnica de análise de variância (ANOVA). Nós testamos nossa primeira hipótese nula e encontramos evidências para rejeitá-la (p$-value = 0,004 < 0,05 = \alpha$). Em relação ao tamanho do efeito, temos um valor moderadamente grande ($0,06 < \eta^2 = 0,101 < 0,14$). Já em relação ao intervalo de confiança, para 95% de confiança, obtivemos um p$-value = 0,0132197$, temos, assim, a primeira conclusão da segunda análise.

Baseado em uma análise exploratória dos dados e em um teste de hipótese, concluímos que o uso de anotações não disciplinadas faz com que o tempo para realizar tarefas de manutenção com o tratamento não disciplinado seja maior que o tempo gasto em um tratamento disciplinado.

Nós seguimos uma abordagem similar para testar a segunda hipótese, relacionada a checar se o desenvolvedor está mais susceptível a errar ao fazer manutenção em um código não disciplinado. Primeiramente, faremos uma análise exploratória dos dados. A Figura 33 mostra um gráfico com estatísticas descritivas relacionadas ao número de erros cometidos por cada participante (em outras palavras, o número de tentativas até o participante conseguir o resultado esperado). Note que a mediana para o tratamento

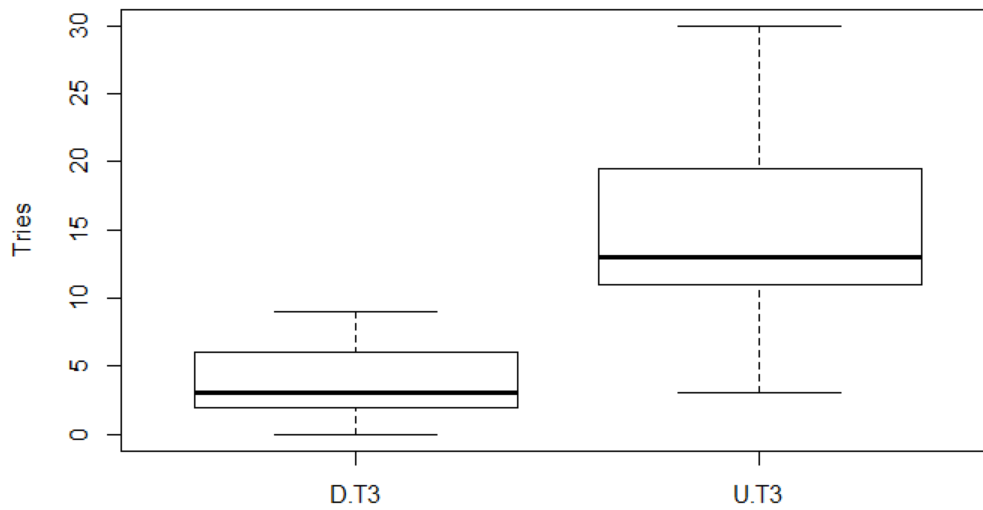

Figura 30 – Tentativas necessárias para concluir tarefas do Tipo 3. Onde “D” é relativo a disciplinado e “U” a não disciplinado.

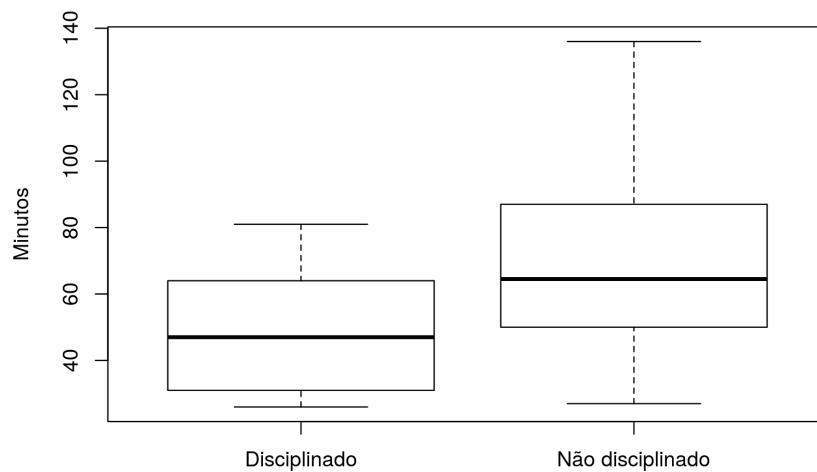

Figura 31 – Tempo total para concluir todas as tarefas.

de anotações não disciplinadas é quase 300% (15.00 tentativas) maior que o número de erros utilizando anotações disciplinadas (4.50 tentativas). Contudo, diferentemente da Figura 31, aqui encontramos os *boxplots* se sobrepondo moderadamente. Note que a maioria das observações relacionadas ao tratamento disciplinado se encontra entre 3,00 e 10,75 (primeiro e terceiro quartis, respectivamente). Em contraste, a maioria das observações

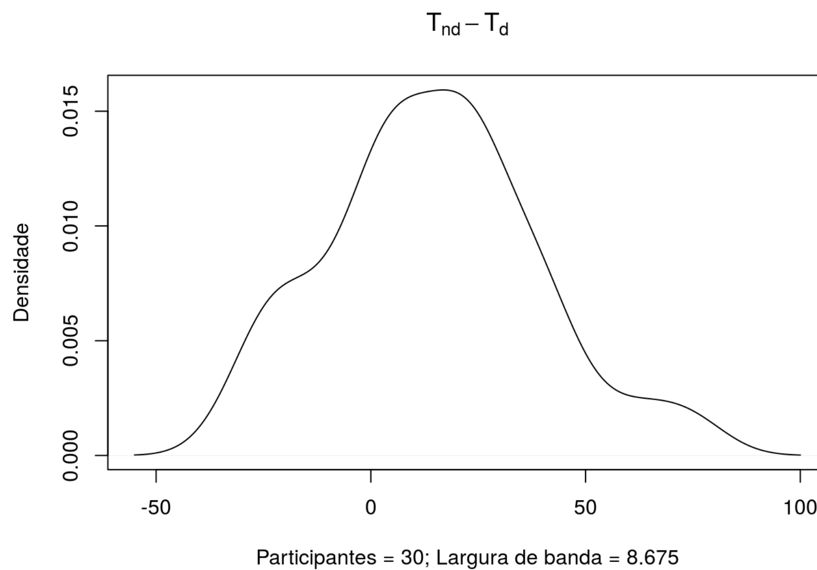

Figura 32 – Gráfico de densidade considerando a diferença de tempo entre $(T_{nd} - T_d)$, onde T_{nd} é o tempo gasto por um participante para resolver tarefas na forma não disciplinada e T_d é o tempo gasto pelo mesmo participante para responder na forma disciplinada.

não disciplinadas se encontram entre 6.00 e 23.25. Consequentemente, não podemos inferir uma conclusão sólida em relação à nossa segunda hipótese apenas com a observação da Figura 33.

Sete[6] Participantes (23%) dos 30 submeteram mais respostas quando estavam utilizando anotações disciplinadas. Os demais submeteram mais respostas na forma não disciplinada. Isso sugere que fazer manutenção em códigos com anotações não disciplinadas é mais susceptível a erros que um código com anotações disciplinadas. Para melhor checar essa afirmação, testamos a segunda hipótese $(H2_0)$ usando a ANOVA. O modelo de regressão nesse caso satisfaz as premissas da ANOVA. Nós testamos a hipótese nula e encontramos evidências para rejeitá-la (p$-value = 0,0001 < 0,05 = \alpha$). Em relação ao tamanho do efeito, temos um valor grande ($\eta^2 = 0,207 > 0,14$). Já em relação ao intervalo de confiança, para 95% de confiança, obtivemos um p$-value = 0,0002572$, temos, assim, o que nos leva à segunda conclusão desta análise.

> *Baseado no teste de hipótese da ANOVA, nós concluímos que o uso de anotações não disciplinadas é mais susceptível a erros que o uso de anotações disciplinadas, quando se está realizando tarefas de manutenção.*

[6] Esses sete participantes não são necessariamente os mesmos sete participantes citados na discussão da primeira hipótese. O número é somente uma coincidência.

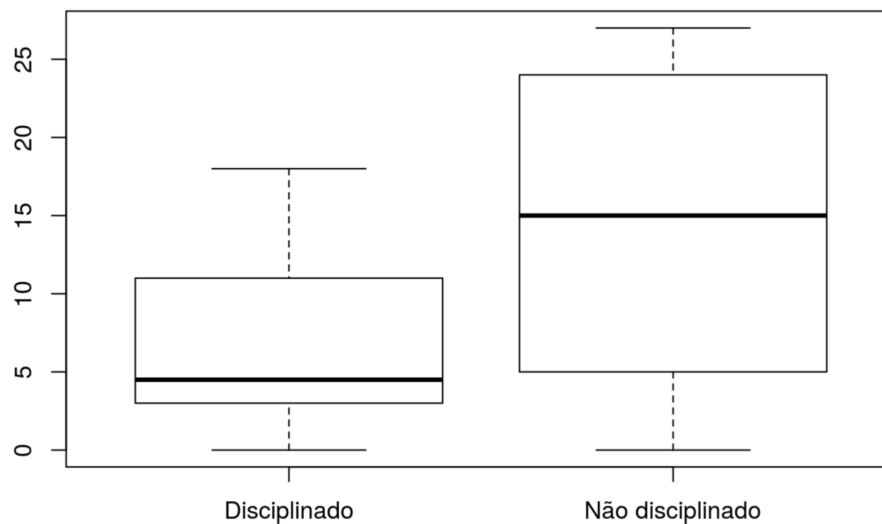

Figura 33 – Número de tentativas para concluir todas as tarefas.

5.7 Ameaças à Validade

Nessa seção, consideramos a classificação de Wohlin et al. [58]. Primeiramente, focaremos na validade de conclusão. Nossas conclusões se baseiam no poder estatístico elevado de modelos regressão que satisfazem todas as premissas para uso da ANOVA. Assim, nós mitigamos duas possíveis fontes de ameaças: baixo poder estatístico e violações em suposições de testes estatísticos. A confiabilidade das medidas também diz respeito à validade da conclusão. Uma possível ameaça ao nosso experimento está relacionada aos procedimentos que os participantes deveriam ter seguido para medir o tempo de conclusão das tarefas. Provavelmente alguns participantes não seguiram corretamente os procedimentos e não iniciaram o cronômetro no momento correto. Contudo, note que essa situação pode ocorrer em ambos os tratamentos, disciplinado e não disciplinado. Para mitigar essa ameaça, decidimos imputar todos os tempos suspeitos (medidas de tempo próximas ao primeiro quartil). Apesar de não estarmos, mostrando os resultados nesse trabalho, nós também fizemos uma análise dos dados sem utilizar as imputações e os resultados também levaram a evidências para rejeitar a hipótese nula do experimento. Note que **o procedimento de imputação não foi utilizado para o teste da segunda hipótese**, uma vez que não tivemos problemas em coletar os dados relativos às tentativas.

Existe uma certa validação cruzada entre os resultados deste estudo com o primeiro estudo reportado nesse trabalho, o qual é baseado em *pull requests*. Além disso nós tentamos evitar ao máximo possíveis fontes de ruídos. Nesse contexto, os resultados após aplicar

as imputações foram ligeiramente mais favoráveis ao tratamento não disciplinado (as tarefas tiveram seu tempo de conclusão diminuído). Mesmo assim obtivemos evidências que indicam que realizar tarefas de manutenção em anotações não disciplinadas consome mais tempo que em anotações disciplinadas. Essas decisões foram tomadas para mitigar possíveis ameaças à validade da construção.

Existe um debate extenso sobre validação interna vs. externa [58, 59]. Em resumo, o objetivo é estabelecer um relacionamento de causa e efeito envolvendo tratamentos e variáveis de resposta. Nesse caso, o uso de experimentos rigorosos é necessário e é realizado incluindo alunos para aumentar o número de réplicas. Isso dá suporte à validação interna do trabalho, embora dificulte a generalização dos resultados para diversas situações. Nesse sentido, nós não podemos generalizar os resultados do nosso experimento para outros cenários, visto que exploramos somente três cenários de manutenção envolvendo anotações de pré-processamento: corrigir um erro de sintaxe, introduzir uma nova funcionalidade, e corrigir um erro semântico numa configuração dada. Nós também não podemos generalizar o nosso resultado para desenvolvedores profissionais, visto que nosso experimento só utilizou alunos de graduação. Contudo, estudos argumentam que o uso de estudantes para a realização de experimentos controlados não chegam a ser prejudiciais ao resultado [60,61].

Uma ameaça em relação à análise dos dados é que parece que as tarefas do Tipo 3 foram o único motivo que fez com que os resultados fossem favoráveis ao tratamento disciplinado. Porém, fizemos mais alguns testes de hipótese para a validação do resultado obtido. Em um dos testes medimos, através do ANOVA, se o tipo de tarefa era um fator relevante para a variação de tempo e tentativas. Essa hipótese foi negada. Outro teste foi executar o ANOVA para as duas hipóteses sem a presença dos dados relativos às tarefas do Tipo 3. Nesse teste não foi possível rejeitar $H1_0$, porém, $H2_0$ continuou sendo rejeitada.

Uma outra ameaça surgiu na análise individual para as tarefas do Tipo 2. Ao rodar os pressupostos do ANOVA para testar a hipótese relativa ao tempo, o teste falhou para o *skewness*. Essa ameaça é reduzida pelo fato do *One Way ANOVA* ser robusto o suficiente para lidar com esse tipo de problema. Adicionalmente, quando o pressuposto do *skewness* falha, pode ocorrer problemas do Tipo 1. O que não é o caso, pois não foi possível rejeitar a hipótese nula. Por fim, ainda realizamos um outro teste de hipótese (*kurtosis*) e obtemos o mesmo resultado.

6 Trabalhos Relacionados

Neste capítulo apresentaremos os trabalhos relacionados. Focaremos em trabalhos que estudam o pré-processador C (Seção 6.1), que lidam com a disciplinaridade de anotações (Seção 6.2) e com refatoramentos de códigos com pré-processamento (Seção 6.3).

6.1 Pré-processador do C

O pré-processador do C é uma ferramenta que permite realizar processamento de texto em códigos fonte. Com isso, é possível criar variações no código. Para criar essas variações, ou variantes, o desenvolvedor precisa anotar o código com diretivas condicionais **`#ifdef`**, **`#elif`**, **`#else`**, e **`#endif`**. Caso a condição da diretiva seja verdadeira, o pré-processador irá adicionar aquele trecho de código anotado ao código final. Um conjunto de diretivas definidas é chamado de configuração. Um sistema que utiliza anotações condicionais é chamado de sistema configurável. A propriedade de fazer com o que o código varie de forma simples faz com que o pré-processador do C seja amplamente utilizado para resolver problemas de portabilidade e variabilidade. Apesar do seu amplo uso, essa ferramenta recebe várias críticas. Seu uso é relacionado a um aumento na susceptibilidade à introdução de erros (como erros de sintaxe, identificadores não declarados, vazamentos de memória, etc) [3, 15, 36, 37, 37–41] e obscurecimento do código [19, 44, 45], tornando as tarefas de manutenção e compreender o código mais difíceis [3, 46].

Spencer et. al [3] afirmaram que o impulso de utilizar o pré-processador do C para resolver problemas de portabilidade geralmente é um erro. Para eles, portabilidade é fruto de planejamento avançado ao invés de uma "*trincheira de guerra de* ***`#ifdef's`***," em seu estudo ele discute alternativas ao uso das anotações condicionais e quando elas deveriam ser utilizadas. Ele afirma, por exemplo, que utilizar diretivas para que o código se adapte entre dois sistemas é aceitável. Mas, em um número maior de sistemas os **`#ifdef's`** se proliferam, se aninham e se bloqueiam. Isso torna o código *uma bagunça ilegível e impossível de fazer manutenção*. Ernst et. al. [19] realizaram o primeiro estudo empírico sobre o uso das diretivas de pré-processamento do C. Esse estudo analisou cerca de 1,4 milhões de linhas de código. Nele foi constatado que o uso de diretivas de pré-processamento é complexo, potencialmente problemático ou inexprimível em relações a outras funcionalidades do C/C++. Ele afirma que o pré-processador permite manipulações arbitrárias de código que podem dificultar o entendimento de outros desenvolvedores ou ferramentas. Em sua conclusão, ele sugere aos desenvolvedores que não utilizem diretivas de forma arbitrária. Ele sugere que as utilize de forma que as ferramentas (seja de teste ou que envolva compreensão de código) não deem resultados incorretos. Em nosso estudo "**Contato**

com os Desenvolvedores: Minerando Repositórios" (Capítulo 4), nós encontramos algumas refatorações que foram feitas nesse sentido, de evitar que o pré-processador gere um resultado inesperado. Medeiros et al. [5] realizaram um estudo onde foram entrevistados 40 desenvolvedores. Além das entrevistas, foi realizada uma pesquisa com mais 202 desenvolvedores. Nesse estudo eles tentaram entender como os desenvolvedores se sentem em relação ao uso de diretivas condicionais. Entre outras perguntas, a ideia foi entender se na prática os desenvolvedores sofrem com os problemas que os críticos afirmam existir. Eles concluíram que de fato os problemas existem e que se tornam mais evidentes quando as anotações não são disciplinadas. Esses estudos [5, 19, 62] estão de acordo com os resultados do nosso trabalho, pois encontramos evidências que o uso não disciplinado de anotações condicionais é um fator que agrava os problemas já citados. Ainda em relação a Medeiros et al. [5], nosso estudo "**Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações**" (Capítulo 3) e o estudo "**Contato com os Desenvolvedores: Minerando Repositórios**" (Capítulo 4) convergiram com os resultados compilados da entrevistas realizadas. Ambos os resultados apontam que os desenvolvedores ficam mais "à vontade" em trabalhar com anotações disciplinadas.

Baxter et al. [4] realizaram um estudo empírico em mais de um milhão de linhas de código. Neste estudo, foi verificado que o uso de anotações condicionais em sistemas de grande longevidade geralmente acarreta em um problema de configurações mortas. Isso ocorre em virtude da necessidade de evoluir o sistema. Com o passar do tempo são adicionadas novas funcionalidades a ele. Com essas novas funcionalidades há novas configurações. Eventualmente as configurações mais antigas acabam se tornando mortas. Em outras palavras, obsoletas e sem uso. O ideal é que essas configurações sejam removidas. O processo de remover configurações mortas é tedioso e frustrante, o que acaba distanciando os desenvolvedores de executá-lo. Esse problema de remover configurações mortas é essencialmente um problema de remoção de código morto. Contudo, não é possível utilizar compiladores convencionais como se faz para remover códigos mortos, pois estes removem o código morto a partir do código objeto gerado. Quando o código objeto é gerado, todas as diretivas são perdidas. Esse cenário torna a manutenção de sistemas com anotações condicionais (também chamados de sistemas configuráveis) mais demorada e custosa do que os sem anotações. Esse estudo se relaciona com o "**Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações**" (Capítulo 3). No nosso estudo também foram analisados códigos de sistemas *open source* e, mesmo que com intenções diferentes, algumas de nossas descobertas convergiram. Por exemplo, nós também constatamos que alguns desenvolvedores possuem uma certa resistência para executar refatorações. Alguns desenvolvedores rejeitaram nossas sugestões por considerarem refatorações uma tarefa de menor importância em relação a outras. Outro ponto de convergência é que Baxter et al. [4] nos alerta sobre o aparecimento de configurações mortas. Isso se reflete no nosso estudo, pois, eventualmente criamos *pull requests* para códigos depreciados.

Abal et al. [62] realizaram um estudo para entender melhor a complexidade e a natureza de erros de variabilidade que ocorrem em sistemas altamente configuráveis. Para isso, eles analisaram o código do *Kernel do Linux*. Nesse estudo foram encontrados e estudados 42 erros relativos ao uso de anotações condicionais. Desses 42, 30 erros estavam relacionados à interação entre funcionalidades. Foi constatado que a variabilidade aumenta a complexidade dos erros. Outro ponto levantado, é que erros relacionados a variantes geralmente são tratados como erros normais. Em outras palavras, a correção é aplicada localmente e sem levar em conta o contexto. Esse tipo de correção é problemático, pois pode afetar alguma outra configuração que depende daquele trecho de código. Outro estudo [49] disserta sobre a dificuldade de se fazer manutenção em sistemas altamente configuráveis. Nesse estudo são analisadas falhas de interação e discutidas formas de identificá-las. Essas falhas ocorrem em função da capacidade de ativar ou desativar funcionalidades. Dependendo da configuração pode ocorrer que funcionalidades dependentes não estejam ativas ao mesmo tempo. Portanto, o sistema irá falhar. Esse tipo de falha pode ocorrer somente em algumas configurações do sistema. Identificar quais configurações geram falhas de interação é um processo de alta complexidade. Nem sempre é possível testar todas as configurações de um sistema altamente configurável. Nosso estudo também faz uma análise a respeito de erros relacionados ao uso de anotações condicionais. Em nosso "**Experimento Controlado**" (Capítulo 5) nós consideramos a corretude contando quantas vezes os participantes erraram. Na tarefa em que se pede para corrigir um erro de comportamento do experimento é simulado um cenário de manutenção em um código com interação entre funcionalidades. Ainda sobre a complexidade de erros em sistemas configuráveis. Medeiros et al. [63] detectaram erros de variáveis e funções não declaradas, estes erros estão relacionados ao uso de anotações condicionais. Em nosso "**Contato com os Desenvolvedores: Minerando Repositórios**" (Capítulo 4) foi possível encontrar relatos de desenvolvedores que fizeram correções no código a fim de remover avisos de variáveis ou funções sem uso.

6.2 Disciplinaridade das Anotações

Uma anotação não disciplinada é uma anotação que não engloba subárvores inteiras da árvore abstrata correspondente [17]. Muitos estudos relacionam o uso desse tipo de anotações a um aumento de susceptibilidade à erros [15, 19, 37, 64], diminuição da manutenibilidade e compreensibilidade do código [18,19] e limitações no suporte ferramental (as ferramentas não conseguem lidar corretamente com esse tipo de disciplinaridade) [18, 48, 65, 66].

Medeiros et al. [16] realizaram um estudo onde sugere um catálogo de refatorações. Esse contém formas de transformar anotações não disciplinadas em disciplinadas. Nesse mesmo estudo foi verificada também a aplicabilidade e aceitabilidade deste catálogo em

projetos *open source*. A fim de testar a aplicabilidade do catálogo, foram analisados 63 projetos em C de diferentes tamanhos. Nestes, foram detectadas 5670 oportunidades de aplicação. Para testar a aceptabilidade das refatorações, foram submetidos 28 *pull requests* para 23 projetos *open-source*. Ainda neste estudo foram entrevistados 246 desenvolvedores onde foram feitas seis perguntas relativas a um trecho de código nas duas disciplinas. Essa entrevista revelou que a maioria dos desenvolvedores entrevistados preferiam a versão disciplinada. Os resultados se refletem na taxa de aceitação das refatorações que eles sugeriram. Este estudo serviu como base para o nosso. Em nosso estudo "**Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações**" (Capítulo 3) ampliamos a abordagem dos *pull requests*. Aumentamos a população de 28 *pull requests* para 110 e assim ampliando a confiabilidade dos resultados encontrados. Assim como em Medeiros et al. [16], em nosso "**Contato com os Desenvolvedores: Minerando Repositórios**" (Capítulo 4) também analisamos os códigos de projetos. Contudo, em nossa análise, procuramos encontrar transformações realizadas por desenvolvedores.

Medeiros et al. [37] realizaram um estudo empírico em 41 *releases* de famílias de *software* e 51 mil *commits*, a fim de entender melhor como surgem os erros de sintaxe causados por anotações condicionais. Foi detectado que na maioria dos casos esses são introduzidos quando o desenvolvedor adiciona diretivas no código existente. Por exemplo ao adicionar suporte para um novo sistema. Esse resultado reforça o que foi afirmado por Spencer et al. [3]. Outra descoberta relevante é que erros de sintaxe não são comuns em famílias de *software*, tanto em *releases* quanto em *commits*. Porém, dos sete erros encontrados em *releases* seis (85.71%) estavam relacionados ao uso de anotações não disciplinadas. Em nosso "**Experimento Controlado**" (Capítulo 5), também verificamos que os desenvolvedores costumam cometer mais erros quando estão lidando com anotações não disciplinadas.

Outro indício que aponta o uso de anotações não disciplinadas como uma má prática pode ser visto em guias de desenvolvimento (*Code guidelines*). Em alguns desses guias é possível encontrar afirmativas que advogam contra o uso desse tipo de anotações. Por exemplo, o guia de desenvolvimento do Linux "*Prefira compilar funções inteiras ao invés de partes de funções ou expressões. Ao invés de colocar um ifdef em uma expressão, fatore parte ou ela toda em uma função separada e aplique os condicionais àquela função.*"[1] Porém, mesmo com guias de desenvolvimento com afirmativas contra o uso de anotações não disciplinadas, estudos verificaram que essas regras nem sempre são seguidas [15, 19]. Liebeg et al. [15] realizaram um estudo empírico onde analisou mais de 30 milhões de linhas de código. Nele foi verificado que 84% das diretivas, de 40 projetos de código aberto, são disciplinadas. Esse é um indício que os desenvolvedores se importam com a disciplinaridade das anotações. Neste estudo é levantada uma discussão sobre como esses

[1] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/coding-style.rst#n1004

16% restantes deveriam ser tratados. Uma das abordagens seria criar ferramentas que pudessem lidar com anotações não disciplinadas. Outra abordagem é disciplinar essas anotações. Em nosso estudo "**Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações**" (Capítulo 3) colocamos à prova a abordagem de disciplinar as anotações não disciplinadas e constatamos uma alta taxa de aceitação. Ainda sobre a abordagem de disciplinar anotações, no nosso estudo "**Contato com os Desenvolvedores: Minerando Repositórios**" (Capítulo 4) encontramos *commits* onde os desenvolvedores disciplinaram anotações.

Apesar de todas essas críticas, Schulze et al. [17] realizaram um experimento controlado com 19 estudantes de graduação de um curso de Sistemas operacionais da Universidade de Magdeburg. Para o experimento, os 19 alunos foram divididos em dois grupos, onde um grupo iria resolver tarefas na forma disciplinada e o outro na não disciplinada. Cada grupo recebeu um total de sete tarefas. As seis primeiras eram iguais aos dois grupos (com disciplinaridades diferentes) e a sétima era distinta para cada grupo. As tarefas envolviam compreender e realizar manutenção em trechos de código. Como variáveis dependentes, eles definiram o tempo para conclusão da tarefa e sua corretude. Por corretude a tarefa podia ser "correta", "quase correta" e "incorreta". Schulze et al. [17] concluiu que não é possível encontrar diferenças observáveis no uso de anotações disciplinadas e não disciplinadas no contexto de manutenibilidade e compreensibilidade do código. Mesmo com o nosso experimento tendo foco nas mesmas variáveis, nós conseguimos alcançar resultados diferentes.

Outro estudo que defende a ideia de que a disciplinaridade não causa impacto na manutenção do código pode ser visto no trabalho de Fenske et al. [67]. Este trabalho, através da métrica de suscetibilidade a alterações do código que se baseia em número de *commits* e alterações de linhas de código, compara o impacto das diferentes disciplinaridades no desenvolvimento de código. Um código é considerado susceptível a alterações quando o mesmo sofre várias alterações no decorrer do desenvolvimento do projeto. Neste estudo ele comparou códigos que tivessem o mesmo nível de complexidade de diferentes disciplinaridades. Concluiu-se que é possível observar que a disciplinaridade afeta o desenvolvimento, porém, não o afeta de forma significativa. Esse resultado conflita com a ideia de que a disciplinaridade afeta a manutenibilidade do código. Essa conclusão se relaciona com parte do resultado obtidos no "**Experimento Controlado**" (Capítulo 5) do nosso trabalho, visto que, na análise individual dos resultados da primeira tarefa não conseguimos encontrar diferenças no desempenho para a solução das atividades entre as duas disciplinaridades. Contudo, não é possível extrapolar essa conclusão, pois, encontramos diferenças para as outras tarefas.

6.3 Refatoração em Código com Anotações Condicionais

Existem várias técnicas para refatoração de código C com anotações condicionais. Uma delas foi proposta por Baxter e Mehlich, o DMS. Este se trata de uma ferramenta de transformação de código fonte para C/C++ [4, 18]. Ela foca em engenharia reversa para pegar informações de *design* e facilitar tarefas de manutenção como a remoção de configurações mortas. Platoff et al. [68] também propôs uma ferramenta para refatorar programas em C. Contudo, essa ferramenta usa heurísticas e limita o desenvolvedor a utilizar somente anotações disciplinadas. Garrido et al. [69] criaram uma uma ferramenta (chamada CRefactory) que provê refatorações sem alterar o comportamento do código. Ela retira informações da árvore sintática abstrata como base para análises e transformações. Ela é capaz de renomear elementos de um programa e extrair funções. Essa ferramenta é capaz de lidar com anotações não disciplinadas. Para isso, ela cria uma representação interna da anotação e a transforma em uma anotação disciplinada.

Em outro estudo [2] foi explicado como a CRefactory disciplina as anotações. Para isso ele executa um processo chamado de *pseudo pré-processamento* através de uma ferramenta chamada P-Cpp. A P-Cpp disciplina a anotação criando versões completas das anotações. Ao criar essas versões completas, o código é duplicado. Por exemplo, no lado esquerdo da Figura 34 temos a anotação condicional quebrando uma parte da condição do `if`. Esse tipo de anotação é caracterizado como não disciplinada. Ao quebrar o `if` com a anotação ele ficou incompleto. Então, para completar ele, o P-Cpp cria as duas versões possíveis com ele completo. Nesse processo, o código é duplicado, como pode ser visto no lado direito da Figura 34. Essa abordagem que envolve duplicar código nem sempre é bem vista pelos desenvolvedores, pois ao duplicar código a manutenção ganha complexidade.

Medeiros et al. [1] desenvolveram uma estratégia para disciplinar anotações que não envolve duplicação. Para isso, eles classificaram nove tipos de ocorrência de anotações não disciplinadas e criaram regras de como as disciplinar. Essas regras foram escritas em um catálogo de refatorações. Por exemplo, a Figura 35 se trata de uma refatoração retirada do catálogo. Nessa situação o catálogo sugere que se crie uma variável de teste. Essa variável vai conter a condição do `if`. A Figura 35 ilustra essa transformação na abordagem do catálogo. O número de linhas entre o disciplinado e o não disciplinado é bem próximo. Por esse motivo utilizamos o catálogo para os estudos realizados neste trabalho. Ainda sobre o catálogo, Medeiros et al. [16] realizaram um estudo onde verificou que as refatorações sugeridas não alteram o comportamento do código.

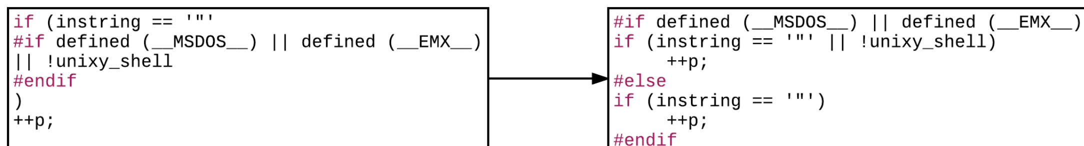

Figura 34 – Exemplo de transformação de um código não disciplinado (esquerda) em um disciplinado (direita) feito pelo P-Cpp. Exemplo extraído de Garrido et al [2]

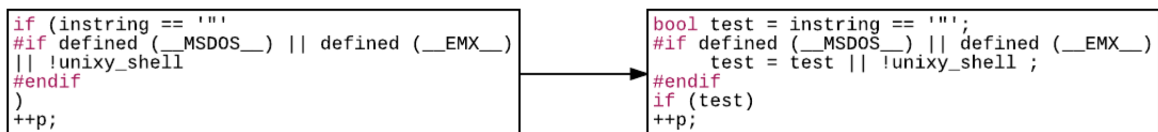

Figura 35 – Exemplo de transformação de um código não disciplinado (esquerda) em um disciplinado (direita) sugerido pelo catálogo de refatorações.

7 Conclusão

Nesse trabalho foram conduzidos três estudos a respeito da disciplinaridade de anotações de pré-processamento. No primeiro nós submetemos 110 *pull requests* para disciplinar anotações não disciplinadas que encontramos em projetos de código aberto. Até então, somente 11 submissões não receberam algum tipo de resposta. Quase dois terços (63%) dos *pull requests* foram aceitos pelos desenvolvedores dos sistemas. Contudo, por estarmos lidando com opiniões pessoais, nós recebemos algumas respostas que se contradizem (por exemplo, alguns desenvolvedores afirmam que disciplinar melhorou a compreensibilidade do código, enquanto outros não concordam). Por outro lado, 18% dos *pull requests* foram rejeitados por terem sido submetidos a códigos depreciados ou de terceiros. No entanto, ao perguntar aos desenvolvedores se essas alterações teriam sido aceitas caso tivessem sido submetidas para trechos de código em uso e que integram o projeto principal, 50% dos desenvolvedores responderam que aceitariam. Logo, atingimos uma taxa de aceitação de 71%. Isso significa que a maioria dos desenvolvedores que entramos em contato se importam com a disciplinaridade das anotações de pré-processamento. Ou seja, a maioria dos desenvolvedores concorda que a disciplinaridade das anotações é importante e não deve ser negligenciada.

No segundo estudo, nós mineramos código de repositórios *open-source* para tentar entender se os desenvolvedores disciplinam código (1), como ocorre o processo de disciplinar (2) e qual a motivação (3). Nesse contexto, encontramos 90 transformações e entramos em contato com 40 desenvolvedores, onde apenas 19 responderam. Nesse estudo concluímos que os desenvolvedores disciplinam código e na maioria das vezes esse processo ocorre como uma refatoração que visa melhorar a qualidade do código. Essas transformações também podem ser um meio para corrigir o código ou evitar possíveis efeitos secundários do uso de anotações condicionais.

No terceiro estudo, nós conduzimos um experimento controlado para investigar possíveis *relacionamentos de causa* entre manter códigos não disciplinados e dois atributos de engenharia de *software*: produtividade e susceptibilidade a erros. Nesse sentido, nós medimos e analisamos o tempo total e o número total de erros durante a execução do nosso experimento, que envolvia 30 participantes que tiveram que implementar seis cenários típicos de manutenção que foram adaptados de sistemas reais. Como resultado da análise dos dados, encontramos evidências de que manter código com anotações não disciplinadas custa mais tempo e faz com que os desenvolvedores estejam mais susceptíveis a cometer erros, em comparação ao uso de anotações disciplinadas.

Em resumo, considerando todos os dados levantados pelos três estudos, nós temos

evidências para afirmar que: o desenvolvedor prefere código disciplinado (1); que ele disciplina o código (2); e o uso de anotações disciplinadas torna a realização de tarefas de manutenção menos custosas, em relação ao tempo, e menos suscetível a erros (3). Concluímos que a disciplinaridade das anotações é importante e não deve ser negligenciada. Portanto, a *TAG* do título deve ser mantida desativada. Contudo, gostaríamos de evidenciar a necessidade da realização de mais experimentos controlados, afim de validar os resultados que obtivemos.

Todos os dados e materiais usados nos estudos apresentados neste trabalho estão disponíveis online.[1] O objetivo é fornecer informações suficientes para que outros pesquisadores possam replicar nossos estudos.

7.1 Revisão das Contribuições

Esse trabalho tem como principais contribuições:

- Um *quasi-experimento* baseado em *pull requests* para um melhor entendimento sobre como os desenvolvedores se relacionam com a disciplinaridade das anotações condicionais. Este estudo complementa o estudo realizado por Medeiros et al. [16]. Nele, não só aumentamos a quantidade de *pull requests* como também fizemos contato com os desenvolvedores e realizamos uma análise qualitativa deste contato. Adicionalmente, nós fizemos uma análise mais aprofundada sobre quais os motivos que levaram os *pull requests* a serem recusados. Nesse estudo, verificamos que os desenvolvedores aceitam sugestões de disciplinar as anotações condicionais.

- Um estudo empírico baseado em minerar *commits* para um melhor entendimento sobre se os desenvolvedores refatoram as anotações condicionais. Este é um estudo inédito que gerou uma ferramenta capaz de identificar suspeitas de refatoração de código não disciplinado em disciplinado. Adicionalmente, identificamos um número razoável de anotações que foram disciplinadas e categorizamos as motivações para essas refatorações. Neste estudo conseguimos verificar que os desenvolvedores disciplinam o seu código.

- Um experimento controlado, que investiga o efeito da disciplinaridade das anotações condicionais, em termos de tempo de resposta e corretude em tarefas de manutenção. Esse estudo, apesar de ser baseado em um experimento controlado anterior [17], trouxe um *design* diferente. Adicionalmente, foi possível encontrar também resultados diferentes. Os resultados obtidos evidenciam a necessidade da realização de mais experimentos controlados para investigar o efeito da disciplinaridade nas tarefas de manutenção de código.

[1] https://sites.google.com/site//icpc2017easylab

7.2 Implicações

Nosso estudo sugere a importância da disciplinaridade das anotações e com isso temos as seguintes implicações:

- Realização de novos experimentos controlados: O resultado do nosso experimento controlado divergiu do realizado anteriormente [17], é necessário que esses resultados sejam revalidados.
- Ferramentas para auxílio ao desenvolvimento: Nossos resultados apontam que o desenvolvedor se preocupa com a disciplinaridade das anotações, portanto, é necessário que estas estejam preparadas para sugerir refatorações que disciplinem as anotações não disciplinadas.
- Refatoração de sistemas legado: Nossos estudos sugerem que código que utiliza anotações não disciplinadas é mais difícil de manter, portanto, é adequado que os códigos legado sejam disciplinados.
- Novos guias de codificação: Uma vez que a disciplinaridade das anotações não pode ser negligenciada, é importante que os guias de codificação se alinhem a essa ideia. Os guias de codificações devem amparar o uso de anotações disciplinadas.

7.3 Trabalhos Futuros

Como trabalhos futuros, temos:

- Realização de um novo experimento controlado. Até então foram publicados dois experimentos controlados relacionados à disciplinaridade das anotações condicionais. Os resultados desses experimentos não estão de acordo. Em Schulze et al. [17] temos que a disciplinaridade das anotações não causam impacto na compreensão e manutenibilidade do código. Já em Malaquias et al. [70] temos que a disciplinaridade causa impacto e não deve ser negligenciada.
- Buscar entender o caminho contrário. Acreditamos que um estudo sobre o uso de anotações não disciplinadas pode trazer alguns resultados interessantes. Algumas questões como: (1) Os desenvolvedores transformam código disciplinado em não disciplinado? (2) Qual a motivação para realizar tal alteração? (3) os desenvolvedores aceitariam sugestões de transformações de disciplinados para não disciplinados? Acreditamos que existe uma relação de *tradeoff* na escolha da disciplinaridade da anotação. Por exemplo, em alguns casos os desenvolvedores preferem usar uma anotação não disciplinada para diminuir o número de linhas de código. Entender

essa relação pode trazer respostas que aprimorem o desenvolvimento de catálogos de refatoração.

- Aprimorar a ferramenta para detecção de transformações. Nosso estudo "**Contato com os Desenvolvedores: Minerando Repositórios**" (Capítulo 4) desenvolveu uma ferramenta para detectar a ocorrência de transformações de anotações não disciplinadas em disciplinadas. A ferramenta que foi desenvolvida é bastante limitada. Ela não é capaz de identificar todos os tipos de transformações além de consumir muitos recursos. Seria interessante que ela fosse refeita a fim de realizar um estudo mais detalhado sobre as motivações de realizar transformações entre disciplinaridades, além da sua frequência de ocorrência.

- Verificar a aceitação do catálogo de refatorações. Em nosso estudo, "**Contato com os Desenvolvedores: Sugestões para Disciplinar Anotações**" (Capítulo 3), nós criamos vários *pull requests* a partir do catálogo de refatoração [1]. Contudo, nós utilizamos apenas parte das transformações sugeridas (refatorações 2, 4 e 6). Seria interessante ver se a taxa de aceitação se mantém para os outros tipos de refatorações.

A Apêndice

A.1 Lista de Sistemas que Foram Submetidos *Pull Requests*

Nome do projeto	**Domínio**	***Commits***	**Contribuidores**	***Forks***	***releases***
Opencryptoki	API	1.017	13	4	0
Qira	*Debugger*	1.279	39	190	1
Rebol	Linguagem	308	17	195	0
Simh	Simulador	2.100	23	107	43
Swoole	*Framework*	3.305	47	1.498	209
Zlib	Biblioteca	312	16	486	69
Xrdp	Gerenciador	2.783	39	197	4
Yabause	Emulador	3.248	12	55	0
H2o	Servidor HTTP	3.602	64	478	57
Rathena	Servidor de MMORPG	14.267	54	504	0
Rsyslog	Processador de Log	10.215	151	238	402
Torque	Gerenciador de recursos	9.102	40	83	7
Toxcore	Plataforma de comunicação	3.769	169	1.025	1
Open62541	Implementação do OPC	3.506	42	111	10
Phpredis	Extensão do Redis	1.619	73	1.340	55
Slurm	Gerenciador	29.638	128	168	303
Stb	Biblioteca	1.199	81	565	0
Tcpdump	Monitor de rede	4.178	71	275	170
Remmina	Cliente remoto	1.882	58	160	21
Smoothieware	Interpretador	3.334	60	486	0
Jq	Processador de JSON	1.068	66	455	0
Kamailio	Servidor SIP	24.319	128	243	170
Libelektra	*Framework*	8.640	26	33	28
Openwsn-fw	*Firmware*	3.360	35	141	9
Rrdtool-1.x	Banco de dados	2.749	70	154	78
Toxic	Cliente Tox	1.741	75	106	20
Urbit	Servidor	2.249	52	96	4
Pacemaker	Gerenciador de recursos	16.262	78	171	71
TauLabs	Piloto automático	16.144	90	386	9
Jemalloc	Implementação do Malloc	1.504	63	402	39
Libpcap	Interface do Libpcap	2.729	58	215	171
Librdkafka	Cliente	1.745	55	435	28
Luv	Libuv para Lua	689	27	64	21
Marco	Gerenciador	394	45	29	40
Mate-utils	Utilidades	346	21	15	29
Metasploit-payloads	*Framework* 1.955	48	136	1	
Miniupnp	UPnP IGD	1.268	50	161	11
Naev	Jogo	12.729	67	131	16
Nut	Ferramentas de rede	3.924	40	68	25
Archipelago	Armazenamento distribuído	1.239	11	13	21
Ali_kernel	*Kernel*	39	9	161	0
OSEmu	Emulador	315	9	44	0
IntWars	*Framework*	668	39	162	1
Mupen64	Emulador 4.371	45	57	0	
Ffi	Extensão Ruby	1.556	71	190	74

Glfw	Biblioteca	3.205	83	632	10
Cocos2d-x	Jogo	10.315	164	5.890	32
FFmpeg	Biblioteca	82.853	834	2.948	24
Rspamd	Filtro de *Spam*	8.856	33	80	96
Sysdig	Ferramenta	3.410	79	384	141
Bfgminer	Minerador	12+651	94	400	208
Fontforge	Editor de fontes	18.124	93	248	21
Redis	Servidor	6.047	214	7.853	188
Meridian59	Servidor	3.689	29	144	14
Contiki	Sistema operacional	12.041	148	1.706	16
Cleanflight	*Firmware*	5.059	190	1.430	34
Cgminer	Minerador	7.840	74	968	208
Librdkafka	Biblioteca	1.745	55	435	28
Elemental-ircd	*Daemon*	6.639	25	21	31
Flint2	Biblioteca	5.999	48	113	0
Egoboo	Jogo	3.810	8	8	1
Libevent	Biblioteca	3.751	92	1.364	47
Deadbeef	Reprodutor de música	6.916	85	71	37
Cmus	Reprodutor de música	2+042	82	176	55
RaceCapture-Pro_firmware	*Firmware*	2.894	9	16	69
HAL	Biblioteca	1.279	20	13	22
ADCore	Detector de área	4.387	26	33	7
Qfusion	Jogo	4.436	13	40	2
Ioq3	Jogo	2.909	43	749	0
Libereport	Biblioteca	1.719	29	19	91
Cronus	Emulador	562	30	81	1
Etlegacy	Jogo	6.160	52	61	12
Atheme	Serviço	8.683	74	126	65
Radare2	*Framework*	13.164	301	823	32
Paparazzi	Sistema de veículo	14.644	91	651	51
Collectd	*Daemon*	8.952	276	763	150
Ompi-release	Open MPI	24.836	84	83	68
Mpv	Reprodutor de vídeo	44.227	184	604	61
JohnTheRipper	Quebra senhas	13.238	74	268	14
DarkTable	Aplicativo de imagem	18.931	186	337	75
RetroArch	API	35.015	152	444	57
Wiredtiger	Gerenciador	18.592	19	156	122
Systemd	Ferramenta	27.825	698	845	165
Czmq	Vinculador de C	3.781	131	327	18
Freeradius-server	Servidor	23.289	99	422	98
Fossa	Biblioteca	808	12	123	4
Capstone	*Framework*	2.431	83	470	22
Lxc	Contêiner	4.721	232	564	88
Osgearth	Ferramenta	8.566	70	309	26
Flux-core	Gerenciador	5.092	9	14	7
Mdsplus	Gerenciador	13.786	13	9	1.046
Sqlite3.07.14	Biblioteca	1.057	124	147	0
Libwebsockets	Biblioteca	1.854	125	421	43
Composite	Sistema operacional	1.301	14	29	0
DO-CV	Biblioteca	1.684	3	4	3
Espruino	Interpretador	3.443	48	286	37
Gauche	*Engine*	9.587	24	44	88
Open-AVB	Rede	1.035	41	108	3
ZipArchive	Compressor	353	61	710	23

Boostorg	Construtor	11.307	85	111	75
Quick-Cocos2d-x	*Framework*	2.337	45	870	11
Contrail-vrouter	Roteador virtual	948	41	108	8
F1rmware	*Firmware*	827	49	107	0
FACS	Classificador	881	8	11	0
Forked-daapd	Servidor	3.679	23	99	20
Desura-app	Jogo	1.902	12	32	10
ModSecurity	*Firewall*	1.660	31	463	82
MoarVM	VM	6.914	62	77	36
Arduino_STM32	Arquivos de *Hardware*	643	32	214	0
EusLisp	Sistema	858	15	26	14
Civetweb	Servidor *web*	3.373	92	271	8
Cyassl	Implementação do TLS/SSL	2.390	19	72	53
Cnscrypt-proxy	Protetor de comunicações	2.016	46	361	30
Dump1090	Decodificador	386	17	516	0
Enable	Biblioteca	1.686	32	26	20
Fcitx	*Framework*	2.672	28	108	48
Galera	Biblioteca	3.619	18	64	16
Heimdal	Implementação do Kerberus	28.329	74	79	118

Tabela 5 – Caracterização dos sistemas.

A.2 Tarefas do Experimento

```
...
int cur_test = cur[0] == ':';
#ifdef SUPPORT_IP6
    cur_test = (cur_test && cur[3] != ' ');
    #if defined PF_INET
    cur_test = cur_test && (cur[1] != '/' ||
cur[4] != ' ');
    #else
    cur_test = cur_test && (cur[2] != '0' ||
cur[4] != ' ');
    #endif
#else
    cur_test = (cur_test && cur[3] != '-'));
    #endif
if (cur_test) {
    buf[index] = 0;
    index = 0;
    cur += 3;
    break;
}
```

```
...
if (cur[0] != ':' && (
    #ifdef SUPPORT_IP6
        cur[3] != ' '
    #if defined PF_INET || AF_INET
        &&
        #if defined PF_INET
            (cur[1] != '/'
        #else
            (cur[2] != '0'
        #endif
        || cur[4] != ' ')
    #endif
    #else
        cur[3] != '-')
    #endif
)) {
    buf[index] = 0;
    index = 0;
    cur += 3;
    break;
```

Figura 36 – Versão disciplinada (esquerda) e não disciplinada (direita) para a tarefa de corrigir um erro de sintaxe no *Libxml2*.

```
#ifdef HAVE_SPRINTF
 len = sprintf(buf, "PORT %d,%d,%d,%d,%d,%d\r\n",
       adp[0] & 0xff, adp[1] & 0xff, adp[2] & 0xff, adp[3] & 0xff,
       portp[0] & 0xff, portp[1] & 0xff);
#else /* HAVE_SNPRINTF */
 len = snprintf(buf, sizeof(buf), "PORT %d,%d,%d,%d,%d,%d, \r\n",
       adp[0] & 0xff, adp[1] & 0xff, adp[2] & 0xff, adp[3] & 0xff,
       portp[0] & 0xff, portp[1] & 0xff);
#endif /* HAVE_SNPRINTF */
```

```
#ifdef HAVE_SPRINTF
 len = sprintf(buf, "PORT %d,%d,%d,%d,%d,%d\r\n",
#else /* HAVE_SNPRINTF */
 len = snprintf(buf, sizeof(buf), "PORT %d,%d,%d,%d,%d,%d\r\n",
#endif /* HAVE_SNPRINTF */
             adp[0] & 0xff, adp[1] & 0xff, adp[2] & 0xff, adp[3] &
0xff,
       portp[0] & 0xff, portp[1] & 0xff);
```

Figura 37 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de adicionar nova funcionalidade no *Libxml2*.

```
#else
 returnValue += 5;
     #ifdef PF_INET
          #ifdef AF_INET
 returnValue = returnValue * ( 10 - xmlNanoFTPTestEntries(22, 30, 32) + port) * protocol;
          #else
 returnValue = returnValue * ( 10 - xmlNanoFTPTestEntries(22, 30, 32) + proxy * protocol);
          #endif
     #else
          #ifdef AF_INET
 returnValue = returnValue * ( 10 - xmlNanoFTPTestEntries(22, 31, 37) + port) * protocol;
          #else
 returnValue = returnValue * ( 10 - xmlNanoFTPTestEntries(22, 31, 37) + proxy * protocol);
          #endif
     #endif
#endif
```

```
#else
     5 * (10 - xmlNanoFTPTestEntries(22,
     #ifdef PF_INET
          30, 32)
          #ifdef AF_INET
               + port) * protocol
          #else
               - proxy + protocol)
          #endif
     #else
          31, 37)
          #ifdef AF_INET
               + port) * protocol
          #else
               - proxy + protocol)
          #endif
     #endif
#endif
```

Figura 38 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de corrigir um erro de comportamento no *Libxml2*.

```
...
#ifdef FEAT_XCLIPBOARD
 update_time = xterm_Shell != (Widget)0;
#ifdef USE_XSMP
 update_time = update_time || xsmp_icefd !=
-1;
#endif
#ifdef FEAT_MZSCHEME
 update_time = update_time ||
(mzthreads_allowed() && p_mzq > 0);
#endif
#else
#ifdef USE_XSMP
 update_time = xsmp_icefd != -1;
#ifdef FEAT_MZSCHEME
 update_time = update_time ||
(mzthreads_allowed() && p_mzq > 0);
#endif
#else
#ifdef FEAT_MZSCHEME
 update_time = ((mzthreads_allowed() &&
p_mzq > 0);
#endif
 if (update_time)
```

```
…
#ifdef FEAT_XCLIPBOARD
      xterm_Shell != 0
#if defined(USE_XSMP) ||
defined(FEAT_MZSCHEME)
      ||
#endif
#endif
#ifdef USE_XSMP
      xsmp_icefd != -1
#ifdef FEAT_MZSCHEME
      ||
#endif
#endif
#ifdef FEAT_MZSCHEME
  ((mzthreads_allowed() && p_mzq > 0)
#endif
      ))
```

Figura 39 – Versão disciplinada (esquerda) e não disciplinada (direita) para a tarefa de corrigir um erro de sintaxe no *VIM*.

```
int regexec_return = 0;
 #ifdef WIN_32
 regexec_return = vim_regexec(starts_with_dot, d_name, colnr_T[0]);
 #else
 regexec_return = vim_regexec(starts_with_dot, d_name, colnr_T[1]);
#endif
```

```
if ((d_name[0] != '.' || starts_with_dot)
     && vim_regexec(
     #ifdef WIN_32
          d_name[0] == '/'
     #else
          starts_with_dot
     #endif
     , d_name,
     #ifdef WIN_32
          colnr_T[0]
     #else
          colnr_T[1]
     #endif
))
     {
```

Figura 40 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de adicionar nova funcionalidade no *VIM*.

```
#else
  returnValue += 5;
 #ifdef USE_XSMP
  #ifdef FEAT_XCLIPBOARD
    returnValue = returnValue * ( 1 - memCheckCorrupted(2, 3, 3) + mem_index) * lc_jump;
  #else
  returnValue = returnValue * ( 1 - memCheckCorrupted(2, 3, 3) + signal * lc_jump);
 #endif
 #else
  #ifdef FEAT_XCLIPBOARD
   returnValue = returnValue * ( 1 - memCheckCorrupted(2, 2, 7) + mem_index) * lc_jump;
  #else
   returnValue = returnValue * ( 1 - memCheckCorrupted(2, 2, 7) + signal * lc_jump);
  #endif
 #endif
#endif
```

```
#else
     5 * (1 - memCheckCorrupted(2,
     #ifdef USE_XSMP
         3, 3)
         #ifdef FEAT_XCLIPBOARD
             + mem_index) * lc_jump
         #else
             - signal + lc_jump)
         #endif
     #else
         2, 7)
         #ifdef FEAT_XCLIPBOARD
              + mem_index) * lc_jump
         #else
              - signal + lc_jump)
         #endif
     #endif
#endif
;
```

Figura 41 – Versão disciplinada (acima) e não disciplinada (abaixo) para a tarefa de corrigir um erro de comportamento no *VIM*.

Referências

[1] F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonseca, “A catalogue of refactorings to remove incomplete annotations,” *Journal of Universal Computer Science*, vol. 20, no. 5, pp. 746–771, 2014.

[2] A. Garrido and R. E. Johnson, “Embracing the C preprocessor during refactoring,” *Journal of Software: Evolution and Process*, vol. 25, no. 12, pp. 1285–1304, 2013.

[3] H. Spencer and G. Collyer, “Ifdef considered harmful, or portability experience with C news,” in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1992, pp. 185–197.

[4] I. Baxter, “Design maintenance systems,” *Communication of the ACM*, vol. 35, no. 4, pp. 73–89, 1992.

[5] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, “The love/hate relationship with the C preprocessor: An interview study,” in *Proceedings of the European Conference on Object-Oriented Programming*. Schloss Dagstuhl, 2015, pp. 999–1022.

[6] C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines,” in *International Conference on Software Engineering (ICSE)*. ACM, 2008.

[7] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, “Can we refactor conditional compilation into aspects?” in *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, ser. AOSD ’09. ACM, 2009, pp. 243–254.

[8] M. Anastasopoulos and D. Muthig, *An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology*. Springer Berlin Heidelberg, 2004, pp. 141–156.

[9] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “A case study in refactoring a legacy component for reuse in a product line,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, 2005, pp. 369–378.

[10] C. Gacek and M. Anastasopoules, “Implementing product line variabilities,” in *Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context*. ACM, 2001, pp. 109–117.

[11] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho, “Extracting and evolving mobile games product lines,” in *Proceedings of the 9th International Conference on Software Product Lines*, ser. SPLC’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 70–81.

[12] M. Goedicke, K. Pohl, and U. Zdun, *Domain-Specific Runtime Variability in Product Line Architectures.* Springer Berlin Heidelberg, 2002, pp. 384–396.

[13] B. Zhang and M. Becker, "Recovar: A solution framework towards reverse engineering variability," in *2013 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, 2013, pp. 45–48.

[14] M. Volter, "Handling variability," in *14th annual European Conference on Pattern Languages of Programming*, EuroPLoP. Kelly & Weiss, 2009.

[15] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *Proceedings of the International Conference on Aspect-Oriented Software Development.* ACM, 2011, pp. 191–202.

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

[17] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: A controlled experiment," in *Proceedings of International Conference on Generative Programming: Concepts and Experiences*, 2013, pp. 65–74.

[18] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings of the Conference on Reverse Engineering.* IEEE, 2001, pp. 281–290.

[19] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, pp. 1146–1170, 2002.

[20] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," *SIGPLAN Not.*, vol. 51, no. 3, pp. 35–44, Oct. 2015.

[21] M. Vittek, "Refactoring browser with preprocessor," in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, ser. CSMR '03. IEEE Computer Society, 2003, pp. 101–.

[22] B. McCloskey and E. Brewer, "Astec: a new approach to refactoring c." vol. 30, pp. 21–30, 09 2005.

[23] C. Kästner, S. Apel, and M. Kuhlemann, "A model of refactoring physically and virtually separated features," in *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, ser. GPCE '09. ACM, 2009, pp. 157–166.

[24] S. Trujillo, D. Batory, and O. Diaz, “Feature refactoring a multi-representation program into a product line,” in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE ’06. ACM, 2006, pp. 191–200.

[25] K. Narasimhan and C. Reichenbach, “Copy and paste redeemed (T),” in *International Conference on Automated Software Engineering*. IEEE/ACM, 2015, pp. 630–640.

[26] J. Creswell and V. Clark, *Designing and Conducting Mixed Methods Research*. SAGE Publications, 2011.

[27] W. F. Opdyke, “Refactoring object-oriented frameworks,” Ph.D. dissertation, 1992, uMI Order No. GAX93-05645.

[28] J. Antony, *Design of experiments for engineers and scientists*. Elsevier, 2014.

[29] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, “The confounding effect of class size on the validity of object-oriented metrics,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 7, pp. 630–650, 2001.

[30] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.

[31] T. Hall, M. Zhang, D. Bowes, and Y. Sun, “Some code smells have a significant but small effect on faults,” *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014.

[32] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[33] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, “Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM ’10. IEEE Computer Society, 2010, pp. 1–10.

[34] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, “Quantifying the effect of code smells on maintenance effort,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[35] Y. Zhou, H. Leung, and B. Xu, “Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 607–623, 2009.

[36] J. M. Favre, "Understanding-in-the-large," in *Proceedings of the 5th International Workshop on Program Comprehension (WPC).* IEEE Computer Society, 1997, pp. 29–.

[37] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating preprocessor-based syntax errors," in *Proceedings of International Conference on Generative Programming: Concepts & Experiences.* ACM, 2013, pp. 75–84.

[38] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *Journal of Object Technology (JOT)*, vol. 8, no. 6, 2009.

[39] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of International Conference on Software Engineering*, 2014.

[40] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE).* IEEE Computer Society, 2001, pp. 281–.

[41] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *Proceedings of the 16th International Conference on Software Engineering.* IEEE Computer Society Press, 1994, pp. 49–57.

[42] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., 1978.

[43] J. Melo, C. Brabrand, and A. Wkasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 679–690.

[44] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A quantitative analysis of aspects in the ecos kernel," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems.* ACM, 2006, pp. 191–204.

[45] T. T. Pearse and P. W. Oman, "Experiences developing and maintaining software in a multi-platform environment," in *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Society, 1997, pp. 270–.

[46] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of International Conference on Software Engineering.* ACM, 2010, pp. 105–114.

[47] R. C. Seacord, *The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems*, 2nd ed. Addison-Wesley Professional, 2014.

[48] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *Proceedings of the International Conference on Software Maintenance.* IEEE, 2005, pp. 379–388.

[49] B. J. Garvin and M. B. Cohen, "Feature interaction faults revisited: An exploratory study," in *Proceeding of the International Symposium on Software Reliability Engineering.* IEEE, 2011.

[50] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. ACM*, vol. 21, no. 6, pp. 466–471, Jun. 1978.

[51] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for experimenters: design, innovation, and discovery*, 2nd ed. Wiley-Interscience, 2005, vol. Wiley series in probability and statistics.

[52] D. C. Montgomery, *Design and Analysis of Experiments.* John Wiley & Sons, 2006.

[53] D. R. Eric Clayberg, *Building Commercial-Quality Plug-ins.* Addison-Wesley Professional, 2006.

[54] N. J. Horton and S. R. Lipsitz, "Multiple imputation in practice," *The American Statistician*, vol. 55, no. 3, pp. 244–254, 2001.

[55] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

[56] E. A. Pena and E. H. Slate, "Global validation of linear model assumptions," *Journal of the American Statistical Association*, 2012.

[57] M. S. Jeremy Miles, *Applying Regression and Correlation: A Guide for Students and Researchers.* SAGE Publications, 2001.

[58] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[59] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1.* IEEE Press, 2015, pp. 9–19.

[60] D. I. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, "Conducting realistic experiments in software engineering," in *Empirical Software Engineering. Proceedings. International Symposium n.* IEEE, 2002, pp. 17–26.

[61] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects—a comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[62] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of International Conference on Automated Software Engineering*. IEEE/ACM, 2014.

[63] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," in *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*. ACM, 2015, pp. 35–44.

[64] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of ACM SIGPLAN Object-Oriented Programming Systems Languages and Applications*. ACM, 2011, pp. 805–824.

[65] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proceedings of International Workshop on Principles of Software Evolution*. ACM, 2002, pp. 6–14.

[66] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *Compiler Construction*. Springer, 2009, pp. 109–125.

[67] G. S. Wolfram Fenske, Sandro Schulze, "How preprocessor annotations (do not) affect maintainability: A case study on change-proneness," 2017, To appear.

[68] M. Platoff, M. Wagner, and J. Camaratta, "An integrated program representation and toolkit for the maintenance of C programs," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1991, pp. 129–137.

[69] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2003, pp. 323–326.

[70] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC. IEEE Press, 2017, pp. 297–307.