UNIVERSIDADE FEDERAL FLUMINENSE

EDUARDO SMIL PRUTCHI

COMO A ADOÇÃO DE *FEATURE TOGGLES* AFETA *MERGES* DE RAMOS E DEFEITOS EM PROJETOS DE *SOFTWARE* LIVRE?

Niterói

2018

UNIVERSIDADE FEDERAL FLUMINENSE

EDUARDO SMIL PRUTCHI

COMO A ADOÇÃO DE *FEATURE TOGGLES* AFETA *MERGES* DE RAMOS E DEFEITOS EM PROJETOS DE *SOFTWARE* LIVRE?

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Sistemas e Informação.

Orientador:
Prof. D.Sc. LEONARDO GRESTA PAULINO MURTA

Niterói
2018

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

P971c Prutchi, Eduardo Smil
Como a Adoção de Feature Toggles afeta Merges de Ramos e Defeitos em Projetos de Software Livre? : / Eduardo Smil Prutchi ; Leonardo Gresta Paulino Murta, orientador. Niterói, 2018.
74 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2018.

DOI: http://dx.doi.org/10.22409/PGC.2018.m.08048637766

1. Engenharia de software. 2. Software livre. 3. Código aberto (Computação). 4. Produção intelectual. I. Murta, Leonardo Gresta Paulino, orientador. II. Universidade Federal Fluminense. Escola de Engenharia. III. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

EDUARDO SMIL PRUTCHI

COMO A ADOÇÃO DE *FEATURE TOGGLES* AFETA *MERGES* DE RAMOS E DEFEITOS EM PROJETOS DE *SOFTWARE* LIVRE?

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Sistemas e Informação.

Aprovada em dezembro de 2018.

BANCA EXAMINADORA

Prof. D.Sc. Leonardo Gresta Paulino Murta – Orientador, IC/UFF

D.Sc. Troy Costa Kohwalter, IC/UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira, IC/UFF

Prof. D.Sc. Márcio de Oliveira Barros, UNIRIO

Niterói

2018

Aos amigos e colegas, pelo incentivo e pelo apoio constantes.

AGRADECIMENTOS

À minha esposa, Erica Cristina R. Prutchi, por todo companheirismo, apoio e paciência que sempre teve comigo durante o tempo deste curso. Aos meus filhos queridos, Arthur e Gustavo, por serem companheiros um do outro, tornando assim possível a conclusão deste curso. À minha família, que, por vezes, apoiou-me nos prestando auxílio com os meus filhos, possibilitando-me estudar para as disciplinas do curso.

Em especial, ao meu orientador, Leonardo Murta, por confiar em meu trabalho e por sua extrema dedicação com seus orientandos. Agradeço por seus ilustres ensinamentos e conselhos, além de sempre me receber tão cordialmente em nossas reuniões.

Aos membros da banca examinadora, Troy Kohwalter, Daniel de Oliveira e Márcio Barros, por disponibilizarem seu tempo e por contribuírem na análise crítica de minha dissertação.

Ao ex-diretor da Fundação Getulio Vargas, Gerson Lachtermacher, e também ao superintendente, João Carlos da Silva Freitas, por me incentivarem a realizar este curso, liberando-me de certas atividades durante o expediente normal de trabalho.

Por fim, aos amigos que fiz na UFF, durante o curso de mestrado, com os quais tive a oportunidade de trocar diversas experiências profissionais e acadêmicas, e que, por vezes, ajudaram-me profissionalmente e em minha vida pessoal.

“Quando menos esperamos, a vida coloca diante de nós um desafio para testar nossa coragem e nossa vontade de mudança: nesse momento, não adianta fingir que ainda não estamos prontos. O desafio não espera. A vida não olha para trás. Uma semana é tempo mais que suficiente para sabermos decidir se aceitamos ou não o nosso destino.” (*Paulo Coelho*)

RESUMO

A técnica de ramos tem sido amplamente adotada em sistemas de controle de versões para permitir o desenvolvimento colaborativo de *software*. Contudo, o isolamento propiciado pelos ramos pode impor desafios no processo futuro de *merge*. Recentemente, empresas como Google, Microsoft, Facebook e Spotify, entre outras, têm desenvolvido funcionalidades diretamente no ramo principal de projetos por meio da adoção da técnica de *Feature Toggles*. Essa técnica possibilita o isolamento sem a necessidade da criação de ramos, potencialmente reduzindo os desafios de *merge*. Entretanto, faltam evidências na literatura sobre os benefícios e as limitações de *Feature Toggles* no tocante ao desenvolvimento colaborativo de *software*. Nesta dissertação, foram analisados os efeitos da aplicação de *Feature Toggles* em 949 projetos de *software* livre, sendo estes escritos em seis linguagens de programação diferentes. Primeiramente, foi identificado o momento da adoção de *Feature Toggles* em cada projeto. Em seguida, foi observado se a adoção de *Feature Toggles* implicou alterações significativas na frequência ou na complexidade de *merges* provenientes de ramos, e também no número de defeitos e no tempo médio necessário para corrigi-los. Foi observada uma redução no esforço de *merge*, mas também foi possível detectar um aumento tanto no número de defeitos quanto no tempo necessário para corrigi-los.

Palavras-chave: *Feature Toggles*, desenvolvimento na linha principal, ramos, *merges* e defeitos.

ABSTRACT

Branching has been widely adopted in version control to enable collaborative software development. However, the isolation caused by branches may impose challenges on the upcoming merging process. Recently, companies like Google, Microsoft, Facebook, and Spotify, among others, have adopted trunk-based development together with feature toggles. This strategy enables collaboration without the need of isolation through branches, potentially reducing the merging challenges. However, the literature lacks evidences about the benefits and limitations of feature toggles to the collaborative software development. In this thesis, we study the effects of applying feature toggles on 949 open-source projects written in 6 different programming languages. We first identified the moment in which each project adopted feature toggles. Then, we observed whether the adoption implied significant changes in the frequency or complexity of branch merges as well as in the number of defects and the average time to fix them. We could observe a reduction in the merge effort but also an increase in the number and the time needed to fix defects after the adoption of feature toggles.

Keywords: Feature toggles, trunk-based development, branch, merges e defects.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de implementação do carrinho de compras sem (a) e com (b) a adoção de FT 19
Figura 2 – Painel de controle do *framework Togglz* 23
Figura 3 – Painel de controle do *framework LaunchDarkly*. 24
Figura 4 – Exemplo da heurística adotada para identificar um *commit* de *merge* proveniente de ramos 35
Figura 5 – Métrica definida por PRUDÊNCIO *et al.* (2012) e implementada por MOURA e MURTA (2018) 36
Figura 6 – *Corpora* utilizado para responder a cada questão de pesquisa 37
Figura 7 – Distribuição da quantidade de *commit* por *merge* baseado no *corpus* C_{Merge} 38
Figura 8 – Distribuição de *commits* por defeito, considerando projetos do *corpus* $C_{Defeitos}$ 40
Figura 9 – Momento médio da adoção de FT por linguagem de programação 45
Figura 10 – Distribuição de *merge* por 100 *commits* antes e após a adoção de FT 47
Figura 11 – Gráfico de dispersão com o número de *merges* por 100 *commits* antes e depois da adoção de FT 48
Figura 12 – Comparação do número de *merges* por 100 *commits* por linguagem de programação 49
Figura 13 – *Boxplot* sobre $C_{QP2.2}$ – comparação entre o esforço de *merge* (em linhas) necessário para cada *merge* antes e depois da adoção de FT 50
Figura 14 – *Boxplot* sobre $C_{QP2.1\ \&\ QP2.3}$ – comparação do esforço total de *merge* necessário (em linhas) por 100 *commits* antes e depois da adoção de FT 52
Figura 15 – *Boxplot* sobre o *corpus* $C_{QP3.1\ \&\ QP3.3}$ – comparação entre a distribuição de defeitos por 100 *commits* antes e depois da adoção de FT 53
Figura 16 – Distribuição do tempo necessário (em dias) para correção de um defeito antes e depois da adoção de FT 54
Figura 17 – Distribuição de tempo (em dias) para corrigir defeitos em intervalo de 100 *commits* antes e depois da adoção de FT 56
Figura 18 – Comparativo das distribuições de *merge* por 100 *commits* por cada desenvolvedor 57

LISTA DE TABELAS

Tabela 1 – *Frameworks* que implementam FT e suas respectivas palavras-chave 32
Tabela 2 – Distribuição de projetos por linguagem de programação sobre o *corpus* C_{Limpo} 33
Tabela 3 – Distribuição de projetos do *corpus* $C_{QP2.1\ \&\ QP2.3}$ 39
Tabela 4 – Distribuição de projetos do *corpus* $C_{QP2.2}$ 39
Tabela 5 – Distribuição de projetos com suas respectivas linguagens do *corpus* $C_{QP3.1\ \&\ QP3.3}$ 41
Tabela 6 – Distribuição de projetos com suas respectivas linguagens do *corpus* $C_{QP3.2}$ 41
Tabela 7 – Comparação entre projetos que foram criados com FT e projetos que adotaram FT em algum momento de sua história 46
Tabela 8 – *Merges* por 100 *commits* usando o *corpus* $C_{QP2.1\ \&\ QP2.3}$ 48
Tabela 9 – Média do esforço de *merge* (em linhas) utilizando o *corpus* $C_{QP2.2}$ 51
Tabela 10 – Esforço total de *merge* (em linhas) em 100 *commits* 52
Tabela 11 – Defeitos por 100 *commits* antes e depois da adoção de FT 54
Tabela 12 - Média de tempo necessário (em dias) para correção de um defeito antes e depois de FT, utilizando o *corpus* $C_{QP3.2}$ 55
Tabela 13 – Estatísticas sobre o tempo necessário (em dias) para corrigir defeitos antes e depois da adoção de FT. 56

SUMÁRIO

Capítulo 1 – INTRODUÇÃO 14
1.1 MOTIVAÇÃO 14
1.2 OBJETIVOS 15
1.3 ORGANIZAÇÃO 17
Capítulo 2 – *FEATURE TOGGLES* 18
2.1 INTRODUÇÃO 18
2.2 APLICAÇÕES DE *FEATURE TOGGLES* 18
2.2.1 TIPOS DE *TOGGLES* 21
2.3 DÍVIDA TÉCNICA DE *TOGGLES* E MECANISMOS DE IMPLEMENTAÇÃO 22
2.4 TRABALHOS RELACIONADOS 24
2.4.1 MÉTODO UTILIZADO 24
2.4.2 ANÁLISE DOS ESTUDOS RELEVANTES 25
2.5 CONSIDERAÇÕES FINAIS 27
Capítulo 3 – MATERIAIS E MÉTODOS 29
3.1 INTRODUÇÃO 29
3.2 QUESTÕES DE PESQUISA 29
3.3 *FRAMEWORKS* DE *FEATURE TOGGLES* 31
3.4 METODOLOGIA DE OBTENÇÃO DO *CORPUS* 33
3.4.1 *MERGES* DE RAMOS E MEDIÇÃO DE ESFORÇO 34
3.4.2 IDENTIFICAÇÃO DE DEFEITOS 36
3.5 FILTRAGEM DO *CORPUS* 37
3.5.1 FORMAÇÃO DO *CORPUS* PARA ANÁLISE DA QP.2 ($C_{QP2.1\ \&\ QP2.3}$, $C_{QP2.2}$) 38
3.5.2 FORMAÇÃO DO *CORPUS* PARA ANÁLISE DA QP.3 ($C_{QP3.1\ \&\ QP3.3}$, $C_{QP3.2}$) 40
3.6 MÉTODO DE ANÁLISE 41
3.7 AMEAÇAS À VALIDADE 42

3.8 CONSIDERAÇÕES FINAIS 44
Capítulo 4 – RESULTADOS E DISCUSSÕES 45
4.1 INTRODUÇÃO 45
4.2 QUAL É O NÍVEL DE ADOÇÃO DE *FEATURE TOGGLES* EM PROJETOS DE *SOFTWARE* LIVRE (QP.1)? 45
4.3 QUAIS SÃO OS EFEITOS DA ADOÇÃO DE *FEATURE TOGGLES* NO PROCESSO DE *MERGE* PROVENIENTE DE RAMOS (QP.2)? 46
4.3.1 NÚMERO DE *MERGES* (QP.2.1) 47
4.3.2 COMPLEXIDADE DE *MERGES* (QP.2.2) 50
4.3.3 ESFORÇO TOTAL DE *MERGES* EM 100 *COMMITS* (QP.2.3) 51
4.4 QUAIS SÃO OS EFEITOS DA ADOÇÃO DE *FEATURE TOGGLES* NO NÚMERO DE DEFEITOS E NO TEMPO DE CORREÇÃO (QP.3)? 52
4.4.1 NÚMERO DE DEFEITOS (QP.3.1) 53
4.4.2 TEMPO NECESSÁRIO PARA CORRIGIR UM DEFEITO (QP.3.2) 54
4.5 TEMPO PARA CORRIGIR DEFEITOS EM 100 *COMMITS* (QP.3.3) 55
4.6 INFLUÊNCIA DO NÚMERO DE DESENVOLVEDORES NOS RESULTADOS 57
4.7 COMPARAÇÃO COM OS TRABALHOS RELACIONADOS 58
4.8 CONSIDERAÇÕES FINAIS 59
Capítulo 5 – CONCLUSÃO 60
5.1 CONTRIBUIÇÕES 60
5.2 TRABALHOS FUTUROS 61

CAPÍTULO 1 – INTRODUÇÃO

1.1 MOTIVAÇÃO

Em função da necessidade de respostas mais rápidas às demandas exigidas pelo mercado no tocante ao desenvolvimento de *software*, é necessário que haja linhas de desenvolvimento em paralelo. Sendo assim, as técnicas de ramificação (*branching*), suportadas por sistemas de controle de versão, têm sido amplamente adotadas para permitir o desenvolvimento colaborativo de *software*. Se, por um lado, essas técnicas permitem um isolamento temporário do código, favorecendo o desenvolvimento em paralelo, por outro, demandam um esforço adicional do desenvolvedor para realizar a integração (*merge*) do código desenvolvido em paralelo.

Dependendo de vários fatores, como o tempo do isolamento de desenvolvimento de funcionalidades, o número de desenvolvedores e a complexidade do código, a técnica de ramos pode trazer riscos para o projeto, tendo, como consequência, a realização incorreta de *merges* (SHIHAB; BIRD; ZIMMERMANN, 2012). Tais *merges* incorretos podem acarretar um risco de quebra de código ou de comportamento indesejado de funcionalidades.

Esse cenário se agrava quando são utilizados métodos modernos de desenvolvimento, baseados em iterações com ciclos curtos de entrega. Por exemplo, métodos ágeis, como Scrum e XP, quando combinados com o processo de entrega contínua (do inglês, *continuous delivery*), realizam entregas de produtos de *softwares* em ciclos de desenvolvimento mais frequentes, ou seja, permitem que funcionalidades sejam entregues de forma mais rápida aos clientes (CHEN, 2015). Em função do aumento do número de ciclos, a frequência de abertura de ramos e a necessidade de merges de código tendem a aumentar, potencializando os riscos anteriormente citados (CHEN, 2015).

Diante desse problema, grandes empresas globais, tais como Google, Microsoft, Facebook e Spotify, entre outras, têm adotado recentemente a técnica de *Feature Toggles* (denominada daqui em diante, neste estudo, como FT) como alternativa à utilização de ramos no desenvolvimento de seus produtos. Essa técnica permite um processo colaborativo de desenvolvimento mais simples, em que novas funcionalidades são implementadas diretamente na linha principal de desenvolvimento do projeto (*trunk-based development*). Como consequência natural esperada, a equipe de desenvolvedores deverá ficar menos exposta aos problemas relacionados aos *merges* provenientes de ramos. Adicionalmente, a técnica permite que sejam efetuados testes A/B em novas funcionalidades com o intuito de avaliar o respectivo

desempenho ou usabilidade antes de incorporá-las em definitivo no projeto. Por fim, de forma a minimizar a exposição a possíveis defeitos no lançamento de funcionalidades, a técnica de FT permite que sejam efetuadas liberações de funcionalidades concluídas ou parcialmente concluídas somente a certos grupos de usuários.

Embora o uso de FT tenha aumentado nas empresas, a literatura carece de evidências científicas sobre seus benefícios e limitações no desenvolvimento colaborativo de *software*. Os poucos trabalhos encontrados são baseados em estudos de caso sobre projetos individuais, estudos teóricos ou pesquisas de opinião. Por exemplo, RAHMAN *et al.* (2016) mencionam os resultados da adoção de FT no projeto do Google Chrome, que é considerado um *software* de grande porte e utilizado por milhares de usuários no mundo todo. Adicionalmente, SCHERMANN *et al.* (2016) apresentam uma visão geral de práticas de entrega contínua com o uso de FT por meio de uma pesquisa de opinião com desenvolvedores. Já NEELY e STOLT (2013) descrevem a experiência no uso de FT durante a implantação do processo de entrega contínua. Finalmente, SCHNEID (2017) e POOL (2013) discutem a utilização da técnica de FT como alternativa ao uso de ramos na prática de integração contínua (do inglês *continuous integration*) (FOWLER, MARTIN, 2006).

Nenhum dos estudos encontrados conseguiu fornecer evidências quantitativas, baseadas em um grande número de projetos, sobre os benefícios ou as limitações da utilização de FT em comparação com a prática de ramos no desenvolvimento colaborativo de *software*. Da mesma forma, também não foi possível encontrar estudos baseados em um grande número de projetos que tenham analisado ou medido os efeitos de FT com relação ao número de defeitos gerados em aplicações ou ao tempo necessário para corrigi-los.

1.2 OBJETIVOS

Diante da motivação apresentada, em linhas gerais, o objetivo principal deste estudo é compreender, por meio da análise de centenas de projetos, de que forma a adoção de FT em projetos de *software* afeta o desenvolvimento colaborativo, mais especificamente, no âmbito de *merges* de ramos e sobre os defeitos gerados. Sendo assim, o objetivo deste estudo está subdividido em três metas:

- analisar qual é a popularidade da adoção de FT em projetos de *software* livre;
- analisar o efeito da utilização de FT em projetos de *software* e compreender as mudanças na frequência do processo de *merge* provenientes de ramos e na complexidade exigida por esse processo;

- compreender se a adoção de FT afeta a qualidade de entrega de produtos, mais especificamente, em relação ao número de defeitos gerados e ao tempo necessário para sua resolução.

De forma a atingir essas metas, foi analisado um grande número de projetos reais de *software* livre, obtidos no GitHub, que implementam a técnica de FT. Com o intuito de automatizar a obtenção das informações necessárias, foram desenvolvidos *scripts* de código em Python. Todas as informações coletadas por esses *scripts* foram categorizadas e armazenadas em tabelas, em um banco de dados relacional MYSQL. Isso permitiu, por meio do desenvolvimento de consultas, a obtenção dos dados necessários, que foram utilizados para as análises deste estudo. Em função de projetos menores e outros duplicados, houve também necessidade de limpeza e tratamento das informações inicialmente coletadas. Assim, o corpus limpo para análise foi formado por 949 projetos escritos em seis linguagens de programação conhecidas.

Em um próximo passo, todos os projetos pertencentes a esse *corpus* inicial foram clonados, na sua versão mais recente, para ambiente local, o que permitiu obter informações mais precisas sobre cada projeto, entre elas, o momento exato da adoção de FT. De posse dessa informação, foi possível realizar análises comparativas sobre cada projeto referente ao impacto da introdução de FT, mais especificamente em termos de *merges* realizados. Para atingir a meta referente a defeitos, as informações obtidas até então foram complementadas com informações sobre *issues* e *pull requests* de cada projeto.

Em seguida, as metas anteriormente mencionadas foram convertidas em questões de pesquisas mais específicas, as quais são descritas no Capítulo 3. Adicionalmente, foram estabelecidos critérios e métricas de análise que permitiram responder às questões definidas. Finalmente, foi observado se a adoção de FT implicou alterações significativas na frequência ou na complexidade de *merges* provenientes de ramos, e também no número de defeitos e no tempo médio necessário para corrigi-los. Foi observada uma redução no esforço de *merge*, mas também foi possível detectar um aumento tanto no número de defeitos quanto no tempo para corrigi-los.

Com o intuito de permitir a reprodutibilidade deste estudo, estão disponíveis no GitHub[1], todos os artefatos utilizados, tais como: o *dump* do banco de dados com todos os dados referentes aos projetos, os *scripts* em Python utilizados para minerar os projetos obtidos no GitHub e os *scripts* em R para realização das análises estatísticas. Adicionalmente, foi

[1] https://github.com/gems-uff/feature-toggles

disponibilizado uma orientação mais prática de como se procederam as filtragens nos *corpora* e a aplicação de cada *script* em R.

1.3 ORGANIZAÇÃO

Este estudo está organizado em cinco capítulos. No Capítulo 2, são apresentados os conceitos da técnica de *Feature Toggles*, juntamente com os tipos de *toggles*, suas aplicações e os trabalhos relacionados a esta dissertação. No Capítulo 3, são apresentadas as questões de pesquisa sobre o estudo e sua relevância. Além disso, também são descritos os métodos de obtenção de projetos e os métodos adotados para formação de cada *corpus*, que servirão de base para realizar as análises e responder a cada uma das questões de pesquisa levantadas. No Capítulo 4 são apresentados os resultados obtidos, respondendo a cada questão de pesquisa, juntamente com as respectivas discussões. Finalmente, no Capítulo 5, são ponderadas as contribuições deste estudo e sugeridos possíveis trabalhos futuros relacionados ao assunto.

CAPÍTULO 2 – *FEATURE TOGGLES*

2.1 INTRODUÇÃO

Feature Toggles (FT), também conhecida como *Feature Flags*, *Feature Flippers* ou *Feature Switches*, é uma técnica que permite ativar ou desativar blocos de código em função da necessidade (RAHMAN *et al.*, 2016). Para tal, é utilizada uma instrução condicional (*if then else*) controlada por uma variável booleana que representa a funcionalidade em questão. Essa técnica se apresenta como uma alternativa à utilização de ramos, de tal forma que uma funcionalidade pode ser testada mesmo antes de estar completa ou pronta para ser liberada. Assim, a técnica permite, entre outras características, que o código-fonte de novas funcionalidades seja continuamente integrado diretamente na linha principal do projeto enquanto ainda está sendo desenvolvido (HEYS, 2014).

Muitas empresas têm adotado a prática de integração contínua (DUVALL; MATYAS; GLOVER, 2007) visando minimizar riscos na introdução de defeitos associados à integração de código (processo de *merge*). De uma forma geral, a integração contínua é uma prática de desenvolvimento em que os desenvolvedores envolvidos em um projeto integram seus trabalhos continuamente. Essa prática baseia-se na premissa de que processos de desenvolvimento mais curtos geram integrações de código mais simples, e, mesmo se houver quebras de código, serão mais fáceis de serem resolvidas (MEYER, 2014). Então, em função do aumento necessário de integrações que é proposta nessa metodologia, RAHMAN *et al.* (2016) sugerem a utilização de FT para facilitar esse processo.

Neste capítulo, são abordados os principais conceitos da técnica de FT no tocante ao desenvolvimento de *software*, por meio da elucidação de suas principais características e aplicações. São detalhados também os tipos de *toggles* existentes e suas respectivas características, as quais são demonstradas com um exemplo hipotético. Adicionalmente, são apresentadas algumas formas de implementação de FT que vêm sendo adotadas por grandes empresas de *software*. Finalmente, são indicados e discutidos os trabalhos relevantes de outros autores referentes à utilização dessa técnica em projetos de *software*, mais precisamente em substituição ao processo de ramificação e no que diz respeito a defeitos.

2.2 APLICAÇÕES DE *FEATURE TOGGLES*

A implementação de FT consiste em realizar alterações no código de uma aplicação, visando à segregação de funcionalidades por meio da utilização de comandos condicionais "*if then else*".

Assim, de forma a exemplificar essa definição, é apresentado, na Figura 1, um código intencionalmente simples de FT, escrito na linguagem C#, referente à implementação de um carrinho de compra para um *e-commerce*. Na ocasião, a equipe de desenvolvimento desejava construir uma nova implementação do carrinho de compras (*ShowNewShoppingCart*) e, ao mesmo tempo, continuar utilizando a implementação atual (*ShowShoppingCart*).

Na célula da esquerda, é apresentada a forma como estava implementado o código antes da adoção de FT. Já na célula da direta, é apresentada a implementação depois da introdução da técnica. Com o uso de FT, o novo carrinho de compras poderá estar habilitado somente para a equipe de desenvolvimento enquanto permanece desligado para os outros usuários.

No exemplo, é possível notar que a questão central da implementação de FT está relacionada à função descrita pelo código "isFeatureEnabled", a qual é responsável por determinar o acesso à nova implementação. Esse código condicional, denominado *toggle*, visa verificar se a funcionalidade deve ou não ser executada. Como será apresentado mais à frente, neste mesmo capítulo, existem implementações mais sofisticadas para a utilização de *toggles*.

Essa segregação por meio da utilização de *toggles* permite que desenvolvedores possam decidir quando e para quem essa funcionalidade deverá estar disponível. De uma forma geral, uma aplicação pode conter várias *toggles*, cada uma delas representando a execução de uma simples funcionalidade ou mesmo um conjunto maior de procedimentos. Assim, com a ativação ou desativação de *toggles*, é possível obter um maior controle sobre o processo de liberação de uma versão para o ambiente de produção, sem a necessidade de múltiplas implantações.

(a)

```
static void main() {
    ...
    ShowShoppingCart();
    ...
}

public bool ShowShoppingCart() {
    ...
}
```

(b)

```
static void main() {
    if (isFeatureEnabled("newShoppingCart"))
        ShowNewShoppingCart();
    else
        ShowShoppingCart();
}
public bool ShowNewShoppingCart() {
    ...
}
public bool ShowShoppingCart() {
    ...
}
```

Figura 1 – Exemplo de implementação do carrinho de compras sem (a) e com (b) a adoção de FT

Muitas são as aplicações de FT no âmbito do desenvolvimento de *software*. Além de ser uma alternativa à utilização de ramos, a técnica de FT permite a viabilização de *Dark Launching*, a utilização de *Canary Realeases* e, finalmente, a realização de testes A/B em funcionalidades. A seguir, são apresentadas cada uma delas.

O processo de *Dark Launching* consiste na liberação de funcionalidades ainda em desenvolvimento, porém desabilitadas, diretamente no ambiente de produção para um

determinado grupo de usuários (NEELY; STOLT, 2013). Dessa forma, FT provê aos desenvolvedores a capacidade de realizar alterações, de forma segura, em códigos de aplicações diretamente no ramo principal do projeto (*i.e.*, *trunk* ou *master*), sem a necessidade de criação de ramos de funcionalidades (BERCZUK; APPLETON; BROWN, 2003) para isolamento de alterações em paralelo. Como consequência disso, é esperada a redução do isolamento decorrente do uso de ramos, minimizando então a necessidade de realização de futuros *merges*.

Por exemplo, uma equipe pode criar uma nova funcionalidade diretamente na linha principal do projeto, mantendo-a desabilitada para os usuários finais com o uso de uma *toggle* enquanto a funcionalidade estiver sendo desenvolvida. Mesmo que o código esteja no ambiente de produção, ele não estará disponível para os usuários devido à *toggle*. Quando a funcionalidade estiver concluída e testada, será possível disponibilizá-la a todos os usuários por meio da ativação da respectiva *toggle* ou de sua incorporação definitiva ao código final da aplicação, via remoção da *toggle*. Vale notar que essa forma de isolamento ocorre no próprio código e, como consequência, não é total. Apesar de o código isolado pela *toggle* não ser executado em produção, ele está sendo continuamente integrado (*i.e.*, compilado e testado) com o restante do sistema.

Além das características já citadas, FT também permite a utilização de *Canary Releases*, também conhecida como *Phased Rollouts* ou *Incremental Rollouts*. Segundo SATO (2014), *Canary Releases* objetivam reduzir o risco de introdução de uma nova funcionalidade em ambiente de produção. Essa técnica permite liberar uma determinada funcionalidade para um grupo específico de usuários que podem ser selecionados por meio de critérios estabelecidos, tais como: região geográfica, perfil do usuário ou quaisquer outras características que venham a ser convenientes. Por exemplo, conforme mencionado anteriormente, para reduzir o risco de uma nova implantação, seria possível a sua disponibilização, com a ativação de uma *toggle*, somente para um determinado grupo de usuários. Caso se compreenda que a funcionalidade teve o comportamento esperado, sua disponibilidade poderia ser ampliada, de forma gradativa, para outros grupos de usuários, até que esteja completamente implantada. Se for detectado algum tipo de problema na nova funcionalidade, é possível retornar à versão anterior, bastando desativar a respectiva *toggle* por completo.

Empresas como Facebook, por exemplo, costumam utilizar uma estratégia de liberação de múltiplas *Canary Releases*, sendo que uma dessas versões é liberada somente internamente para empregados da companhia, enquanto outras são liberadas externamente para certos grupos de usuários. Esse procedimento é feito com o uso de FT, aplicando uma configuração que ativa

ou inibe determinadas *toggles* para cada uma das versões disponibilizadas (FEITELSON; FRACHTENBERG; BECK, 2013).

Ainda segundo SATO (2014), FT, além de permitir a redução do risco de novas implementações, promove também a possibilidade de execução de testes A/B. Os testes A/B visam avaliar a viabilidade de adoção de uma determinada funcionalidade. Portanto, pode-se habilitar uma determinada funcionalidade para um grupo específico de usuários e outra funcionalidade para outro. Por meio da observação de uma variável dependente, pode-se coletar os dados necessários e, em seguida, executar testes de hipóteses, identificando qual funcionalidade demonstrou melhores resultados para uma determinada tarefa.

2.2.1 TIPOS DE *TOGGLES*

De acordo com FOWLER (2010), existem dois tipos principais de *toggles*: de lançamento (do inglês, *Release Toggles*) ou de negócio (do inglês, *Business Toggles*).

As *toggles* de lançamento (ou de desenvolvimento) têm usualmente o caráter temporário e são desabilitadas por padrão durante o desenvolvimento de uma funcionalidade. Portanto, a *toggle* que implementa uma determinada funcionalidade estará habilitada somente para um grupo de desenvolvedores e permanecerá desabilitada para o restante dos usuários ou outros grupos de desenvolvedores. Uma vez que a funcionalidade esteja completamente desenvolvida e os devidos testes efetuados, a *toggle* de lançamento pode ser removida do código. Assim, de uma forma geral, as *toggles* são classificadas pelo tempo que irão fazer parte do projeto e de acordo com seu propósito.

Por exemplo, voltemos ao caso mencionado neste capítulo, em que uma equipe estaria desenvolvendo um novo carrinho de compras com características mais modernas, proporcionando uma nova experiência em compras, referente a um projeto de *e-commerce*. Dessa forma, durante o tempo de desenvolvimento, a equipe que está atuando na construção da funcionalidade estará experimentando o novo carrinho, enquanto todos os demais usuários estarão ainda visualizando o carrinho de compras antigo.

As *toggles* de negócio, por sua vez, têm um caráter permanente ou são implementadas por tempo superior ao tempo de desenvolvimento da respectiva funcionalidade. Elas têm a responsabilidade de permitir, propositalmente, que haja a existência de fluxos de funcionalidades dinâmicos. Isto é, em determinadas situações, é necessário que certas *toggles* sejam deixadas intencionalmente no código.

Ainda com base no exemplo citado anteriormente, o gestor do projeto deseja liberar o novo carrinho de compras apenas para um grupo de usuários. Logo, mesmo após a conclusão

da funcionalidade, a *toggle* ainda permanecerá no código e ativa para esse grupo de usuários, objetivando, neste caso, a redução do risco de introdução da nova funcionalidade. Em seguida, o gestor do projeto decide que é necessário realizar um teste A/B no carrinho de compras. Para isso, são implantadas métricas de coleta de informações no *e-commerce*, e a respectiva *toggle* é ativada para usuários aleatórios ou de um determinado perfil. Ao término do teste A/B, o gestor compara, por exemplo, o total de produtos comprados por usuários que foram expostos ao novo carrinho com o total de produtos comprados por aqueles que usaram o carrinho antigo. Ao final do teste A/B, já com a certeza do funcionamento do novo carrinho e com o resultado positivo da análise dos dados coletados, o gestor pode solicitar a remoção da respectiva *toggle*, incorporando o novo carrinho de forma definitiva ao código principal do projeto.

Finalmente, com a utilização do novo carrinho de compras, é permitido ao administrador do *e-commerce* selecionar as formas de pagamento disponíveis no momento, como pagamento via cartão de crédito, boleto bancário, etc. Sendo assim, esta *toggle* teria uma duração permanente no projeto, conduzindo o usuário a experimentar tipos de fluxos de compras diferentes.

Muitas empresas de *software* adicionam *toggles* em seus produtos de forma a proporcionar customizações dinâmicas, sem a necessidade de novas instalações ou implantações. Como exemplo, a Google, por meio de seu navegador Chrome, oferece uma lista extensa de *toggles* que podem ser gerenciadas pelos usuários finais. Para acessá-la, basta digitar, na url, *chrome://flags/*.

2.3 DÍVIDA TÉCNICA DE *TOGGLES* E MECANISMOS DE IMPLEMENTAÇÃO

Um dos desafios na utilização de FT é manter sob controle a dívida técnica de *toggles*. FOWLER (2010) enfatiza que é sempre importante gerenciar todas as *toggles*, em particular, aquelas do tipo lançamento, removendo-as assim que a respectiva funcionalidade esteja implantada em ambiente de produção. O objetivo é não perder o controle da responsabilidade de cada uma delas. Além disso, BIRD (2014) menciona que o mau gerenciamento das *toggles* pode tornar o código da aplicação confuso, deixá-lo mais difícil de testar e menos seguro. RAHMAN *et al.* (2016) destacam essa importância e demostram, em um estudo de caso baseado no Google Chrome, como ocorreu esse controle no *software* da Google.

As *toggles* podem ser implementadas nos projetos de diversas formas. É possível citar implementações de baixa complexidade, realizadas diretamente no código por meio de instruções “if then else”, ou implementações mais sofisticadas. As implementações variam conforme a necessidade de cada projeto, no entanto, algumas delas favorecem também a

obtenção de um melhor controle sobre as *toggles* aplicadas. Quanto mais simples e direta for a implementação das *toggles*, mais difícil será o controle delas.

Entre as implementações mais sofisticadas, é possível citar o gerenciamento com o uso de banco de dados para registro das *toggles* (HEYS, 2014) ou mesmo a utilização de *frameworks* que suportam a técnica.

Os *frameworks* de FT, em sua maioria, fornecem uma camada que auxilia a implementação das *toggles* nas aplicações. Além disso, alguns *frameworks* oferecem também um painel de controle em que é possível listar todas as *toggles* implementadas e agrupá-las conforme diversas características, tais como nível de desempenho, versão da aplicação e funcionalidades que já foram introduzidas em produção e que deverão ser removidas. Por meio do painel de controle, também é possível acompanhar quais *toggles* estão ativas ou inativas e para quais usuários estão vinculadas, colaborando para uma melhor administração do débito de *toggles*.

Como exemplo, é possível citar os painéis de controle do *framework Togglz*, de *software* livre para linguagem Java (Figura 2), e do *framework LaunchDarkly*, de código proprietário voltado para múltiplas linguagens (Figura 3).

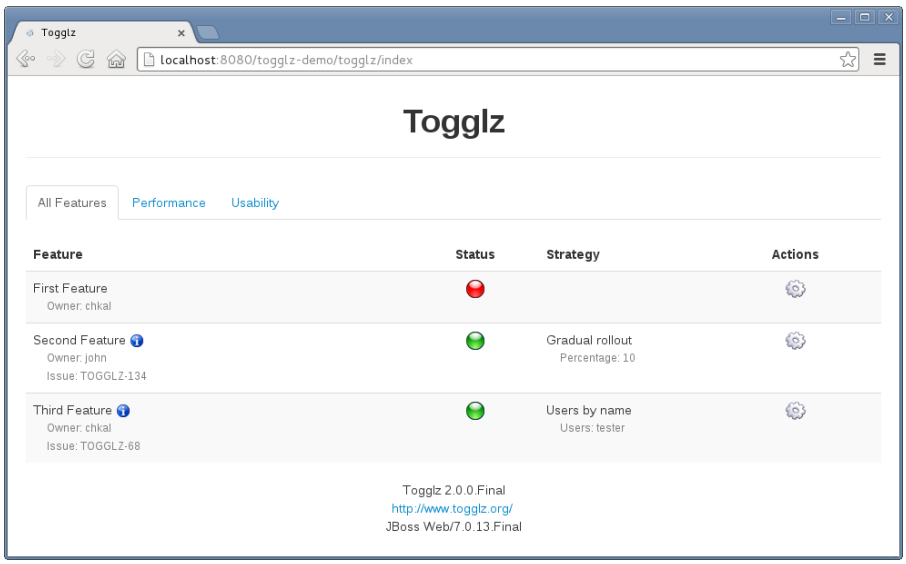


Figura 2 – Painel de controle do *framework Togglz*

Algumas empresas têm desenvolvido seu próprio *framework* baseado em FT, como é o caso do Facebook. Conhecida como *GateKeeper*, a ferramenta utilizada no Facebook permite aos desenvolvedores selecionarem o grupo de usuários, com determinado perfil, que deverá experimentar uma nova funcionalidade que foi liberada em ambiente de produção (FEITELSON; FRACHTENBERG; BECK, 2013). Finalmente, vale ressaltar que estão

disponíveis, na internet, diversos *frameworks* de FT baseados em código-aberto para muitas linguagens de programação. No Capítulo 3 será apresentado um apanhado desses *frameworks*.

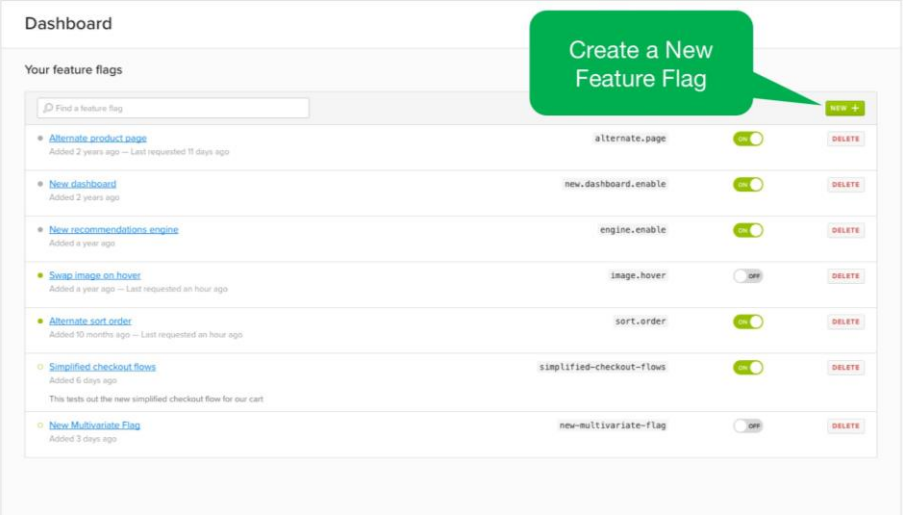


Figura 3 – Painel de controle do *framework LaunchDarkly*.

2.4 TRABALHOS RELACIONADOS

Apesar de estar sendo adotado em projetos de *software* por grandes empresas, ainda há pouca evidência na literatura sobre os benefícios ou limitações da adoção de FT. Os poucos trabalhos existentes discutem a experiência de utilização de FT por meio de análises teóricas, estudos de casos específicos ou baseados em pesquisas diretas com desenvolvedores, conforme detalhado no restante dessa seção.

2.4.1 MÉTODO UTILIZADO

De forma a buscar trabalhos relacionados com o assunto desta pesquisa, foi utilizada uma metodologia baseada em quatro etapas. A primeira etapa (E1) consiste em realizar diversas consultas sobre FT no Google Acadêmico. A segunda etapa (E2) objetiva selecionar e avaliar, entre os resultados obtidos em E1, aqueles que, de alguma forma, relacionavam-se ao assunto deste estudo. Na terceira etapa (E3), foi aplicada a técnica de *snowballing* (WOHLIN, 2014) no conjunto de trabalhos selecionados na etapa E2. O objetivo da etapa E3 foi buscar estudos relevantes citados pelo conjunto de trabalhos selecionados na etapa E2 e também localizar outros que citavam os estudos importantes da etapa E3. Por se tratar de um assunto relativamente recente na comunidade, as pesquisas realizadas até esta etapa não retornaram muitos resultados. Sendo assim, foi necessário complementar as três etapas anteriores com uma etapa adicional (E4), que consistiu em realizar buscas sobre FT em páginas e *blogs* de autores de referência nesse tema e também em *websites* sobre entrega contínua (do inglês *continuous delivery*) e/ou sobre integração contínua (do inglês *continuous integration*).

A seguir, são detalhados os conjuntos obtidos nas quatro etapas.

Na etapa E1, foram realizadas buscas no Google Acadêmico utilizando palavras-chave como "*feature toggle(s)*", "*feature flag(s)*", "*feature switch(es)*" combinando-as com as palavras "*branch*" ou "*merge*" ou "*continuous delivery"* ou "*continuous integration"*. Adicionalmente, repetiu-se a mesma consulta anterior ("*feature toggle(s)*", "*feature flag(s)*", "*feature switch(es)*") combinando-as com "*defect(s)*" ou "*bug(s)*" ou "*error(s)*". Assim, o conjunto inicial foi formado por 48 estudos entre artigos e livros.

Na etapa E2, foram selecionados artigos baseados na leitura de seus resumos (*abstract*) e de suas introduções. Para cada um dos livros selecionados, foi visitado o respectivo índice. Quando encontrada alguma referência sobre FT, o capítulo/seção foi então avaliado. Dessa forma, o conjunto E2 foi composto de 13 artigos/livros.

Na etapa E3, com a aplicação da metodologia de revisão sistemática de *snowballing*, foram encontrados mais 7 estudos relevantes, totalizando 20 artigos. No entanto, nesse processo, foi observado que muitas das referências bibliográficas também apontavam para páginas de internet e *blogs* de autores de relevância no assunto. Assim, conforme informado anteriormente, foram realizadas pesquisas diretamente na internet que contribuíram com 11 páginas/*blogs*, totalizando 31 estudos. A relação completa com todos os artigos encontrados, suas referências e também a respectiva fase em que foi selecionado, encontra-se descrito no APÊNDICE A.

2.4.2 ANÁLISE DOS ESTUDOS RELEVANTES

Parte dos 30 estudos levantados fazia referência ao mesmo assunto e os pontos mais relevantes sobre a utilização de FT estão descritos a seguir, nesta seção. A maior parte dos estudos encontrados refere-se a estudos de casos, pesquisas entre desenvolvedores e empresas, e também a estudos teóricos sobre sua aplicabilidade no processo de entrega contínua e na prática de integração contínua.

FOWLER (2010) introduz os conceitos básicos de FT e HODGSON (2016) descreve cada tipo de *toggles*, demonstrando, por meio de exemplos simples, cada uma de suas respectivas aplicabilidades. Além disso, HODGSON (2016) apresenta algumas formas de implementação da técnica por meio de um exemplo hipotético, enfatizando a possibilidade de proceder com o desenvolvimento colaborativo diretamente na linha principal do projeto.

NEELY e STOLT (2013) descrevem, em um estudo de caso, as alterações ocorridas nas diversas etapas do desenvolvimento de um projeto após a implantação do processo de entrega contínua em uma empresa de desenvolvimento de *software*. Em função dos esforços necessários para integração de código proveniente de ramos de longa duração, que ocorriam de forma

frequente, juntamente com os respectivos atrasos no desenvolvimento de funcionalidades, em função da dificuldade de integração, os integrantes da equipe de desenvolvimento passaram a desenvolver diretamente na linha principal do projeto com o auxílio de um *framework* baseado em FT. Além disso, a adoção de FT permitiu, sem que tivessem ocorrido efeitos colaterais, que fossem realizados testes de usabilidade e de desempenho em novas funcionalidades diretamente em produção, por meio da ativação da respectiva *toggle*.

Da mesma forma, REHN (2012), em um estudo teórico, discute a importância da adoção da prática de integração contínua no tocante à possibilidade de redução de problemas técnicos e na detecção, com antecedência, de defeitos de *software*. Para praticar corretamente a integração contínua, é mencionada, em seu estudo, a necessidade de automatizar rotinas de *build*, sendo apresentadas, em seguida, algumas ferramentas que dão suporte à sua implantação, tais como *Ant* e *Maven*, e também servidores específicos de *build*, como *CI Jenkins* e *CruiserControl*. Finalmente, ainda neste contexto da integração contínua, são comparadas algumas técnicas de desenvolvimento colaborativo (como utilização de ramos, FT e ramos por abstração (FOWLER, M., 2014) (HTTERMANN, 2012), sendo então sugerida a utilização de FT como forma alternativa ao uso de ramos, com o intuito de redução do esforço de *merge*.

RAHMAN *et al.* (2016) mencionam os resultados da adoção de FT no projeto do Google Chrome. Foram comparados alguns processos relativos ao desenvolvimento antes e depois da adoção de FT. No estudo, os autores relatam que anteriormente a adoção da técnica, a equipe de desenvolvimento trabalhava em ciclos de seis semanas e realizavam a elaboração de novas funcionalidades em conjunto com as alterações corretivas, diretamente em um único ramo de *release*. Além disso, este ramo era bloqueado até que todas as funcionalidades, que foram planejadas, estivessem totalmente concluídas. Este processo remetia a um esforço considerável por parte dos desenvolvedores em estabilizar o código alterado e mesclado (muitas vezes por volta de 500 correções). Adicionalmente, contribuía com a introdução de atrasos aos prazos finais de entrega de novas funcionalidades e sobre a correção de defeitos. Em função desses problemas anteriormente mencionados, os autores relatam que empresas como Google (entre outras, como Facebook), que costumam disponibilizar pequenas atualizações de seus projetos, preferem utilizar FT de forma a liberar mais rapidamente uma nova funcionalidade em produção e reduzir o esforço total de *merge*. Além disso, também foi reportado que desenvolvedores puderam realizar a correção de defeitos com menor esforço (tempo menor) devido à não necessidade de troca de ramos, evitando atualizações de código dos ramos e potenciais perdas de código nessas ações.

Adicionalmente, RAHMAN *et al.* (2016) apresentam a evolução da utilização de FT no projeto do Google Chrome, descrevendo também sua importância e o que foi necessário para controlar a dívida de *toggles*, de forma a evitar problemas de manutenção no código. Ademais, é mencionado que o projeto foi arquitetado de forma a facilitar a aplicação da técnica de FT. Na mesma direção, BIRD (2014) discute as vantagens e desvantagens da adoção da técnica, enfatizando a necessidade da existência de ramos de curta duração e também, conforme mencionado anteriormente, a manutenção do controle da dívida de *toggles* em função da possibilidade de criar comportamentos indesejados de funcionalidades, além de resultados indesejados (defeitos).

Ainda sobre o projeto do *web browser* da Google (Chrome), RAHMAN, RIGBY e SHIHAB (2018) apresentam um estudo que analisou a arquitetura do projeto, extraindo todas as FT de seu código e identificando o relacionamento de dependência entre as *toggles*. Sendo assim, os autores foram capazes de mapear todas as FT, por meio de uma representação modular, para criar uma visão arquitetônica do Google Chrome. Por meio desse estudo, os autores puderam prover, então, uma nova perspectiva para compreensão da arquitetura de um sistema, através do entendimento de suas funcionalidades e seus relacionamentos.

SCHERMANN *et al.* (2016) apresentam um estudo sobre processo de entrega contínua baseado em uma pesquisa com desenvolvedores, destacando os perfis e as principais técnicas que são adotadas por empresas nesse processo. Embora empresas como Facebook (FEITELSON; FRACHTENBERG; BECK, 2013) e Google (RAHMAN *et al.*, 2016) (RAHMAN; RIGBY; SHIHAB, 2018) tenham adotado FT em seu processo de desenvolvimento, a maioria das empresas que participaram do estudo não considerou a utilização de FT devido à complexidade extra que a técnica impõe ao código desenvolvido.

Nenhum dos trabalhos previamente apresentados fornece evidências quantitativas, baseadas em uma grande quantidade de projetos, sobre os benefícios ou as limitações da adoção de FT em substituição ao uso de ramos. Além disso, não foi possível encontrar quaisquer documentos que tenham medido o impacto da adoção de FT sobre o número de defeitos e sobre o tempo de correção dos mesmos em um grande número de projetos.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo, foi apresentada a fundamentação teórica sobre a técnica de FT, fornecendo informações para a compreensão de sua aplicabilidade em projetos de *software* e, principalmente, das análises que serão realizadas e discutidas nos próximos capítulos.

Conforme exposto, FT permite, em teoria, que ocorra desenvolvimento paralelo de funcionalidades e que este possa ser realizado diretamente na linha principal do projeto, sem a necessidade de utilização de ramos como forma de isolamento. Além disso, foram apresentadas outras características que permitem auxiliar na questão da redução de risco de liberação de uma nova funcionalidade e também no que diz respeito à realização de testes A/B nas mesmas.

Vale ressaltar, conforme dito anteriormente, que empresas têm tentado mitigar riscos de *merge* com a adoção da prática de integração contínua (DUVALL; MATYAS; GLOVER, 2007), pressupondo que integrações de código sejam feitas com maior frequência e em períodos menores de tempo. Para essa prática, RAHMAN *et al.* (2016) sugere a adoção de FT no lugar da utilização de ramos longos de desenvolvimento, de forma a facilitar o processo de integração. No entanto, a aplicação da técnica de FT vai além desse contexto de integração contínua, conforme foi discutido neste capítulo.

Apesar de sua utilização ser crescente por parte de grandes empresas, trata-se de uma técnica recente, havendo pouco conhecimento disponível sobre os efeitos de sua aplicação. Na literatura, não há uma unanimidade sobre os reais benefícios ou limitações da adoção de FT no lugar do uso de ramos. Dessa forma, a caracterização desse fenômeno se torna fundamental para nortear as equipes de desenvolvimento.

CAPÍTULO 3 – MATERIAIS E MÉTODOS

3.1 INTRODUÇÃO

Embora se tenha discutido, no Capítulo 2, em linhas gerais, os possíveis benefícios da adoção de FT em projetos de *software*, neste estudo, são investigados os possíveis benefícios da utilização de FT em projetos de *software* como alternativa ao uso de ramos. Mais precisamente, é avaliada se a adoção de FT pode refletir na redução do número de *merges* ou, ao menos, deixá-los mais simples de serem realizados. Por fim, é verificado o possível impacto sobre a adoção de FT em relação à qualidade de *software*, em especial, sobre a questão de defeitos. De forma a endereçar os principais objetivos, as análises deste estudo foram realizadas exclusivamente em um *corpus* de projetos de *software* livre e que adotaram a técnica de FT por meio da utilização de um *framework* em algum momento de sua história.

A abordagem utilizada por este estudo se baseia em identificar o marco da adoção da técnica em cada um dos projetos de *software* livre e assim, realizar análises pareadas, comparando os resultados antes e depois da adoção da técnica. Esta abordagem foi preferida, já que comparar projetos que utilizam FT desde de o início de sua história contra outros projetos que não fazem uso da técnica traria desafios na seleção adequada dos projetos que não fazem uso de FT, uma vez que existem milhares de projetos nestas condições.

Assim, neste capítulo, são apresentadas as questões de pesquisa de forma a responder aos objetivos mencionados na Seção 1.2. Além disso, são detalhados os *frameworks* de FT juntamente com as abordagens que foram utilizadas para identificar os projetos que fazem uso da técnica e como esses projetos foram filtrados para compor os *corpora* de análise. Adicionalmente, é descrito o método de análise que foi aplicado aos *corpora* de projetos, com o intuito de permitir responder às questões de pesquisas. Finalmente, são elencadas e discutidas as possíveis ameaças à validade referente aos métodos utilizados por este estudo, para obtenção dos projetos e filtragem dos *corpora* para cada de tipo análise que foi efetuada.

3.2 QUESTÕES DE PESQUISA

Foram elaboradas três questões de pesquisa que guiarão este estudo. Cada uma delas é descrita e detalhada a seguir, e, finalmente, todas são discutidas ao longo do Capítulo 4, juntamente com a apresentação de suas respostas. São elas:

QP.1) Qual é o nível de adoção de FT em projetos de *software* livre?

Nesta questão de pesquisa, é investigada a distribuição da utilização de *frameworks* de FT em projetos pela comunidade de *software* livre. Além disso, é caracterizado o *corpus* de projetos que utilizam FT, apresentando as linguagens de programação que são comumente utilizadas, o tamanho médio dos projetos, por meio da quantidade de *commits* e, finalmente, o momento exato em que esses projetos adotaram a técnica de FT.

QP.2) Quais são os efeitos da adoção de FT no processo de *merge* proveniente de ramos?

De acordo com HODGSON (2016), FT permite praticar o desenvolvimento colaborativo diretamente na linha principal (do inglês, *trunk-based development*), evitando a criação de ramos de longa duração. Sendo assim, tomando como marco o momento exato da adoção de FT, foi investigado se esse fato promoveu efeitos sobre o processo de integração de código. Para que fiquem mais precisos os resultados da pesquisa, essa questão foi dividida em três outras subquestões mais específicas (QP.2.1, QP.2.2 e QP.2.3).

Na QP.2.1, foi avaliado se houve alterações na quantidade de *merges* provenientes de ramos após a adoção de FT. Adicionalmente, foi verificado se esse comportamento também se repetiu nas diferentes linguagens de programação estudadas. Já na QP.2.2, foi verificado se houve modificações na complexidade das integrações após a adoção de FT, ou seja, se o esforço necessário para realizar *merges* foi alterado. Finalmente, na QP.2.3, foi investigado se o esforço total de *merge* provenientes de ramos, a cada em 100 *commits*, sofreu alteração após a introdução de FT. Esta última questão de pesquisa combina as duas questões anteriores, considerando tanto o número de *merges* provenientes de ramos quanto o seu respectivo esforço para um intervalo de 100 *commits*.

Mais adiante neste capítulo, serão detalhadas as abordagens utilizadas na obtenção de métricas para serem utilizadas nesta análise, tais como momento da adoção de FT e *merges* provenientes de ramos.

QP.3) Quais são os efeitos da adoção de FT sobre o número de defeitos gerados e o tempo necessário para correção desses defeitos?

Segundo FOWLER (2010), FT traz um desafio para que sejam realizados testes em projetos de *software* devido ao número de combinações diferentes que os testes podem gerar. Contudo, FOWLER (2010) menciona, nesse mesmo estudo, que não é necessário executar todas as combinações de testes, sendo suficiente executar testes sobre dois tipos de combinações: com todas as *toggles* ligadas e com a ativação apenas das *toggles* específicas que são esperadas na

próxima versão a ser liberada. O débito de *toggles* também é outro fator que pode elevar a complexidade do código, dificultando, portanto, a realização de testes consistentes (BIRD, 2014). Logo, nesta questão de pesquisa, é avaliado o efeito da adoção de FT sobre a ótica da qualidade do *software*, em termos do número de defeitos e do tempo necessário para sua correção. Foi levantada essa questão justamente em função da dificuldade apontada pelas afirmações anteriormente mencionadas.

Portanto, da mesma forma que a QP.2, a QP.3 também foi dividida em três subquestões (QP.3.1, QP.3.2 e QP.3.3).

Na QP.3.1, é verificado se a adoção de FT implica mudanças significativas na média de defeitos nos projetos. Na QP.3.2, são analisados os impactos de FT sobre o tempo médio de correção de defeitos. Finalmente, na QP.3.3, são avaliados os efeitos sobre o tempo total para correção de defeitos a cada 100 *commits*. Novamente, na QP.3.3, são analisados os resultados da QP.3.1 e QP.3.2 combinados, de forma que nesta questão é considerado tanto o número de defeitos e sua respectiva duração em um período específico de tempo (100 *commits*).

Da mesma forma que a questão anterior, ainda neste capítulo, serão detalhadas as abordagens para obtenção dos registros de defeitos em um projeto e o tempo necessário para corrigi-los.

3.3 *FRAMEWORKS* DE *FEATURE TOGGLES*

Conforme mencionado no Capítulo 2, a técnica de FT pode ser implementada no código de uma aplicação de diversas formas. Portanto, para que não houvesse equívocos na seleção de projetos para formação do *corpus* de análise, foi adotada como estratégia de busca a identificação de projetos de código-aberto que fazem uso de um *framework* que implementa FT. Dessa forma, estaria garantida a seleção de projetos, independentemente do tipo de implementação que foi adotada no projeto. Em função dessa abordagem, primeiramente, foi necessário identificar e estudar os *frameworks* existentes.

Não foi possível a identificação de nenhum estudo específico que tenha realizado esse tipo de levantamento. Sendo assim, inicialmente, a busca pelos *frameworks* foi baseada em *websites* de referência[2] e em livros sobre DevOps (HTTERMANN, 2012). Adicionalmente, foram minerados repositórios no GitHub que referenciavam, em sua descrição, a expressão "*feature toggles framework*" (ou "feature flags framework", "feature switches framework").

[2] http://enterprisedevops.org/feature-toggle-frameworks-list/
http://featureflags.io/resources/

Consequentemente, foram encontrados *frameworks* baseados em seis diferentes linguagens de programação.

Uma vez identificados os *frameworks*, o próximo passo foi reconhecer, para cada *framework*, palavras-chave ou trechos de código que remetiam à implantação do respectivo *framework*. A maioria dos trechos de códigos identificados remete à importação de classes. A Tabela 1 apresenta os *frameworks* que implementam FT que foram utilizados neste estudo, agrupados por sua respectiva linguagem de programação. Também são apresentadas, nessa mesma tabela, as palavras-chave utilizadas para indicar se um projeto instanciou o respectivo *framework*.

Tabela 1 – *Frameworks* que implementam FT e suas respectivas palavras-chave

Linguagem de programação	***Framework***	**Palavras-chave**
C# (*.cs)	Switcheroo	*IFeatureToggle*
	FeatureSwitcher	*using FeatureSwitcher*
	FeatureToggle	*using FeatureToggle;*
		using FeatureToggle.Toggles;
Java (*.java)	Togglz	*import org.togglz.core.feature*
	FF4J	*import org.ff4j.FF4j*
JavaScript (*.js)	Ericelliot/feature-toggle	*require(" feature-toggles")*
	angular-toggle-switch	*module.provider\('toggleSwitchConfig'*
	fflip	*require(. fflip.)*
	ember-feature-flags	*config.featureFlags*
PHP (*.php)	Qandidate-toggle	*Qandidate Toggle*
Python (*.py)	Gutter	*gutter.client*
	Gargoyle	*from gargoyle import gargoyle*
	django-waffle	*waffle.decorators*
		from waffle
	Flask-FeatureFlags	*from flask_featureflags*
Ruby (*.rb)	Rollout	*$rollout = Rollout.new($redis)*
	featureflags	*Featureflags.defaults*
		class Admin::FeaturesController
	feature_flipper	*FeatureFlipper.features do*

Cada um dos *frameworks* mencionados na Tabela 1, possui características diversas, entre elas, a possibilidade de controle de *toggles* por meio de um painel de controle, a seleção de perfis para liberação de *Canary Releases* e suporte a testes A/B. No APÊNDICE B, é apresentada, por meio de uma tabela, as respectivas características de cada *framework*.

3.4 METODOLOGIA DE OBTENÇÃO DO *CORPUS*

Uma vez identificados os *frameworks* de FT existentes e suas respectivas palavras-chave, com a utilização da API v3 do GitHub[3], foram realizadas buscas por projetos que possuíam alguns dos termos apresentados na Tabela 1 em seu código-fonte. As informações extraídas de cada repositório e do GitHub, conforme mencionado na Seção 1.2, foram persistidas em um banco de dados MYSQL e suportadas por um esquema apresentado por meio de um diagrama de classes no APÊNDICE C. Após a realização do processo de busca, um *corpus* inicial ($C_{Inicial}$) foi formado, composto por 1.001 projetos, compreendendo seis linguagens de programação. Logo na primeira análise dos projetos desse *corpus*, foi verificada a existência de alguns projetos que representavam instâncias dos próprios *frameworks* ou "*forks*" de projetos originais já contabilizados nesse mesmo *corpus*. Assim, após a remoção desses projetos, um *corpus* limpo (C_{Limpo}) foi formado, composto por 949 projetos, ainda assim compreendendo as mesmas seis linguagens de programação. A distribuição do *corpus* C_{Limpo} é apresentada na Tabela 2. Para cada linguagem de programação, o desvio padrão é bem superior à respectiva média, o que significa a existência de uma variação grande no tamanho dos projetos (espalhamento do número de *commits* nos projetos). É possível observar também que a maioria dos projetos no *corpus* foi escrita em JavaScript, se comparado com as outras linguagens de programação.

Tabela 2 – Distribuição de projetos por linguagem de programação sobre o *corpus* C_{Limpo}

Linguagem de programação	Total de repositórios	Média de *commits*(σ)	Mínimo de *commits*	Máximo de *commits*
C#	85	337 (1.015)	1	5.196
Java	197	791 (2.151)	1	18.035
JavaScript	373	411 (3.278)	1	44.228
PHP	14	411 (723)	1	2.071
Python	151	3.438 (10.084)	1	59.712
Ruby	129	792 (2.518)	1	19.847
Total	**949**			

[3] https://github.com/

Em seguida, todos os 949 projetos foram clonados e cada um deles teve sua história analisada para segregar os *commits* que ocorreram antes e depois da introdução do *framework* de FT. Para identificar o exato momento da introdução do *framework* em cada projeto, foram verificados cada um dos *commits* do projeto, do mais recente para o mais antigo, em busca da mesma palavra-chave indicada na Tabela 1, correspondente ao *framework* utilizado. Se o termo procurado foi encontrado, então esse *commit* foi sinalizado com FT. Assim, o momento da adoção ocorreu no *commit* mais antigo sinalizado com FT. Em seguida, foram extraídas, de cada projeto, as seguintes informações antes e depois da adoção de FT, de forma a auxiliar nas respostas às questões de pesquisa:

1) Informações coletadas para análises da QP.1 e QP.2 (antes e depois de FT)

Número de *commits*, data de cada *commit* realizado, o respectivo desenvolvedor que realizou o *commit*, o número de *merges* provenientes de ramos e o esforço necessário para se efetuar o respectivo *merge*, conforme será detalhado a seguir.

2) Informações coletadas para análise da QP.3 (antes e depois de FT)

Todas as *issues* e *pull requests* encerradas (*closed*), *commit* vinculado ao encerramento, datas de abertura e de encerramento, o título, a descrição e as *labels*.

3.4.1 *MERGES* DE RAMOS E MEDIÇÃO DE ESFORÇO

O objetivo deste estudo se concentra em analisar *commits* de *merges* provenientes especialmente de ramos, em vez de *merges* de espaço de trabalho, que ocorrem como consequência natural do processo de desenvolvimento concorrente. Os *merges* provenientes de espaço de trabalho são mais usuais, decorrentes de ramos não nomeados de curto período de tempo criados a partir da operação de clonagem. Eles integram contribuições de apenas um desenvolvedor, e esse desenvolvedor é encarregado de realizar o *merge*. Por outro lado, *merges* provenientes de ramos nomeados são usualmente mais longos e envolvem múltiplos desenvolvedores (COSTA, C. *et al.*, 2014), sendo mais difíceis de gerenciar e, consequentemente, evitados por desenvolvedores.

Neste estudo, foi adotada a heurística proposta por COSTA *et al.* (2016) para identificar se um *commit* de *merge* provém de um ramo nomeado. Essa abordagem considera que um *commit merge* provém de um ramo se mais de um desenvolvedor tenha contribuído em cada lado do *merge*, conforme Figura 4. Cada círculo exibido na Figura 4 representa um *commit* que foi realizado, juntamente com a indicação do seu respectivo autor (desenvolvedor). Logo, para

o ramo azul, há a atuação dos desenvolvedores D1 e D2, e, para o ramo amarelo, a atuação dos desenvolvedores D3 e D4. Pela heurística adotada, o *commit* de *merge* **M1** não seria considerado como *merge* de ramo, pois somente um lado desse *merge* possui mais de um desenvolvedor. Por outro lado, o *commit* de *merge* **M2** seria considerado como *merge* de ramos, já que há dois desenvolvedores distintos atuando em cada um dos ramos. No entanto, é muito comum encontrar *commit* de *merges* provenientes de ramos onde somente se tenha a atuação de um único desenvolvedor, e neste caso, a abordagem utilizada, não estaria considerando o respectivo *merge*. Então, para corrigir a limitação dessa heurística, foram analisadas também as mensagens armazenadas em cada *commit* de *merge* (*log*), nas quais buscou-se pela expressão "*merge branch*".

Nesse sentido, de forma a simplificar o texto, a partir deste ponto, toda menção a *merge* neste estudo refere-se a *merges* provenientes de ramos, não considerando *merges* provenientes de espaços de trabalho.

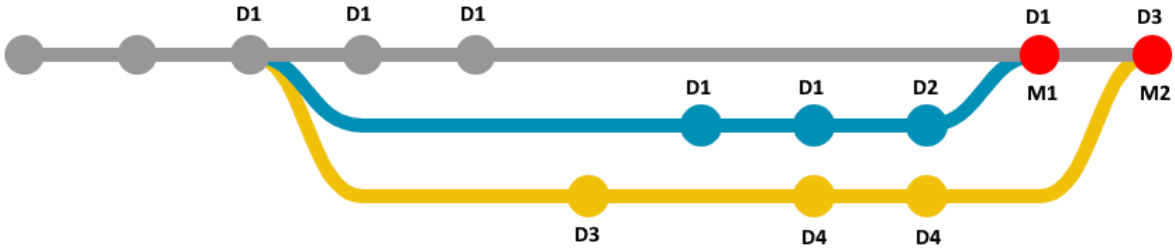


Figura 4 – Exemplo da heurística adotada para identificar um *commit* de *merge* proveniente de ramos

Para quantificar o esforço de cada *merge*, foi adotada, neste estudo, a métrica definida por PRUDÊNCIO *et al.* (2012) e implementada por MOURA e MURTA (2018). Essa métrica quantifica o número necessário de ações de adição ou remoção manuais de linhas de código na execução do *merge* (*i.e.*, *code churn*). Por exemplo, um *merge* que combina dois métodos independentes criados em diferentes arquivos provavelmente não deverá levar a nenhum (ou zero) esforço de *merge*. Por outro lado, uma integração de uma nova funcionalidade implementada em paralelo com uma ampla refatoração pode levar a um esforço significativo de *merge* para ajustar o novo código em função da refatoração.

É apresentada, na Figura 5, a métrica de esforço de *merge* definida por PRUDÊNCIO *et al.* (2012) por meio de um digrama de *Venn*. O conjunto *A* contém ações feitas em código proveniente de um ramo A. Da mesma forma, o conjunto *B* contém as ações em código proveniente de um ramo B. Finalmente o conjunto *M* contém as ações que foram efetivamente incorporadas no processo de *merge* entre os ramos *A* e *B*.

Desse modo, os conjuntos M_A, M_B e $M_{A \cap B}$ contém as ações efetuadas nos ramos que foram efetivamente incorporadas durante o *merge*, mas que não exigiram qualquer esforço por parte do desenvolvedor que realizou o *merge*. Por outro lado, $M_{esforço} = M \backslash A \cup B$ contém as ações extras, efetuadas pelo desenvolvedor que realizou o *merge* para compatibilizar o código feito nos ramos *A* e *B*. Nesta dissertação, foi quantificado o esforço de *merge* por meio da contagem dessas ações extras, introduzidas no *commit* de *merge* ($|M_{esforço}|$).

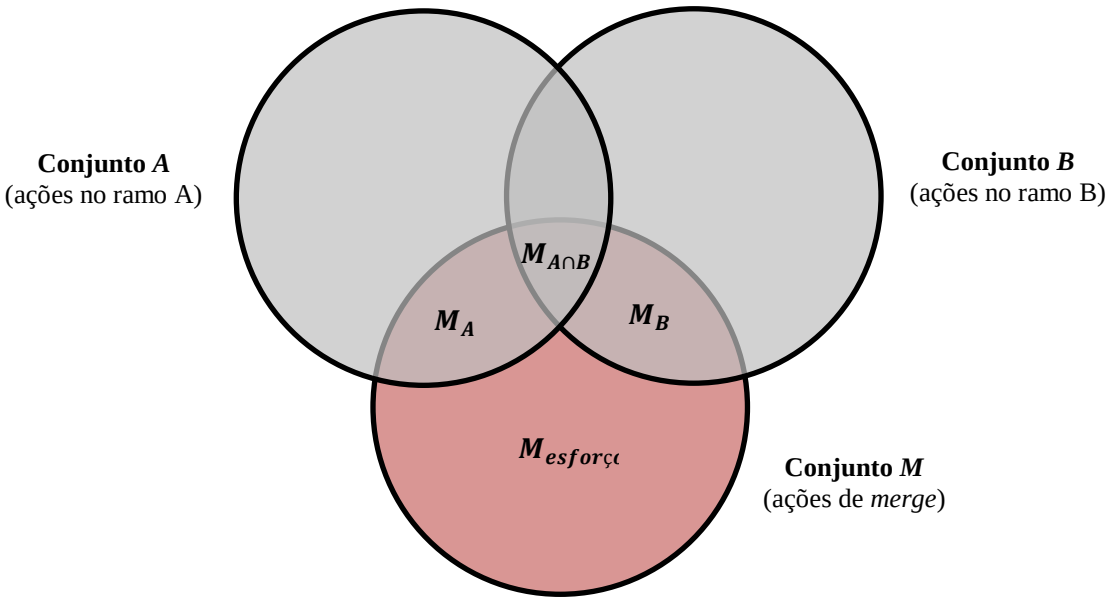


Figura 5 – Métrica definida por PRUDÊNCIO *et al.* (2012) e implementada por MOURA e MURTA (2018)

3.4.2 IDENTIFICAÇÃO DE DEFEITOS

Para responder a essa questão de pesquisa, foram coletados, no GitHub, todas as *issues* e *pull request* com situação encerrada (*closed*) de todos os projetos existentes no *corpus* C_{Limpo}. Inicialmente, era previsto identificar os defeitos nos projetos somente por meio da utilização do recurso de *labels* fornecido pelo GitHub. Esse recurso permite que os responsáveis pelo projeto possam categorizar cada *issues* ou *pull requests* como um defeito ou uma melhoria ou qualquer outra classificação. As classificações encontradas que remetiam a defeitos no projeto estão descritas no APÊNDICE D. No entanto, logo em uma primeira análise da filtragem dos repositórios, percebeu-se que havia pouquíssimos projetos que estavam utilizando o recurso "*labels*". Então, essa abordagem foi complementada com uma busca textual por palavras-chave específicas, tanto no título quanto na descrição de cada *issue* e *pull request*, que remetiam a defeitos ou problemas. Essas palavras-chaves estão descritas no APÊNDICE E.

A próxima etapa foi reconhecer se uma *issue* ou *pull request* aconteceu antes ou depois da introdução de FT. Infelizmente, foi identificado um baixo número de repositórios que

possuíam *commits* vinculados ao encerramento de *issues*. Portanto, de forma alternativa, foi necessário então comparar a data da introdução de FT de cada projeto com a data de criação e de encerramento de cada *issue* ou *pull request*. A seguir, relacionam-se os critérios adotados:

- se uma *issue* ou *pull request* foi criada e encerrada após o momento da adoção de FT, então ela foi considerada como um defeito pós-FT;
- se uma *issue* ou *pull request* foi criada e encerrada antes do momento da adoção de FT, então ela foi considerada como um defeito pré-FT;
- se uma *issue* ou *pull request* foi criada antes do momento da adoção de FT e foi encerrada após, então ela foi descartada.

3.5 FILTRAGEM DO *CORPUS*

Em uma análise inicial, foram encontrados vários repositórios com um baixo número de *commits*. Dessa forma, esses repositórios poderiam não ser adequados para a análise e para responder às questões de pesquisa QP.2 e QP.3. Portanto, surgiu a necessidade de realizar novas filtragens de repositórios para atender a cada questão de pesquisa. A Figura 6 apresenta as filtragens sobre o *corpus* inicial, assim como quantidade de projetos no *corpus* para cada questão de pesquisa. Note que foi utilizado um *corpus* diferente de projetos para a QP.2 (QP.2.1, QP.2.2 e QP.2.3) e para a QP.3 (QP.3.1, QP.3.2 e QP.3.3), sendo que ambos foram derivados do *corpus* C_{Limpo}.

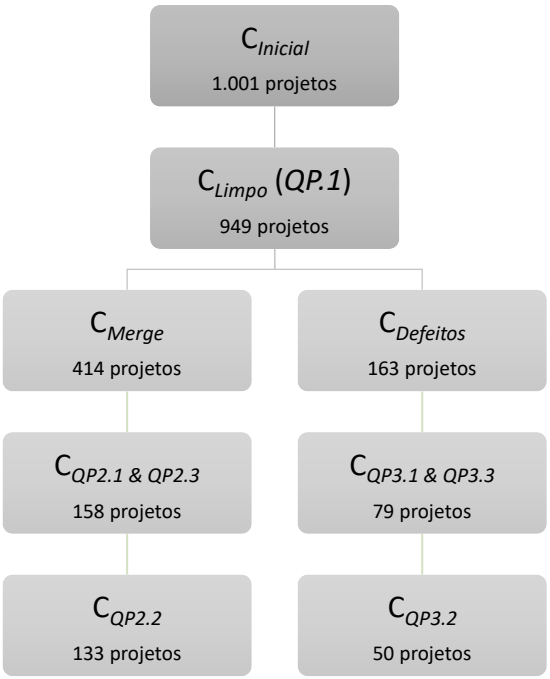


Figura 6 – *Corpora* utilizado para responder a cada questão de pesquisa

Nas seções seguintes, são apresentados os métodos de filtragem de cada *corpus* e quais foram os critérios adotados nesse procedimento.

3.5.1 FORMAÇÃO DO *CORPUS* PARA ANÁLISE DA QP.2 ($C_{QP2.1\ \&\ QP2.3}$, $C_{QP2.2}$)

Como previamente discutido, a QP.2 contrasta a densidade dos *commits* de *merges* antes e depois da introdução de um *framework* de FT. Consequentemente, o *corpus* dessa questão deve conter somente projetos que tenham quantidade suficiente de *commits* (antes e depois da adoção do *framework*), que permita, ao menos, a ocorrência de um *commit* de *merge*.

Portanto, objetivando encontrar tal quantidade de *commits*, primeiramente, foram selecionados todos os projetos que tinham ao menos um *commit* de *merge* no *corpus* C_{Limpo}. Esse subconjunto, composto de 414 projetos, foi denominado C_{Merge}. Em seguida, utilizando o subconjunto C_{Merge}, foi calculado o número médio de *commits* por *merge* em cada projeto.

Na Figura 7, é apresentada a distribuição de "*Commit* por *merge*". O limite superior do *boxplot*, que foi calculado usando o método de *Tukey* $Q3 + 1.5 \times IRQ$ (TUKEY, 1977), é de 82 *commits* por *merge*. A interpretação de tal quantidade é que, em todos os projetos que tenham, ao menos, 82 *commits*, existe a chance de ocorrer, no mínimo, um *commit* de *merge*, excetuando os projetos *outliers*.

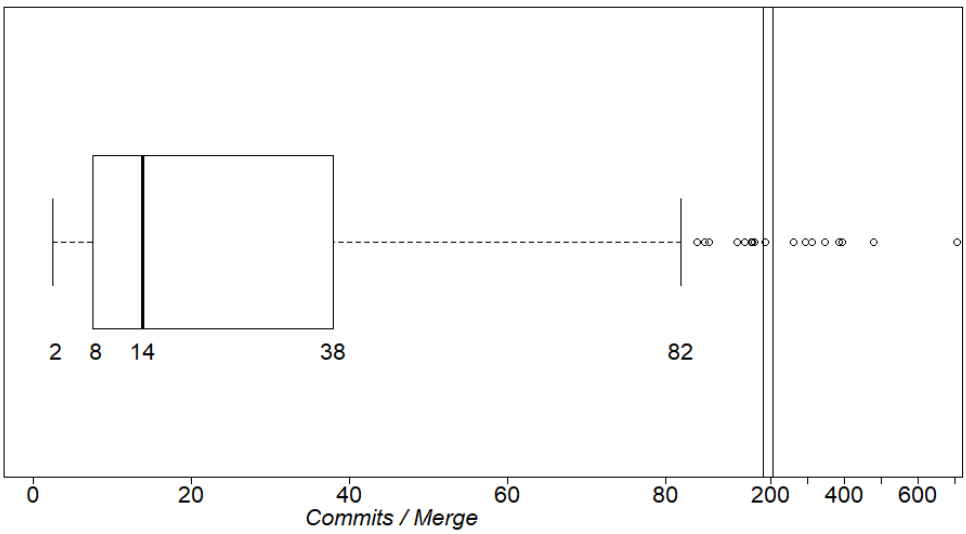


Figura 7 – Distribuição da quantidade de *commit* por *merge* baseado no *corpus* C_{Merge}

Portanto, esse critério foi utilizado para selecionar projetos do *corpus* C_{Merge}, ou seja, uma seleção de projetos que tenham simultaneamente, ao menos 82 *commits* antes e 82 *commits* depois da introdução de FT. Dessa forma, é fornecida a chance, em termos estatísticos, que qualquer projeto no *corpus* tenha quantidade suficiente de *commits* antes e depois da introdução de FT, para que ocorra ao menos um *commit* de *merge*. Adicionalmente, essa abordagem também permitiu que projetos que chegaram ao ponto de cessar o uso de ramos e merges após

a introdução de FT, também pudessem ser selecionados para análise. Assim, um novo *corpus* foi concebido para a análise da QP.2 de forma a responder, especificamente, às subquestões QP.2.1 e QP.2.3, sendo nomeado $C_{QP2.1\ \&\ QP2.3}$. Esse *corpus* contém 158 repositórios, ainda cobrindo seis linguagens de programação, e sua distribuição é apresentada na Tabela 3.

A subquestão QP.2.2 visa verificar as alterações sobre o esforço de um *commit* de *merge* antes e depois da introdução de FT, ou seja, a intenção é comparar se a complexidade de cada *merge* foi alterada. Portanto, foi necessário criar um outro *corpus* derivado de $C_{QP2.1\ \&\ QP2.3}$, que representa o conjunto de repositórios que tenham ao menos um *commit* de *merge* antes e um *commit* de *merge* depois da introdução de FT. Esse novo *corpus* criado foi nomeado $C_{QP2.2}$ e possui um total de 133 projetos. A distribuição de projetos desse *corpus* nas respectivas linguagens é apresentada na Tabela 4.

Tabela 3 – Distribuição de projetos do *corpus* $C_{QP2.1\ \&\ QP2.3}$

Linguagem de programação	**Total de repositórios**	**Média de *commits* (σ)**	**Mínimo de *commits***	**Máximo de *commits***
C#	8	3.200 (1.461)	316	5.196
Java	37	3.990 (3.503)	296	18.035
JavaScript	26	4.582 (11.326)	240	44.228
PHP	3	1.731 (559)	1.086	2.071
Python	55	9.191 (15.200)	387	59.712
Ruby	29	3.116 (4.564)	171	19.847
Total	**158**			

Tabela 4 – Distribuição de projetos do *corpus* $C_{QP2.2}$

Linguagem de programação	**Total de repositórios**	**Média de *commits* (σ)**	**Mínimo de *commits***	**Máximo de *commits***
C#	8	3.200 (1.461)	415	5.196
Java	30	4.156 (2.510)	483	7.949
JavaScript	21	5.780 (12.918)	240	44.228
PHP	3	1.731 (559)	1.086	2.071
Python	48	9.515 (15.297)	404	59.712
Ruby	23	3.670 (5.027)	296	19.847
Total	**133**			

3.5.2 FORMAÇÃO DO *CORPUS* PARA ANÁLISE DA QP.3 ($C_{QP3.1\ \&\ QP3.3}$, $C_{QP3.2}$)

Como previamente discutido, a questão QP.3 visa analisar as alterações sobre o número de defeitos gerados e o tempo necessário para corrigi-los antes e depois da adoção de FT. Portanto, de forma análoga à questão de pesquisa anterior, é preciso selecionar somente projetos que tenham *commits* suficientes (antes e depois de FT), de tal forma que permita a esses projetos terem, no mínimo, uma "*issue*" ou "*pull request*" classificada como defeito. Assim, utilizando a abordagem anteriormente mencionada, foi criado o *corpus* $C_{Defeitos}$, que compreende todos os projetos que utilizam FT e possuem, no mínimo, um defeito registrado. Esse *corpus* possui um total de 163 projetos.

Com base no *corpus* $C_{Defeitos}$, foi calculado o número médio de *commits* para cada defeito encontrado em cada projeto. Na Figura 8, é apresentada a distribuição do número de *commits* por defeito. O limite superior do *boxplot* é de 132 *commits* por defeito. A interpretação para esse limite é que todos os projetos que tenham, ao menos, 132 *commits*, excetuando-se os *outliers*, têm, estatisticamente, chance de possuir, no mínimo, um defeito.

Finalmente, esse limite de 132 *commits* foi aplicado no *corpus* $C_{Defeitos}$ para selecionar os projetos que tenham, simultaneamente, ao menos, 132 *commits* antes e 132 *commits* depois da adoção de FT. Como resultado, formou-se um novo corpus para análise da QP.3, em particular, para responder às subquestões QP.3.1 e QP.3.3. Esse corpus foi nomeado CQP3.1 & QP3.3, com um total de 79 projetos. A distribuição de projetos desse *corpus* nas respectivas linguagens é apresentada na .

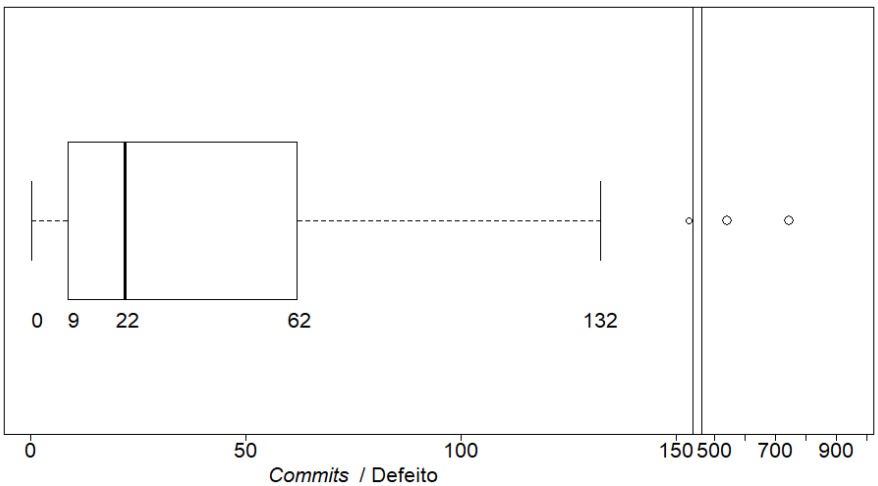


Figura 8 – Distribuição de *commits* por defeito, considerando projetos do *corpus* $C_{Defeitos}$

Tabela 5 – Distribuição de projetos com suas respectivas linguagens do *corpus* $C_{QP3.1\ \&\ QP3.3}$

Linguagem de programação	Total de repositórios	Média de *commits* (σ)	Mínimo de *commits*	Máximo de *commits*
C#	5	2.846 (1.793)	415	5.196
Java	14	5.287 (4.239)	483	18.035
JavaScript	11	1.997 (2.123)	523	7.001
PHP	2	2.054 (24)	1.086	2.071
Python	31	8.438 (14.596)	525	59.712
Ruby	16	4.084 (5.297)	507	19.847
Total	**79**			

Na subquestão QP.3.2, é contrastado o tempo médio necessário para corrigir um defeito antes e depois da introdução de FT. Portanto, objetiva-se comparar se um defeito pode ser corrigido de forma mais rápida ou não depois da adoção de FT. Para alcançar a resposta, foi necessário compor um novo *corpus* de projetos que tivesse ao menos um defeito antes e um defeito depois de FT. Esse *corpus* foi nomeado $C_{QP3.2}$ e foi derivado de $C_{QP3.1\ \&\ QP3.3}$, possuindo um total de 50 projetos. A distribuição de projetos desse *corpus* nas respectivas linguagens é apresentada na Tabela 6.

Tabela 6 – Distribuição de projetos com suas respectivas linguagens do *corpus* $C_{QP3.2}$

Linguagem de programação	Total de repositórios	Média de *commits* (σ)	Mínimo de *commits*	Máximo de *commits*
C#	2	3.553 (2.323)	1910	5.196
Java	3	7.728 (8.990)	1500	18.035
JavaScript	8	1.893 (2.146)	523	7.001
PHP	2	2.054 (24)	1.086	2.071
Python	24	6.251 (11.133)	525	50.461
Ruby	11	4.517 (6.331)	507	19.847
Total	**50**			

3.6 MÉTODO DE ANÁLISE

Com o intuito de avaliar os possíveis impactos da introdução de FT sobre os *corpora* de projetos, as análises que foram procedidas neste estudo estarão baseadas em testes de hipótese. O teste de hipótese é um procedimento estatístico que permite tomar uma decisão (aceitar ou rejeitar a hipótese nula H_0) utilizando os dados observados de um determinado experimento

(SPIEGEL, 1994). Há diversos métodos para realizar o teste de hipóteses. Os testes de hipóteses são utilizados para determinar quais resultados de um estudo científico podem levar à rejeição da hipótese nula H_0 a um nível de significância preestabelecido. Em resumo, quando são realizados testes de hipóteses, assume-se como verdadeira a hipótese nula H_0. Se existirem evidências suficientes contrárias, ela será rejeitada, e a hipótese alternativa H_1 será aceita.

De uma forma geral, nas questões de pesquisas QP.2 e QP.3, o interesse sempre é comparar duas amostras referentes aos dados dos projetos antes e depois da adoção de FT. Portanto, foram aplicados, nas amostras, testes de hipóteses paramétricos ou não paramétricos (dependendo do atendimento a pré-condições relacionadas a distribuição dos dados), que visam comparar duas amostras pareadas, avaliando a equivalência entre elas, ou seja, aceitando ou refutando a hipótese nula H_0 onde $\mu_{antes} = \mu_{depois}$.

Em cada teste de hipótese aplicado foi calculado o respectivo *p-valor* referente a normalidade da amostra e também referente ao respectivo método estatístico em que foi submetido. Dependendo do *p-valor* obtido, a hipótese nula H_0 foi então aceita ou rejeitada. Nestas análises, foi considerado um intervalo de confiança de 95%, isto é, $\alpha = 0{,}05$.

Finalmente, a aplicação dos testes de hipóteses para cada análise foi complementada com a medição de tamanho do efeito (SULLIVAN; FEINN, 2012), por meio da aplicação do teste de Cohen (para distribuições normais) ou de *Cliff's Delta* (para distribuições não normais) (MACBETH; RAZUMIEJCZYK; LEDESMA, 2011).

3.7 AMEAÇAS À VALIDADE

Apesar do cuidado com os dados, ainda há potenciais ameaças à validade dos resultados desse estudo. Nesta seção, essas ameaças são discutidas.

A primeira ameaça refere-se à abordagem usada para a formação de cada *corpus* de projetos que serviu de base para as análises das questões de pesquisa. Essa abordagem consistiu em realizar buscas por projetos de código-aberto no GitHub (utilizando a API v3) e que fazem uso de um *framework* de FT. Durante o processo de busca de projetos, foi notado que outros projetos faziam uso da técnica de FT, porém não foram considerados para análise, pois fugiam à abordagem adotada por não utilizarem um *framework* de FT.

Além disso, foram determinadas, neste estudo, algumas palavras-chave específicas para identificar os projetos e também detectar os *commits* que foram realizados com FT. É possível que haja projetos que introduziram o *framework*, porém sua utilização não se deu em todo o projeto, o que, de certa forma, pode afetar, em parte, o resultado obtido. Mais ainda, como algumas palavras-chave adotadas na busca são ligeiramente genéricas e remetem à FT, é

possível que, para a formação do *corpus*, tenham sido considerados projetos que adotaram FT, porém não empregaram diretamente um *framework*, o que não invalida o estudo.

Ainda com relação à formação dos *corpora* de análise (tanto para análise de *merges* quanto para a análise de defeitos), foi aplicada, em cada amostra, a técnica de IQR $(Q3 - Q1) \times 1{,}5$ e utilizada como critério limite para a seleção de projetos. No entanto, com a adoção desse critério existe a possibilidade de se ter incluído nesses *corpora*, projetos irrelevantes ou mesmo terem sido excluídos outros projetos de relevância. A adoção desse valor foi escolhida para permitir que, em termos estatísticos, que um *merge* ou um defeito tenha uma grande chance de ocorrer.

Para realização das análises, foi considerado que, uma vez que um projeto tenha introduzido FT, essa decisão não foi alterada, sendo então adotada a técnica até seu *commit* mais recente. Isso significa que foi considerado o marco da adoção de FT em cada um dos projetos como um momento único. De modo geral, isso pode ser considerado uma ameaça, já que um projeto poderia parar de usar FT em algum momento posterior à adoção. Contudo, de forma a detalhar um pouco mais o processo de obtenção do marco de adoção de cada projeto, foram listados todos os *commits* de cada projeto, iniciando pelo *commit* mais recente chegando ao mais antigo. O *commit* mais antigo foi então identificado como o marco da adoção. Durante esse intervalo, foi observada a utilização de FT durante todos esses *commits* em cada um dos projetos, sem que haja *commits* sem o uso da técnica. Sendo assim, essa ameaça não afetou o *corpus* adotado neste estudo.

Para a formação do *corpus* para análise de defeitos ($C_{Defeitos}$), conforme mencionado neste capítulo, foram encontradas poucas evidências de *issues* ou *pull requests* que estivessem vinculadas a *labels*, que remetiam a defeitos ou problemas. Portanto, foi adotada, adicionalmente, outra abordagem, realizando buscas diretamente no título e na descrição de cada *issue* ou *pull request*. Dessa forma, é possível que tenha faltado alguma palavra-chave ou ainda que se tenha localizado alguns poucos registros que não remeteriam de fato a um defeito na aplicação. Contudo, foi considerada a mesma probabilidade dessa possível ameaça, tanto para antes quanto para depois da adoção de FT.

De acordo com a abordagem, anteriormente mencionada, sobre classificação de cada *issue* no *corpus* $C_{Defeitos}$, é possível que uma *issue* tenha sido classificado como “pós-FT”, porém seja referente a um defeito introduzido anteriormente à adoção da técnica (“pré-FT”). No entanto, é possível entender que essa possibilidade de classificação equivocada, ocorreria somente logo após a fase de adoção da técnica em cada um dos projetos, ou seja, por um período curto de tempo. Apesar deste critério ter sido adotado para classificação de um defeito “pré-

FT" e "pós-FT", é possível que, com um aumento de funcionalidades em cada projeto e com sua maior utilização, tenhamos um aumento natural do número de defeitos (LEHMAN *et al.*, 1997). Como muitos projetos adotaram a técnica posteriormente, esse fator pode influenciar no número de defeitos "pós-FT".

Finalmente, ainda sobre o *corpus* $C_{Defeitos}$, foram considerados somente projetos que possuíam algum defeito registrado. Apesar dessa seleção, como não existe a obrigatoriedade da utilização de recurso de *issues* no GitHub como fonte de registros de defeitos encontrados, não é possível saber se um determinado projeto adotou esse recurso desde o início da sua história.

3.8 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentadas as questões de pesquisa para este estudo com suas respectivas justificativas. Também foram apresentados os *frameworks* de FT, juntamente com suas respectivas palavras-chave, que serviram de base para formação do *corpus* principal de projetos ($C_{Inicial}$, total de 1.001 projetos). Adicionalmente, foi demonstrada a necessidade de um trabalho de limpeza dos dados desse *corpus*, gerando um novo *corpus* limpo de projetos (C_{Limpo}, total de 949 projetos). Em seguida, também foram descritas as abordagens adotadas para selecionar os projetos para formação de cada *corpus* que será empregado no Capítulo 4 para responder a cada uma das questões de pesquisa levantadas.

Finalmente, foram evidenciadas e discutidas algumas ameaças à validade dos resultados deste estudo, em especial, sobre as abordagens adotadas para composição de cada *corpus* de análise.

CAPÍTULO 4 – RESULTADOS E DISCUSSÕES

4.1 INTRODUÇÃO

Neste capítulo, são apresentadas as análises que foram feitas sobre os *corpora* $C_{QP2.1\ \&\ QP2.3}$, $C_{QP2.2}$, $C_{QP3.1\ \&\ QP3.3}$ e $C_{QP3.2}$ para responder a cada questão de pesquisa. Conforme mencionado anteriormente, para compreender se houve alterações significativas após a introdução de FT nos projetos analisados, tanto com relação a *merges* quanto sobre defeitos, as análises deste estudo foram baseadas em testes de hipóteses, sendo em seguida, aceita ou rejeitada em função do *p-valor* obtido.

Para cada uma das questões de pesquisa analisadas, os resultados obtidos são apresentados, discutidos e interpretados a seguir.

4.2 QUAL É O NÍVEL DE ADOÇÃO DE *FEATURE TOGGLES* EM PROJETOS DE *SOFTWARE* LIVRE (QP.1)?

Como mencionado no Capítulo 3, foi identificado, para cada projeto, o exato momento da adoção de FT. Assim, foram observados 667 projetos (70% do *corpus* C_{Limpo}) criados sem a utilização de FT e em que, em algum momento, foi incorporado um *framework* referente à técnica. O marco médio de adoção de um *framework* de FT foi em torno de 44% de suas histórias, em termos de *commits*, com a indicação de 6% de desvio padrão.

Quando são analisados os 667 projetos agrupados por linguagem de programação, é possível observar que projetos escritos em linguagem JavaScript apresentaram o momento mais antecipado referente à adoção, em contraste com os projetos escritos em PHP, que apresentaram o momento mais tardio. De forma a elucidar melhor o resultado obtido, é apresentado, na Figura 9, o momento médio da adoção de FT em cada linguagem de programação.

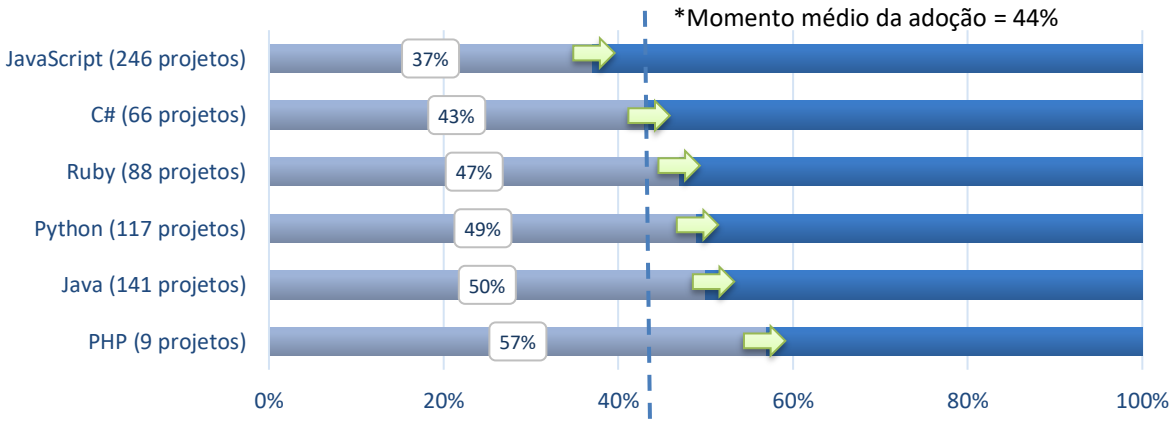


Figura 9 – Momento médio da adoção de FT por linguagem de programação

Quando foram contrastados os projetos que adotaram FT desde sua criação (30% do *corpus* C_{Limpo}) contra os projetos que adotaram a técnica posteriormente ao seu início (70% do corpus C_{Limpo}), pôde-se observar que a média de desenvolvedores e a média de *commits* diferem muito, conforme apresentado na Tabela 7. Nota-se claramente os quão maiores e mais maduros são os projetos que adotam FT após sua criação. Isso pode ser entendido de forma natural, já que se trata de uma tecnologia recente.

Finalmente, fechando essa questão de pesquisa, foi observado que 15% dos projetos que adotaram FT posteriormente ao seu início, cessaram a utilização de *merges* após a introdução da técnica no respectivo projeto.

Tabela 7 – Comparação entre projetos que foram criados com FT e projetos que adotaram FT em algum momento de sua história

Momento da adoção	Total de projetos	Média de desenvolve-dores	Média de *commits*	*Commits* por defeito*	*Merge* por 100 *commits**	Mediana LOC
Desde o início	282	2	32	30,35	1,24	897
Posteriormente	667	22	1.438	10,90	5,25	3545

***** Considerando somente projetos que tiveram defeitos registrados (**desde o início**: 18 e **posteriormente**: 145) e *merges* (**desde o início**: 43 e **posteriormente**: 371).

Resposta (QP.1): Foram identificados 949 projetos no GitHub que utilizam um *framework* de FT. Esse número é pequeno considerando o total de projetos hospedados no GitHub. A maioria dos projetos (70%) foi criada sem um *framework* de FT, mas adotou a técnica em meados de sua história (44% dos *commits*). Os projetos que utilizam um *framework* de FT desde sua criação (30%) são pequenos em termos de número de desenvolvedores e de *commits* e apresentam mais defeitos e poucos *merges*. Se comparados os projetos, JavaScript é a linguagem mais popular (26%) entre os projetos que adotaram um *framework* de FT, seguido por Java (15%) e Python (12%). Finalmente, foi observado que 15% dos projetos que adotaram a técnica posteriormente ao seu início, deixaram de realizar *merges*, praticando somente o desenvolvimento em sua linha principal.

4.3 QUAIS SÃO OS EFEITOS DA ADOÇÃO DE *FEATURE TOGGLES* NO PROCESSO DE *MERGE* PROVENIENTE DE RAMOS (QP.2)?

Nesta questão de pesquisa, o objetivo é verificar se o número e/ou a complexidade dos *commits* de *merge* indicam alguma alteração significativa nos projetos depois da adoção de FT.

4.3.1 NÚMERO DE *MERGES* (QP.2.1)

Como previamente mencionado, a seguinte análise foi baseada no *corpus* $C_{QP2.1\ \&\ QP2.3}$, que representa o total de projetos que têm *commits* suficientes para que ocorra, em termos estatísticos, ao menos um *merge*. Na Figura 10, é apresentado o *boxplot* sobre o número de *merges* por 100 *commits* realizados antes e depois da adoção de FT.

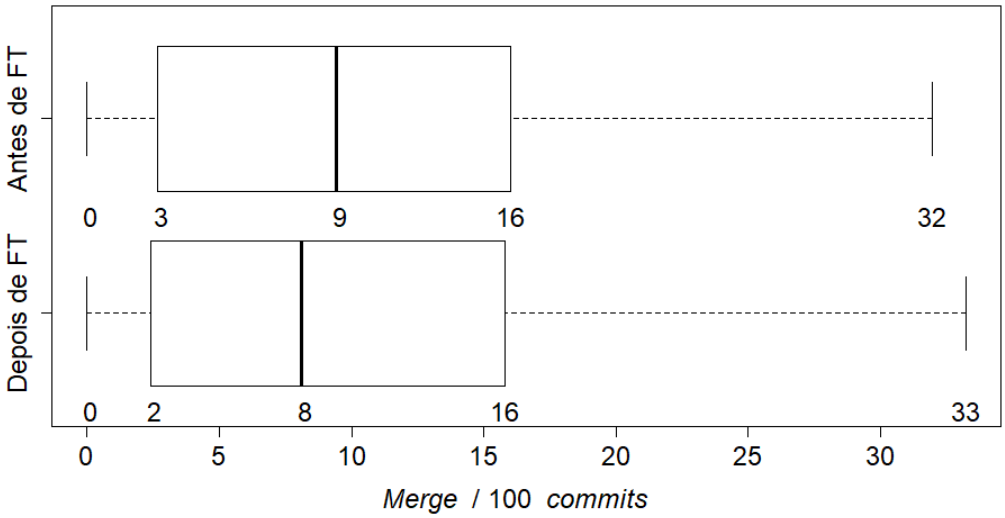


Figura 10 – Distribuição de *merge* por 100 *commits* antes e após a adoção de FT

Apesar de uma inspeção visual demonstrar distribuições similares, foram executados testes de hipóteses de forma a verificar os resultados. Em primeiro lugar, foi executado o teste de *Shapiro-Wilk* para verificar se os dados seguiam uma distribuição normal. Ambas as amostras, antes e depois de FT, não seguiam uma distribuição normal, apresentando um *p-valor* < 0,001. Considerando então a não normalidade das amostras, o teste não paramétrico de *Wilcoxon* (WILCOXON, 1992), de forma pareada, foi aplicado. Foi observado um *p-valor* = 3.292×10^{-1}, indicando que não houve diferença significativa entre as amostras.

A Figura 11 apresenta um gráfico de dispersão dos projetos baseado no número de *merges* por 100 *commits*, antes e depois da introdução de FT. É possível observar que a maioria dos projetos está concentrada no quadrante esquerdo inferior, o que demonstra que esses projetos possuem uma quantidade baixa de *merges* a cada 100 *commits*. Além disso, também é possível notar que, em geral, boa parte dos projetos se concentra próxima da linha diagonal do gráfico, indicando que não existem diferenças significativas no número de *merges* antes e depois da adoção da técnica de FT.

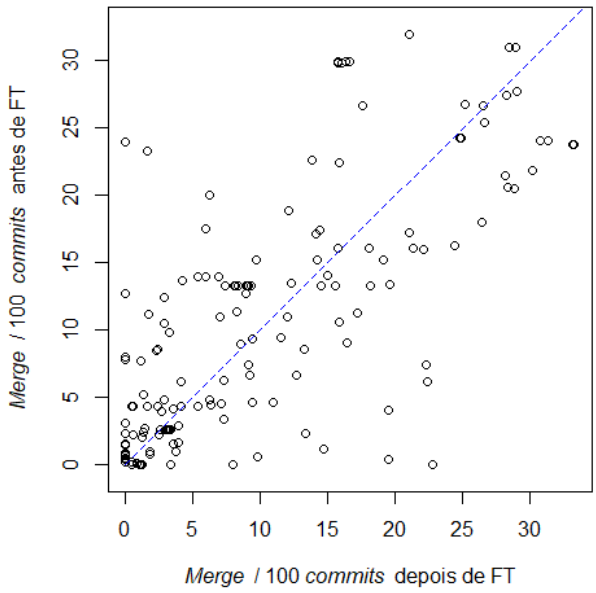


Figura 11 – Gráfico de dispersão com o número de *merges* por 100 *commits* antes e depois da adoção de FT

Portanto, apesar da possibilidade de isolamento no desenvolvimento de funcionalidades sem a necessidade da criação de ramos mais longos, não foi possível observar uma redução significativa no número de merges. Esse resultado é, de certa forma, inesperado, já que uma das premissas da adoção de FT é que há uma redução na necessidade de merges. É apresentado, na **Tabela 8**, o total de merges por 100 *commits*, com as respectivas média e mediana.

Tabela 8 – *Merges* por 100 *commits* usando o *corpus* $C_{QP2.1\ \&\ QP2.3}$

Amostra	*Merges* por 100 *commits*	
	Média	**Mediana**
Antes de FT	10,72	9,40
Depois de FT	10,16	8,10

Conforme mencionado na introdução deste capítulo, em complementação aos testes de hipóteses, será calculado o tamanho do efeito da adoção de FT. Como as amostras se apresentaram como distribuições não normais, foi aplicado o método não paramétrico de tamanho de efeito *Cliff's Delta* nessas distribuições. Adicionalmente, será considerado o critério de ROMANO *et al.* (2006) para interpretar o tamanho do efeito. Supondo d como o tamanho do efeito, então $|d| < 0{,}147$ indica efeito "negligenciável", $0{,}147 \leq |d| < 0{,}330$ indica um efeito "pequeno", $0{,}330 \leq |d| < 0{,}474$ indica efeito "médio" e $|d| \geq 0{,}474$ indica um efeito "grande". Portanto, para esta análise, foi observado um tamanho do efeito "negligenciável" para as amostras ($|d| = 4{,}4383 \times 10^{-2}$).

Finalizando a análise sobre número de *merges*, foi verificado se os resultados obtidos até então também seriam os mesmos se analisados por linguagem de programação. Na Figura 12, é apresentada a distribuição de *merges* por 100 *commits*, com seus respectivos *p-valor*, tamanho do efeito e quantidade de projetos, por cada linguagem de programação que compõe o *corpus* $C_{QP2.1\ \&\ QP2.3.}$ Nenhuma das distribuições analisadas apresentou uma distribuição normal. Sendo assim, o teste não paramétrico de *Wilcoxon*, de forma pareada, foi aplicado.

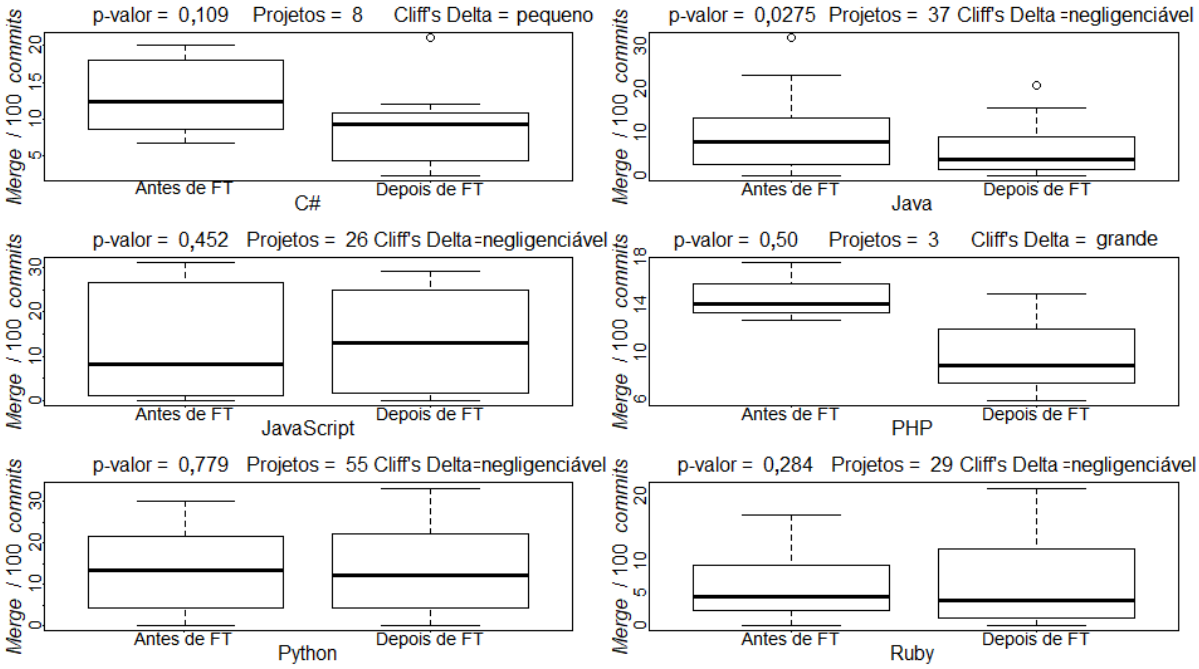


Figura 12 – Comparação do número de *merges* por 100 *commits* por linguagem de programação

Primeiramente, foi possível observar que não existe uma tendência geral entre as linguagens estudadas. Para as linguagens C#, Java e PHP, a mediana com relação ao número de *merges* reduziu depois da adoção de FT. Já para projetos em JavaScript, foi verificado o oposto. Para projetos em Python e Ruby, a mediana dessas distribuições apresentou uma pequena variação. Finalmente, somente para projetos em Java, o *p-valor* apresentou um valor inferior a 0,05, rejeitando assim a hipótese H_0, mas com o tamanho do efeito "negligenciável". Múltiplas comparações podem aumentar os erros Tipo I. De forma a evitá-los, foi aplicada a correção de *Bonferroni* (SHAFFER, 1995), tornando assim o $\alpha = 0{,}008\ (0{,}05\ \div 6)$ e fazendo com que a hipótese H_0 para projetos em Java não seja mais rejeitada.

Como o conjunto de projetos em PHP possuía somente três amostras e seu efeito foi "grande", é possível que, com mais amostras nesse conjunto, ocorra uma diferença significativa.

Resposta (QP.2.1): Embora a técnica de FT, em teoria, permita a prática de desenvolvimento colaborativo diretamente na linha principal de projetos (*trunk-based development*), não foram observadas alterações significativas no número de *merges*. O mesmo resultado foi observado quando a análise foi realizada nas diferentes linguagens de programação.

4.3.2 COMPLEXIDADE DE *MERGES* (QP.2.2)

Nesta questão de pesquisa, é estudada a complexidade dos *commits* de *merges* por meio da análise do esforço necessário para a realização de cada *merge* antes e depois da adoção de FT. Sendo assim, utilizando o *corpus* $C_{QP2.2}$, foi calculado, para cada projeto, o esforço necessário de realização de cada *merge* antes e depois da introdução de FT. O esforço de *merge* foi medido por meio da abordagem apresentada no Capítulo 3, em termos de linhas adicionadas e removidas do código. É apresentado, na Figura 13, o *boxplot* referente à distribuição do esforço médio de *merge* em cada projeto antes e depois da introdução de FT. Vale ressaltar que essa métrica foi utilizada com o intuito de minimizar a dominância de projetos com muitos *merges* no resultado obtido.

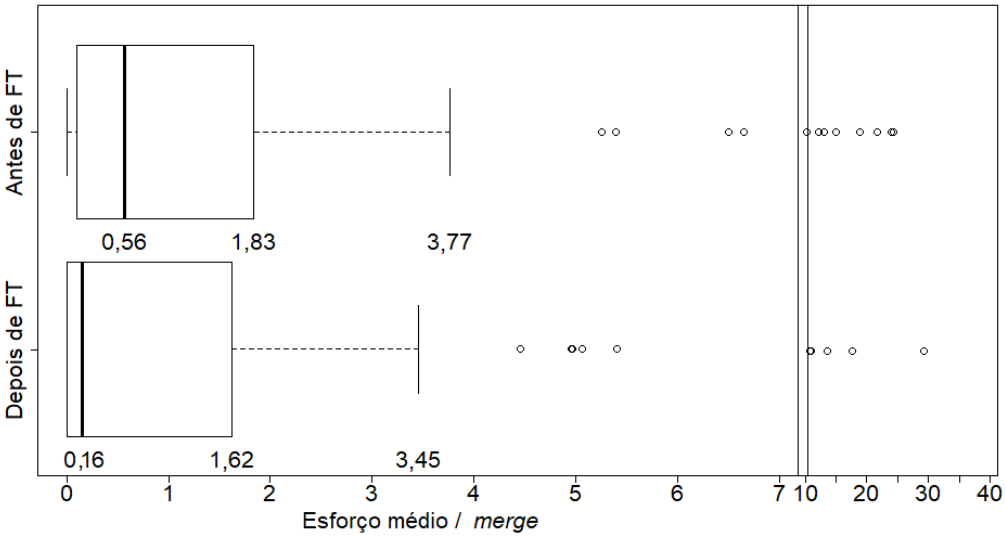


Figura 13 – *Boxplot* sobre $C_{QP2.2}$ – comparação entre o esforço de *merge* (em linhas) necessário para cada *merge* antes e depois da adoção de FT

Procedendo de maneira similar às análises anteriores, primeiramente, foi observada a não normalidade nas distribuições de esforços de *merge* antes e depois de FT, com a aplicação do teste de *Shapiro-Wilk* (*p-valor* < 0.001). Então, novamente, foi utilizado o teste não paramétrico de *Wilcoxon*, de forma pareada, sendo observado um *p-valor* = 9×10^{-2}, não indicando diferença significativa entre as duas amostras.

São apresentadas, na Tabela 9, a média e a mediana do esforço médio de *merge* para cada caso. Enquanto, antes da introdução de FT, cada *merge* demandava por volta de 15,5 linhas de ações extras, esse número caiu para cerca de 1,9 linha com a adoção da técnica nos projetos. As reduções na média e na mediana apresentaram valores em torno de 88% e 71%, respectivamente. Adicionalmente, foi verificado o tamanho do efeito por meio da aplicação do método de *Cliff's Delta*, sendo observado um efeito "pequeno" ($\mathbf{1,8893 \times 10^{-1}}$).

Tabela 9 – Média do esforço de *merge* (em linhas) utilizando o *corpus* $C_{QP2.2}$

Merge	**Média**	**Mediana**
Antes de FT	15,52	0,56
Depois de FT	1,91	0,16

Resposta (QP.2.2): Embora não tenham sido observadas alterações significativas no número de *merges* e também sobre o esforço necessário para cada *merge*, tanto a média quanto a mediana referentes à distribuição do esforço demandado por cada *merge* tiveram uma redução considerável após a adoção de FT, com um tamanho do efeito "pequeno".

4.3.3 ESFORÇO TOTAL DE *MERGES* EM 100 *COMMITS* (QP.2.3)

Até o momento, foram realizadas análises isoladas sobre o processo de *merge*, baseadas em duas variáveis: número e complexidade de cada *merge*. Se, por um lado, o número de *merges* não apresentou alteração significativa, por outro, houve uma queda frente às ações extras para se efetuar cada *merge*. Complementando as análises realizadas até então, foi verificado, em seguida, o efeito da adoção de FT em cima dessas duas variáveis combinadas.

Sendo assim, utilizando o *corpus* $C_{QP2.1\ \&\ QP2.3}$, foi calculado, para cada projeto, o esforço total necessário para realização de todos os *merges*, antes e depois da adoção de FT, normalizando o resultado por 100 *commits*. Isto é, foi calculado o esforço total necessário para se realizar todos os *merges* em um intervalo de 100 *commits* antes e depois da técnica, combinando o número de *merges* realizados e seu respectivo esforço. Na Figura 14, é apresentado um *boxplot* referente à distribuição do esforço total em um intervalo de 100 *commits*. Adicionalmente, é possível observar por meio da Tabela 10, que, enquanto, em média, esse esforço de *merge* necessário era por volta de 87,6 linhas antes da adoção de FT, depois desse marco, o número foi reduzido para 17,97 linhas.

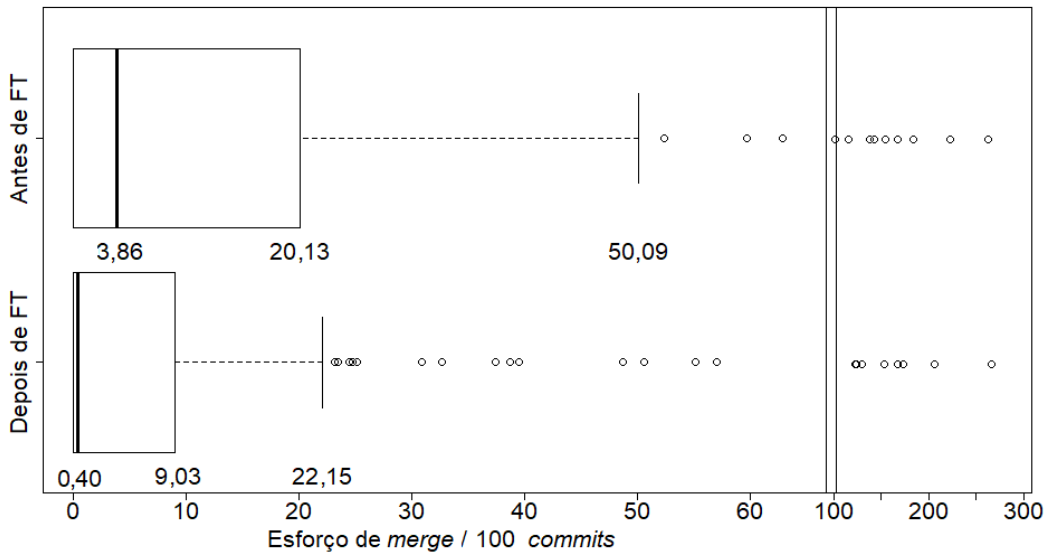


Figura 14 – *Boxplot* sobre $C_{QP2.1\ \&\ QP2.3}$ – comparação do esforço total de *merge* necessário (em linhas) por 100 *commits* antes e depois da adoção de FT

Procedendo de forma similar referente às análises anteriores, primeiramente, foi observada a não normalidade das amostras por meio da aplicação do teste de *Shapiro-Wilk* (*p-valor* < 0.001). Em seguida, o teste não paramétrico de *Wilcoxon*, de forma pareada, foi utilizado para comparar as amostras, sendo então observado *p-valor* = $2{,}2 \times 10^{-3}$, o que indica uma diferença significativa entre elas. São apresentadas, na Tabela 10, a média e a mediana do esforço total de *merge* para cada caso. As reduções na média e na mediana são da ordem de 80% e 90%, respectivamente. Para complementar a análise, aplicando o método de *Cliff's Delta*, foi obtido como resultado um efeito "pequeno" ($2{,}0537 \times 10^{-1}$).

Tabela 10 – Esforço total de *merge* (em linhas) em 100 *commits*

Merge	**Esforço de *merge* em 100 *commits***	
	Média	**Mediana**
Antes de FT	87,58	3,86
Depois de FT	17,97	0,40

Resposta (QP.2.3): Embora não tenham sido observadas alterações significativas no número de *merges*, o esforço total de *merge* demandado em 100 *commits* apresentou uma queda significativa após a adoção de FT, com um tamanho do efeito "pequeno".

4.4 QUAIS SÃO OS EFEITOS DA ADOÇÃO DE *FEATURE TOGGLES* NO NÚMERO DE DEFEITOS E NO TEMPO DE CORREÇÃO (QP.3)?

O objetivo desta questão de pesquisa é avaliar se a adoção de FT nos projetos implicou alterações significativas no número de defeitos gerados e também no tempo necessário para

correção desses defeitos. Os procedimentos executados nesta análise seguem, de forma similar, os mesmos já adotados na questão de pesquisa QP.2.

4.4.1 NÚMERO DE DEFEITOS (QP.3.1)

Esta análise foi conduzida sobre o *corpus* $C_{QP3.1\ \&\ QP3.3}$, que representa os projetos com *commits* suficientes para que exista, ao menos, um defeito antes e depois da adoção de FT. É apresentado, na Figura 15, o *boxplot* referente às amostras com a distribuição do número de defeitos por 100 *commits* antes e depois da introdução de FT.

De forma a identificar se as amostras implicam diferenças significativas após a introdução da técnica, primeiramente foi executado o teste de *Shapiro-Wilk* para verificação da normalidade de cada amostra. Foram observados um *p-valor* < 0,001 tanto para a distribuição com número de defeitos antes da adoção FT quanto para depois, indicando assim, a não normalidade de ambas as distribuições. Consequentemente, foi aplicado o teste pareado de *Wilcoxon* e observado um *p-valor* = $2{,}586 \times 10^{-11}$, indicando uma diferença significativa. Para complementar esta análise, foi calculado o tamanho do efeito utilizando o método de *Cliff's Delta*, e o resultado obtido foi um efeito "médio" para as amostras ($4{,}4720 \times 10^{-1}$).

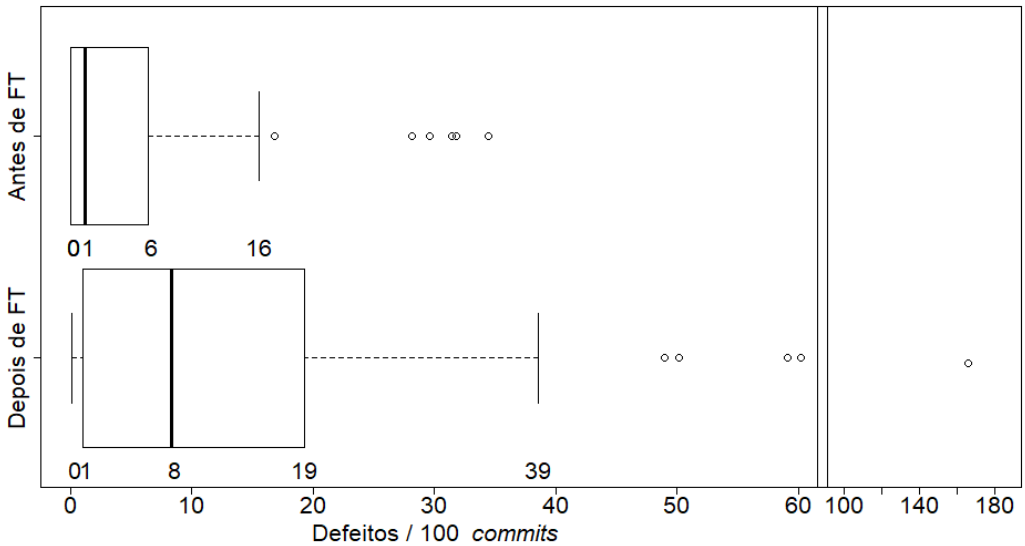


Figura 15 – *Boxplot* sobre o *corpus* $C_{QP3.1\ \&\ QP3.3}$ – comparação entre a distribuição de defeitos por 100 *commits* antes e depois da adoção de FT

É apresentado, na Tabela 11, o total de defeitos de todos os projetos do *corpus* $C_{QP3.1\ \&\ QP3.3}$ em 100 *commits* e as respectivas média e mediana para cada situação. É possível observar que todas as três métricas tiveram um aumento substancial após a introdução de FT, em torno de 197%, 198% e 609% respetivamente.

Tabela 11 – Defeitos por 100 *commits* antes e depois da adoção de FT

Amostra	Defeitos por 100 *commits*		
	Soma dos defeitos	**Média**	**Mediana**
Antes de FT	397	5,02	1,18
Depois de FT	1182	14,96	8,37

De acordo com a motivação descrita na QP.3, apresentada no Capítulo 3, a técnica de FT traz um esforço extra para realizar os testes necessários nas aplicações, em função do número de possíveis combinações de *toggles* que as aplicações podem apresentar. De fato, essa razão pode ajudar a explicar o aumento de defeitos que os projetos apresentaram após a introdução de FT.

Resposta (QP.3.1): Foi observado um aumento estatisticamente significativo no número de defeitos após a adoção de FT nos projetos. Em termos numéricos, foi observado um aumento na mediana de defeitos por 100 *commits* na ordem de 609%.

4.4.2 TEMPO NECESSÁRIO PARA CORRIGIR UM DEFEITO (QP.3.2)

Na questão QP.3.2, é conduzida uma análise sobre quais seriam os efeitos de FT sobre o tempo necessário (medido em dias) para se corrigir um defeito. Para determinar essa possível alteração, foram procedidas análises sobre o *corpus* $C_{QP3.2}$, que representa o total de projetos que apresentaram defeitos antes e depois da adoção de FT. É apresentado, na Figura 16, um *boxplot* comparando as distribuições do tempo médio necessário (em dias) para se corrigir um defeito antes e depois da introdução de FT.

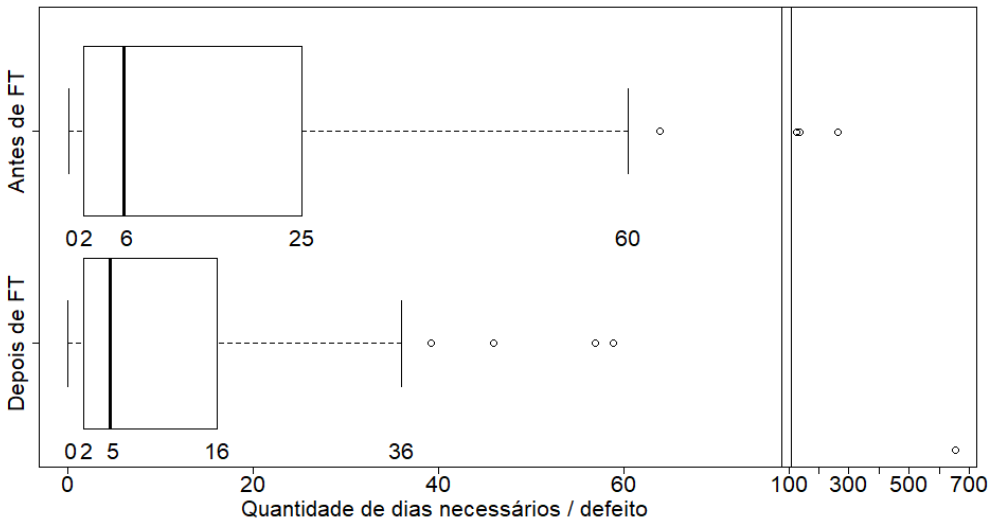


Figura 16 – Distribuição do tempo necessário (em dias) para correção de um defeito antes e depois da adoção de FT

Novamente, foi observada a não normalidade das distribuições de cada amostra com a aplicação do teste de *Shapiro-Wilk*, obtendo-se um *p-valor* < 0,001. Logo, aplicando o teste não paramétrico de *Wilcoxon*, de forma pareada, foi observado um *p-valor* = $1{,}346 \times 10^{-1}$, indicando não haver diferença significativa entre as amostras referentes ao tempo de correção de cada defeito. Por fim, aplicando o *Cliff's Delta*, foi observado o tamanho do efeito como "negligenciável" ($6{,}24 \times 10^{-2}$).

Detalhando essa questão, é apresentado, na Tabela 12, o tempo médio necessário para se corrigir um defeito em cada uma das amostras. Podemos observar um leve aumento na média (em torno de 10%), porém uma queda na respectiva mediana (em torno de 24%), ou seja, alguns projetos possuíam certos defeitos que exigiram um esforço maior, elevando assim a média de tempo.

Tabela 12 - Média de tempo necessário (em dias) para correção de um defeito antes e depois de FT, utilizando o *corpus* $C_{QP3.2}$

Defeitos	Média	Mediana
Antes de FT	23,15	6,08
Depois de FT	25,57	4,61

Resposta (QP.3.2): Não foram observadas alterações significativas no tempo necessário para corrigir um defeito após a adoção de FT. A média de tempo para corrigir um defeito teve um leve aumento em contraste com a respectiva mediana, que apresentou uma pequena redução.

4.5 TEMPO PARA CORRIGIR DEFEITOS EM 100 *COMMITS* (QP.3.3)

Complementando as análises anteriores, foi apurado se o tempo total (medido em dias) necessário para corrigir todos defeitos gerados, em um intervalo específico de 100 *commits*, sofreu alteração significativa após a introdução da técnica de FT nos projetos. De forma similar às análises apresentadas até então, foram avaliados em conjunto o número de defeitos e o tempo necessário para corrigi-los, normalizados por 100 *commits*, antes e depois da introdução de FT nos projetos. A Figura 17 apresenta o *boxplot* referente à distribuição do tempo (em dias) necessário para corrigir defeitos, normalizados por 100 *commits*, antes e depois da introdução de FT nos projetos. Em outras palavras, esta análise compara quantos dias foram necessários para corrigir todos os defeitos ocorridos em um período de 100 *commits*, sem e com a adoção da técnica de FT.

Novamente, foi observada a não normalidade de cada distribuição utilizando o teste de *Shapiro-Wilk*, obtendo um *p-valor* < 0,001. Em seguida, foi aplicado o teste não paramétrico de *Wilcoxon*, de forma pareada, sendo observado um *p-valor* = $3{,}637 \times 10^{-6}$, indicando diferença estatisticamente significativa no tempo total necessário para corrigir defeitos no intervalo mencionado. Adicionalmente, calculou-se o tamanho do efeito aplicando o *Cliff's Delta*, obtendo assim um efeito médio ($3{,}5186 \times 10^{-1}$).

Por meio da Tabela 13, é apresenta a média e mediana referentes ao tempo necessário para corrigir defeitos em um intervalo de 100 *commits*. Após a adoção de FT, é observado que a média de tempo em 100 *commits* teve um aumento de 123% e a mediana apresentou um aumento expressivo, da ordem de 1.108%.

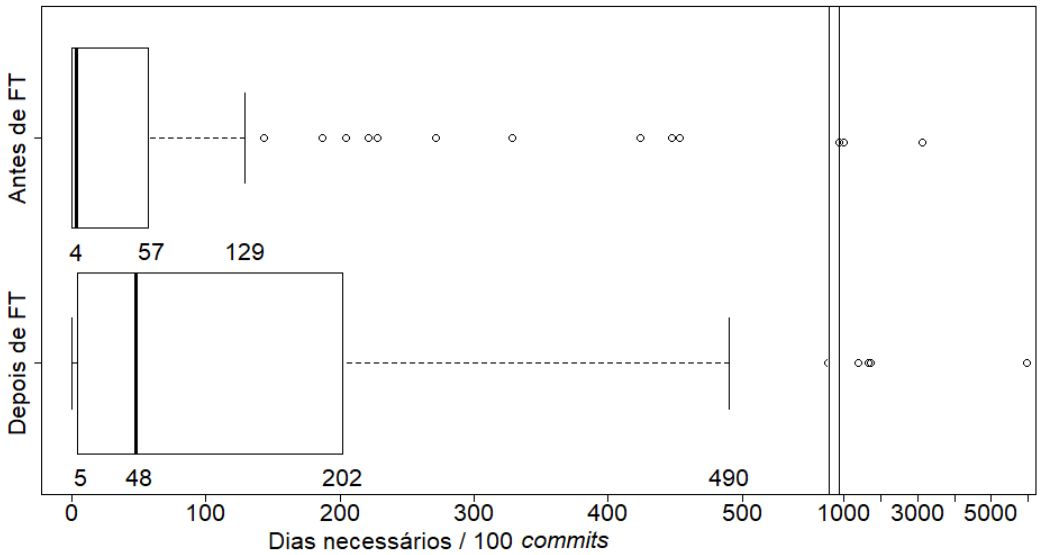


Figura 17 – Distribuição de tempo (em dias) para corrigir defeitos em intervalo de 100 *commits* antes e depois da adoção de FT

Tabela 13 – Estatísticas sobre o tempo necessário (em dias) para corrigir defeitos antes e depois da adoção de FT.

Defeitos	**Tempo necessário para corrigir defeitos em 100 *commits***	
	Média	**Mediana**
Antes de FT	113,57	3,94
Depois de FT	252,82	47,60

Resposta (QP.3.3): Foram observadas diferenças estatisticamente significativas, indicando um aumento no tempo necessário para corrigir defeitos em um intervalo de 100 *commits*, com o tamanho do efeito "médio", depois da adoção de FT nos projetos. Essa alteração representou um aumento em torno de 1.108% na mediana referente ao tempo necessário para corrigir defeitos em 100 *commits*.

4.6 INFLUÊNCIA DO NÚMERO DE DESENVOLVEDORES NOS RESULTADOS

Para finalizar este capítulo, foram realizadas análises sobre o número de desenvolvedores antes e depois da introdução de FT. Apesar de não ser o foco deste estudo, essa variável poderia influenciar o resultado obtido, principalmente no tocante à frequência de *merges*, devido à possibilidade de uma maior concorrência no desenvolvimento colaborativo dos projetos.

Foram calculados, para cada projeto, a quantidade de *merges*, normalizados em um intervalo de 100 *commits*, ponderada pelo número de desenvolvedores atuantes antes e depois da adoção de FT. Em outras palavras, quantos *merges* por desenvolvedor ocorreram em um intervalo de 100 *commits* antes e depois da introdução de FT.

É apresentado, na Figura 18, por meio de um *boxplot*, o comparativo entre as distribuições de número de *merges* em cada 100 *commits* por desenvolvedor antes e depois do uso de FT. Analisando o *boxplot*, percebe-se a similaridade entre as distribuições. Contudo, de forma a evidenciar essa similaridade, foram feitas as devidas análises estatísticas.

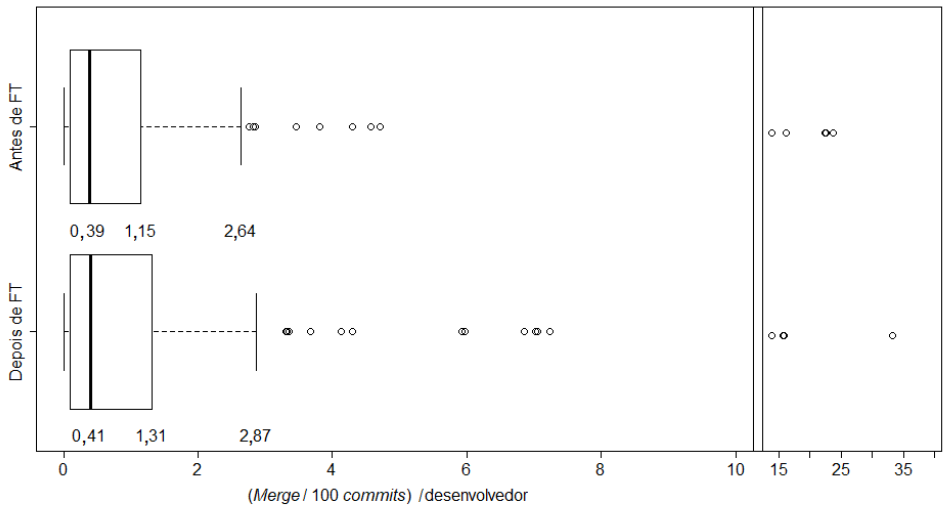


Figura 18 – Comparativo das distribuições de *merge* por 100 *commits* por cada desenvolvedor

Prosseguindo da mesma maneira com relação aos procedimentos adotados nas análises anteriores, foi então avaliada a normalidade de cada distribuição, utilizando o teste de *Shapiro-*

Wilk, obtendo um *p-valor* < 0,001, que indica não normalidade das amostras. Em seguida, foi aplicado o teste não paramétrico de *Wilcoxon*, de forma pareada, sendo observado um *p-valor* = $8{,}208 \times 10^{-1}$, indicando a não existência de diferenças estatisticamente significativas. Adicionalmente, calculou-se o tamanho do efeito aplicando-se o *Cliff's Delta*, obtendo-se um efeito "negligenciável" ($5{,}84842 \times 10^{-3}$).

Portanto, não foi observada a interferência da variável "número de desenvolvedores" nos resultados obtidos nas questões de pesquisa referentes ao *merge* (QP.2.1, QP.2.2 e QP.2.3).

4.7 COMPARAÇÃO COM OS TRABALHOS RELACIONADOS

No Capítulo 2, foram levantados os estudos relevantes feitos por outros autores referentes aos benefícios, às aplicações e às limitações de FT em projetos de *software*. Nesta seção, são comparados e analisados os resultados obtidos neste estudo com aqueles obtidos por outros autores, referenciando, em cada análise, a resposta à respectiva questão de pesquisa.

NEELY e STOLT (2013) apresentam como resultado uma redução no número de *merges* no projeto de *software* em um estudo de caso. Esse resultado foi possível com o auxílio de um *framework* de FT, que permitiu à equipe de desenvolvimento atuar diretamente na linha principal do projeto. Além disso, neste mesmo estudo, é relatado que a adoção de FT permitiu, sem ocorrência de efeitos colaterais, que novas funcionalidades fossem testadas diretamente em produção, por meio da ativação da respectiva *toggle*, reduzindo a possibilidade de defeitos. Da mesma forma, REHN (2012), em um contexto de integração contínua, sugere a adoção de FT frente à utilização de ramos de forma a reduzir o esforço de *merge*. Adicionalmente, FT é sugerida como técnica principal de desenvolvimento de *software*, visando à redução de problemas técnicos e à detecção, com antecedência, de defeitos de *software*.

Os resultados obtidos neste estudo contrastam, de certa forma, com as sugestões apresentadas por esses autores, já que não foram observadas reduções significativas na frequência de *merge* proveniente de ramos depois da adoção de FT (**Resposta QP.2.1**). Se, por um lado, com relação à frequência de *merge*, não foi observado o mesmo resultado, por outro, foi percebida uma redução no esforço total de *merge* (**Resposta QP.2.3**). Além disso, embora os autores destaquem a importância do uso de FT na prática de integração contínua, para se detectar previamente defeitos, e na visão mais ampla do processo de entrega contínua, foi observado, neste estudo, um aumento significativo com relação ao número de defeitos depois da adoção de FT (**Resposta QP.3.1**).

RAHMAN *et al.* (2016) mencionam os resultados da adoção de FT no projeto do Google Chrome. Os autores relataram que, após a introdução de FT no projeto, foi possível reduzir o

esforço total de *merge*, tornando a integração mais previsível. Além disso, também foi reportado que desenvolvedores puderam realizar a correção de defeitos com menor esforço (tempo menor) devido à não necessidade de troca de ramos, evitando perdas de códigos não atribuídos ao projeto. Embora RAHMAN *et al.* (2016) tenham baseado sua análise em somente um projeto (de grande porte e com muitos desenvolvedores), foram observados, neste estudo, resultados similares em um grupo maior de projetos, com um decréscimo significativo sobre o esforço total de *merge* (**Resposta QP.2.3**).

Ainda alinhado com os resultados obtidos no estudo de RAHMAN *et al.* (2016), foi observado um decréscimo no tempo mediano necessário para correção de defeitos (**Resposta QP.3.2**). Todavia, houve um aumento no número de defeitos (**Resposta QP.3.1**) e também foi possível observar um aumento significativo no tempo total necessário para corrigi-los (**Resposta QP.3.3**).

Adicionalmente, BIRD (2014) e RAHMAN *et al.* (2016) destacam a importância de se controlar a dívida de *toggles* de forma a evitar problemas de manutenção no código e inibir comportamentos indesejados (defeitos). Neste estudo, foi possível observar um aumento significativo no número de defeitos (**Resposta QP.3.1)**, o que, dentre outros fatores, pode ser consequência de uma inapropriada gestão da dívida de *toggles*.

4.8 CONSIDERAÇÕES FINAIS

Neste capítulo, foram discutidos os resultados obtidos e respondidas as questões de pesquisa apresentadas no Capítulo 3. Alguns resultados apresentados, de certa forma, foram surpreendentes, como foi o caso da não alteração da frequência de *merges* antes e depois da introdução de FT. Contudo, outros resultados corroboraram afirmações de outros autores, como é o caso da redução no esforço de *merge* e do aumento da complexidade de se lidar com os defeitos gerados.

CAPÍTULO 5 – CONCLUSÃO

5.1 CONTRIBUIÇÕES

Neste estudo, foram apresentados os conceitos da técnica de *Feature Toggles* e suas diversas aplicações no desenvolvimento e na implantação de projetos de *software*, tais como a sua utilização como alternativa ao uso de ramos de desenvolvimento e a possibilidade de lançamento de *Canary Releases* ou testes A/B em funcionalidades.

Embora a técnica permita essas aplicações, este estudo se concentrou especialmente em compreender suas implicações no processo de desenvolvimento na linha principal e na qualidade do produto de *software*. Sendo assim, foi possível obter evidências, por meio da análise de um grande número de projetos reais extraídos do GitHub, sobre os efeitos da adoção da técnica na frequência de *merges* e em sua complexidade. Além disso, foi possível avaliar o impacto na frequência de defeitos gerados e no tempo necessário para resolução desses defeitos.

Foi observada a aplicação de FT, por meio da utilização de *frameworks*, em uma amostra de 949 projetos baseados em seis linguagens de programação amplamente conhecidas. Ademais, 70% dos projetos levantados não se iniciaram utilizando a técnica, mas, em algum momento, passaram a utilizá-la. Em média, essa adoção ocorreu ao se atingir 44% da sua história em termos de *commits* efetuados.

Apesar de a técnica de FT permitir um isolamento de funcionalidades por meio do código-fonte do *software*, sua adoção não acarretou alterações significativas sobre o número de *merges* provenientes de ramos nas amostras estudadas. Este estudo detectou que 15% dos projetos que, antes da adoção de FT realizavam *merge*, passaram a não fazer uso de ramos de desenvolvimento após a introdução da técnica, passando a praticar somente o desenvolvimento diretamente em sua linha principal (*trunk-based development*). Outros projetos, por sua vez, continuaram a fazer uso de ramos.

Se, por um lado, o número de *merges* não apresentou alterações significativas, por outro, o esforço necessário de cada *merge* apresentou uma grande redução em sua mediana (71%). O mesmo ocorreu com o esforço total em um intervalo de 100 *commits* (90%).

Sendo assim, este estudo contribuiu para fornecer evidências de que a técnica de FT pode ser considerada uma alternativa real à utilização de ramos no desenvolvimento colaborativo de *software*, permitindo um processo de *merge* mais simples, contribuindo assim para uma redução no risco de quebra de funcionalidades ou de comportamentos indesejados provenientes do processo de *merge*. Adicionalmente, o número de desenvolvedores que

atuaram antes e depois da adoção de FT não mostrou ter influência no resultado obtido referente a *merges* com base nas amostras analisadas.

Finalmente, em contraste com o menor esforço de *merge*, foi observado, nas amostras levantadas, um aumento significativo no número de defeitos (609%) e no tempo total necessário para corrigi-los (1.108%), em um intervalo de 100 *commits*. Contudo, o tempo médio para corrigir cada defeito não apresentou alteração significativa.

Portanto, para projetos que adotarem a técnica de FT em algum momento, este estudo contribuiu evidenciando a viabilidade da técnica de FT em substituição à utilização de ramos e reforçando a necessidade de se ter um investimento em testes no projeto de forma a evitar o aumento no número de defeitos gerados. Adicionalmente, faz-se necessário também um investimento em manutenção de forma a controlar a dívida de *toggles*, evitando assim comportamentos indesejados.

5.2 TRABALHOS FUTUROS

Como dito anteriormente, este estudo se limitou a compreender quantitativamente quais são os efeitos da utilização da técnica de FT no processo de *merges* provenientes de ramos e seus impactos em defeitos. Então, de forma a aprimorar e ampliar o estudo realizado, são sugeridos alguns trabalhos futuros.

Futuramente, com uma maior adoção de FT em projetos de *softwares* livres, é sugerido executar novamente as análises realizadas neste estudo frente ao número de *merges* e defeitos. O objetivo é verificar se os resultados obtidos neste estudo continuarão consistentes ao longo do tempo.

Diante do resultado obtido referente a não observância de alterações significativas sobre a frequência de *merges*, é proposto realizar um estudo qualitativo, por meio de análises de código e entrevistas a desenvolvedores de projetos após a introdução de FT. Através desta análise será possível verificar o nível exato de adoção da técnica em cada um projetos, isto é, será possível obter informações sobre qual é o percentual de utilização de FT, em termos de arquivos ou módulos, e entender a real razão para a continuidade do uso de ramos mesmo após a adoção de FT. Adicionalmente, de forma a permitir uma melhor avaliação, os projetos poderiam ser analisados e classificados de acordo com o seu domínio, levando em conta, a natureza de cada projeto.

Com o intuito de tentar mitigar o número de defeitos em projetos que adotaram FT, é recomendado estudar de que forma são procedidos os diversos tipos de testes em grandes

projetos, garantindo, diante das inúmeras possibilidades combinatórias, que os testes estão compatíveis com a qualidade esperada pelos gestores da aplicação.

Ainda no contexto de investir em melhorias nos testes, é proposto desenvolver uma ferramenta que permita a realização de testes automatizados em aplicações que adotam FT. A ferramenta faria uso de um algoritmo que detectaria a dependência entre *toggles*, reduzindo consideravelmente o número de combinações entre elas. Dessa forma, esta ferramenta auxiliaria os desenvolvedores na elaboração de planos de testes em que seriam estabelecidas as combinações de *toggles* (ligadas e desligadas) esperadas para cada possível cenário. Com a aplicação dessa ferramenta, deseja-se automatizar as combinações de testes necessários, melhorando, dessa forma, a qualidade do produto entregue.

Em relação à capacidade de FT em executar testes A/B, é proposto o desenvolvimento de uma ferramenta que auxilie na coleta de métricas relacionadas à usabilidade de uma determinada funcionalidade implementada em FT, tais como desempenho, conformidade, operacionalidade, eficiência e etc. As métricas definidas poderiam ser inseridas nas funcionalidades por meio de programação orientada a aspectos ou introspecção e os dados coletados seriam armazenados em banco de dados para uma análise posterior. Os dados coletados poderiam auxiliar o gestor do projeto a compreender a implementação da nova funcionalidade em termos de desempenho ou usabilidade.

Outro campo de pesquisa importante é estudar e propor uma metodologia de desenvolvimento que garanta uma boa administração da dívida de *toggles*. É fundamental gerenciar de forma eficaz as *toggles* que foram implementadas, sejam elas do tipo lançamento ou negócio, com o propósito de reduzir as dificuldades referentes à manutenção de *toggles* no código desenvolvido.

Uma questão importante que carece de evidências científicas é compreender como ocorrem os impactos na persistência de informações em banco de dados de aplicações (que fazem uso desse recurso) quando há o lançamento de uma nova FT. Isto é, entender como grandes empresas procedem alterações em banco de dados (relacionais ou não relacionais) durante o desenvolvimento de uma nova FT, visto que as alterações são colocadas diretamente em produção.

Visando reproduzir o estudo em termos do número de defeitos e do tempo necessário para corrigi-los, é proposto verificar se as aplicações que já iniciaram sua história com a técnica de FT apresentam os mesmos resultados. É possível que aplicações que foram projetadas para a utilização de FT, seja com a utilização de um *framework* ou não, apresentem um resultado diferente em relação a essas questões. Sendo assim, seria possível comparar as curvas de

crescimento de defeitos de projetos no mesmo domínio, contrastando os que fazem uso de FT contra aqueles que não fazem, normalizando por um determinado intervalo de *commits*.

Finalmente, é preciso estudar e compreender melhor, com base em um número considerável de projetos de grande porte, os efeitos da adoção de FT sob a ótica do *design* de arquitetura de *software* exigida por essa técnica. É possível estudar a arquitetura utilizada em projetos maiores de forma a buscar evidências de semelhanças entre eles ou de características em comum que melhor se adaptaram ao uso de FT.

REFERÊNCIAS

BERCZUK, S. P.; APPLETON, B.; BROWN, K. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley Professional, 2003.

BIRD, J. *Feature Toggles are one of the Worst kinds of Technical Debt - DZone DevOps*. Blog. Disponível em: <https://dzone.com/articles/feature-toggles-are-one-worst>. Acesso em: 26 ago. 2018.

CHEN, L. Continuous Delivery: Huge Benefits, but Challenges Too. v. 32, n. 2, p. 50–54, mar. 2015.

COSTA, C. *et al.* Characterizing the Problem of Developers' Assignment for Merging Branches. v. 24, n. 10, p. 1489–1508, 1 dez. 2014.

COSTA, CATARINA *et al.* TIPMerge: Recommending Developers for Merging Branches. In: ACM SIGSOFT INT'L SYMP. FOUNDATIONS OF SOFTWARE ENG. (FSE), 2016, Seattle, WA, USA: [s.n.], 2016.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston, MA: Addison-Wesley, 2007.

FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. v. 17, n. 4, p. 8–17, 2013.

FOWLER, M. *BranchByAbstraction*. Blog. Disponível em: <https://martinfowler.com/bliki/BranchByAbstraction.html>. Acesso em: 20 nov. 2018.

FOWLER, MARTIN. *bliki: FeatureToggle*. Blog. Disponível em: <https://martinfowler.com/bliki/FeatureToggle.html>. Acesso em: 18 jun. 2018.

FOWLER, MARTIN. *Continuous Integration*. Blog. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 28 out. 2018.

HEYS, B. *ALM Rangers - Software Development with Feature Toggles*. Disponível em: <https://msdn.microsoft.com/en-us/magazine/dn683796>. Acesso em: 26 jul. 2018.

HTTERMANN, M. *DevOps for Developers*. New York: Apress, 2012.

LEHMAN, M. M. *et al.* Metrics and laws of software evolution-the nineties view. In: SOFTWARE METRICS SYMPOSIUM, 1997. PROCEEDINGS., FOURTH INTERNATIONAL, nov. 1997, [S.l: s.n.], nov. 1997. p. 20–32.

MACBETH, G.; RAZUMIEJCZYK, E.; LEDESMA, R. D. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. v. 10, n. 2, p. 545–555, maio 2011.

MEYER, M. Continuous Integration and Its Tools. v. 31, n. 3, p. 14–16, maio 2014.

MOURA, T.; MURTA, L. Uma técnica para a quantificação do esforço de merge. In: 6TH WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION AND MAINTENANCE, 2018, São Carlos, SP, Brasil: [s.n.], 2018.

NEELY, S.; STOLT, S. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In: 2013 AGILE CONFERENCE, ago. 2013, [S.l: s.n.], ago. 2013. p. 121–128.

PETE HODGSON. *Feature Toggles*. Disponível em: <https://martinfowler.com/articles/feature-toggles.html>. Acesso em: 8 set. 2017.

POOL, T. *Continuous Integration & Feature Branches*. 2013. Thesis – University of Tartu, Tartu, 2013.

PRUDÊNCIO, J. G. *et al.* To lock, or not to lock: That is the question. v. 85, n. 2, p. 277–289, fev. 2012.

RAHMAN, M. T. *et al.* Feature toggles: practitioner practices and a case study. 2016, [S.l.]: IEEE, 2016. p. 201–211.

RAHMAN, M. T.; RIGBY, P. C.; SHIHAB, E. The modular and feature toggle architectures of Google Chrome. p. 1–28, 2018.

REHN, C. *Continuous Integration: Aspects in Automation and Configuration Management*. Term Paper. Germany: TU Kaiserslautern, 2012.

ROMANO, J. *et al.* Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. 2006, [S.l.]: Citeseer, 2006.

SATO, D. *bliki: CanaryRelease*. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: 25 jul. 2018.

SCHERMANN, G. *et al.* An empirical study on principles and practices of continuous delivery and deployment. v. 4, p. e1889v1, mar. 2016.

SCHNEID, K. Branching Strategies for Developing New Features within the Context of Continuous Delivery. In: CONTINUOUS SOFTWARE ENGINEERING - SE, 2017, [S.l: s.n.], 2017. p. 28–35.

SHAFFER, J. P. Multiple hypothesis testing. v. 46, n. 1, p. 561–584, 1995.

SHIHAB, E.; BIRD, C.; ZIMMERMANN, T. The effect of branching strategies on software quality. In: EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2012, Lund, Sweden: ACM, 2012. p. 301–310.

SPIEGEL, M. *Estatística*. 3. ed. São Paulo, SP: McGraw-Hill, 1994.

SULLIVAN, G. M.; FEINN, R. Using Effect Size—or Why the P Value Is Not Enough. v. 4, n. 3, p. 279–282, set. 2012.

TUKEY, J. W. *Exploratory data analysis*. [S.l.]: Reading, Mass., 1977.

WILCOXON, F. Individual comparisons by ranking methods. p. 196–202, 1992.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. 2014, [S.l.]: ACM, 2014. p. 38. Disponível em: <https://dl.acm.org/citation.cfm?id=2601268>.

APÊNDICE A – TRABALHOS SELECIONADOS NO ESTUDO

Título/ Referência	Ano	Etapa	Origem	Utilizado?
RAHMAN, M. T. et al. Feature toggles: practitioner practices and a case study. 2016, [S.l.]: IEEE, 2016. p. 201–211.	2016	E1/E2	Google Acadêmico	SIM
NEELY, S.; STOLT, S. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In: 2013 AGILE CONFERENCE, ago. 2013, [S.l: s.n.], ago. 2013. p. 121–128	2013	E1/E2	Google Acadêmico	SIM
FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. IEEE Internet Computing, v. 17, n. 4, p. 8–17, 2013	2013	E1/E2	Google Acadêmico	SIM
ROSS, K. Disruptive technology. 2015.	2015	E1	Google Acadêmico	NÃO
HÜTTERMANN, M. Building Blocks of DevOps. [s.l.] Springer, 2012.	2012	E1/E2	Google Acadêmico	SIM
BATORY, D.; HÖFNER, P.; KIM, J. Feature interactions, products, and composition. ACM SIGPLAN Notices. Anais...ACM, 2011	2011	E1	Google Acadêmico	NÃO
SCHERMANN, G. et al. An empirical study on principles and practices of continuous delivery and deployment. PeerJ Preprints, v. 4, p. e1889v1, mar. 2016.	2016	E1/E2	Google Acadêmico	SIM
SCHERMANN, G. et al. Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies. Proceedings of the 17th International Middleware Conference. Anais...ACM, 2016	2016	E1	Google Acadêmico	NÃO
LAUKKANEN, E. et al. Towards continuous delivery by reducing the feature freeze period: a case study. Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. Anais...IEEE Press, 2017	2017	E1	Google Acadêmico	NÃO
HASSELBRING, W.; STEINACKER, G. Microservice architectures for scalability, agility and reliability in e-commerce. Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on. Anais...IEEE, 2017	2017	E1	Google Acadêmico	NÃO
SCHNEID, K. Branching Strategies for Developing New Features within the Context of Continuous Delivery. 2017, [S.l: s.n.], 2017. p. 28–35.	2017	E1/E2	Google Acadêmico	SIM
SCHERMANN, G.; CITO, J.; LEITNER, P. Continuous Experimentation: Challenges, Implementation Techniques, and Current Research. IEEE Software, v. 35, n. 2, p. 26–31, 2018.	2018	E1/E2	Google Acadêmico	SIM
SCHERMANN, G. Continuous experimentation for software developers. Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference. Anais...ACM, 2017	2017	E1/E2	Google Acadêmico	SIM
MASON, S. et al. A Field Guide to Continuous Delivery. 2016.	2016	E1/E2	Google Acadêmico	SIM
REHN, C. Continuous Integration: Aspects in Automation and Configuration Management. Term Paper. Germany: TU Kaiserslautern, 2012	2012	E1/E2	Google Acadêmico	SIM
AUER, F.; FELDERER, M. Current state of research on continuous experimentation: a systematic mapping study. 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Anais...IEEE, 2018	2018	E1	Google Acadêmico	NÃO
POOL, T. Continuous Integration & Feature Branches. 2013. Thesis – University of Tartu, Tartu, 2013	2013	E1/E2	Google Acadêmico	SIM
BUHL, A.; JARFORS, M. Continuous Delivery: How hard can it be? 2017.	2017	E1/E2	Google Acadêmico	SIM
OLAUSSON, M.; EHN, J. Source Control Management. In: Continuous Delivery with Visual Studio ALM 2015. [s.l.] Springer, 2015. p. 65–85.	2015	E1	Google Acadêmico	NÃO

Título/ Referência	Ano	Etapa	Origem	Utilizado?
RAHMAN, M. T.; RIGBY, P. C.; SHIHAB, E. The modular and feature toggle architectures of Google Chrome. Empirical Software Engineering, p. 1–28, 2018.	2018	E3	*Snowballing*	SIM
DE KORT, W. Setting Up Version Control. In: DevOps on the Microsoft Stack. Apress, Berkeley, CA: Springer, 2016. p. 99–136.	2016	E1	Google Acadêmico	NÃO
Operating and Evolving the EDRS Payload and Link Management System	2018	E1	Google Acadêmico	NÃO
ROSSBERG, J.; OLAUSSON, M. Release Management. In: Pro Application Lifecycle Management with Visual Studio 2012. Berkeley, CA: Springer, 2012. p. 515–532.	2012	E1	Google Acadêmico	NÃO
KARESMA, P. Scaling Agile Methods. 2016.	2016	E1	Google Acadêmico	NÃO
RAHMAN, A. A. U. et al. Synthesizing continuous deployment practices used in software development. Agile Conference (AGILE), 2015. Anais...IEEE, 2015	2015	E3	*Snowballing*	SIM
MOORE, C. M.; STEPHENS, T.; HEIN, E. Features, as well as space and time, guide object persistence. Psychonomic Bulletin & Review, v. 17, n. 5, p. 731–736, 2010.	2010	E1	Google Acadêmico	NÃO
PULKKINEN, V. Continuous deployment of software. Proc. of the Seminar. Anais...2013	2013	E1	Google Acadêmico	NÃO
ROSSI, C. et al. Continuous deployment of mobile software at facebook (showcase). Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Anais...ACM, 2016	2016	E3	Google Acadêmico	NÃO
KENNER, A. et al. TypeChef: toward type checking# ifdef variability in C. Proceedings of the 2nd International Workshop on Feature-Oriented Software Development. Anais...ACM, 2010	2010	E1	Google Acadêmico	NÃO
ARACHCHI, S.; PERERA, I. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. 2018 Moratuwa Engineering Research Conference (MERCon). Anais...IEEE, 2018	2018	E1	Google Acadêmico	NÃO
SAVOR, T. et al. Continuous deployment at Facebook and OANDA. Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on. Anais...IEEE, 2016	2016	E1	Google Acadêmico	NÃO
MUDGAL, R. et al. A Novel architecture for question classification based indexing scheme for efficient question answering. arXiv preprint arXiv:1307.6937, 2013.	2013	E1	Google Acadêmico	NÃO
NOTTONSON, K. Yahoo! Sports: Sprint 100 and Beyond. Agile Conference (AGILE), 2011. Anais...IEEE, 2011	2011	E1	Google Acadêmico	NÃO
SUNDERMANN, M. Konzeption und Einführung von DevOps in einem mittelständischen IT-Bereich. 2017.	2011	E1	Google Acadêmico	NÃO
SCHNEIDER, M.; UHLE, J. Versioning for software as a service in the context of multitenancy. 2013.	2013	E1	Google Acadêmico	NÃO
GEBERT, S. et al. Agile management of software based networks. University of Würzburg Tech. Rep., p. 493, 2015.	2015	E1	Google Acadêmico	NÃO
KÄRPÄNOJA, P. et al. Exploring Peopleware in Continuous Delivery. Proceedings of the Scientific Workshop Proceedings of XP2016. Anais...ACM, 2016	2016	E1	Google Acadêmico	NÃO
HEINILUOTO, V.-E. Roadmap for. Maintain framework. 2016.	2016	E1	Google Acadêmico	NÃO
SENAPATHI, M.; BUCHAN, J.; OSMAN, H. DevOps Capabilities, Practices, and Challenges: Insights from a Case Study. Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. Anais...ACM, 2018	2018	E1	Google Acadêmico	NÃO
ERICH, F. M. A.; AMRIT, C.; DANEVA, M. A qualitative study of DevOps usage in practice. Journal of Software: Evolution and Process, v. 29, n. 6, p. e1885, 2017.	2017	E1/E2	Google Acadêmico	SIM
RAHMAN, M. T.; RIGBY, P. Release stabilization on linux and chrome. IEEE Software, n. 1, p. 1–1, 2015	2015	E1	Google Acadêmico	NÃO

Título/ Referência	Ano	Etapa	Origem	Utilizado?
BRIZARD, T. Fake It Until You Make It. In: Broken Agile. Berkeley, CA: Springer, 2015. p. 37–42.	2015	E1	Google Acadêmico	NÃO
SOLTANI, J. Adopting continuous delivery in AAA console games. Proceedings of the 4th International Workshop on Release Engineering. Anais...ACM, 2016	2016	E1	Google Acadêmico	NÃO
RITCHIE, P. Version Control. In: Practical Microsoft Visual Studio 2015. Berkeley, CA: Springer, 2016. p. 51–77.	2016	E1	Google Acadêmico	NÃO
EICHHORN, P. MASTER THESIS THE UNI1 IMMUNE SYSTEM FOR CONTINUOUS DELIVERY. Department Informatik: Technische Fakultat, 2016	2016	E1	Google Acadêmico	NÃO
WOLFF, E. A Practical Guide to Continuous Delivery. 1. ed. [s.l.] Addison-Wesley Professional, 2017.	2017	E1	Google Acadêmico	NÃO
HEBIG, R.; DEREHAG, J. The changing balance of technology and process: A case study on a combined setting of model-driven development and classical C coding. Journal of Software: Evolution and Process, v. 29, n. 11, p. e1863, 2017.	2017	E1	Google Acadêmico	NÃO
MARTENSSON, T.; STAAHL, D.; BOSCH, J. Continuous integration impediments in large-scale industry projects. Software Architecture (ICSA), 2017 IEEE International Conference on. Anais...IEEE, 2017	2017	E1	Google Acadêmico	NÃO
MEYER, M. Continuous Integration and Its Tools. IEEE Software, v. 31, n. 3, p. 14–16, maio 2014.	2014	E3	*Snowballing*	SIM
DUVALL, P. M.; MATYAS, S.; GLOVER, A. Continuous Integration: Improving Software Quality and Reducing Risk. Boston, MA: Addison-Wesley, 2007.	2007	E3	*Snowballing*	SIM
CHEN, L. Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software, v. 32, n. 2, p. 50–54, mar. 2015.	2015	E3	*Snowballing*	SIM
MOURA, T.; MURTA, L. Uma técnica para a quantificação do esforço de merge. VI Workshop on Software Visualization, Evolution and Maintenance. Anais... In: VEM. 2018	2018	E3	*Snowballing*	SIM
SCHERMANN, G. et al. An empirical study on principles and practices of continuous delivery and deployment. PeerJ Preprints, v. 4, p. e1889v1, mar. 2016.	2013	E3	*Snowballing*	SIM
BIRD, J. Feature Toggles are one of the Worst kinds of Technical Debt - DZone DevOps. Blog. Disponível em: <https://dzone.com/articles/feature-toggles-are-one-worst>. Acesso em: 26 ago. 2018	2013	E3	*Snowballing*	SIM
HEYS, B. ALM Rangers - Software Development with Feature Toggles. Disponível em: <https://msdn.microsoft.com/en-us/magazine/dn683796>. Acesso em: 26 jul. 2018..	2014	E4	Google	SIM
PETE HODGSON. Feature Toggles. Disponível em: <https://martinfowler.com/articles/feature-toggles.html>. Acesso em: 8 set. 2017.	2016	E4	Google	SIM
FOWLER, MARTIN. bliki: FeatureToggle. Blog. Disponível em: <https://martinfowler.com/bliki/FeatureToggle.html>. Acesso em: 18 jun. 2018.	2010	E4	Google	SIM
SATO, D. bliki: CanaryRelease. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: 25 jul. 2018.	2014	E4	Google	SIM
FOWLER, MARTIN. Continuous Integration. Blog. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 28 out. 2018.	2010	E4	Google	SIM
Flags vs Branching – Feature Flags, Toggles, Controls. , 2018. Disponível em: <https://featureflags.io/feature-flags-vs-branching/>. Acesso em: 24 ago. 2018	2018	E4	Google	SIM
Continuously deliver with dark launches and feature toggles - IBM Cloud Garage Method. Blog. Disponível em: <https://www.ibm.com/cloud/garage/content/run/practice_dark_launch_feature_toggles/>. Acesso em: 24 ago. 2018.	2018	E4	Google	SIM
Feature Toggles and A/B testing. Disponível em: <https://tech.findmypast.com/feature-toggles/>. Acesso em: 24 nov. 2018.	2018	E4	Google	SIM

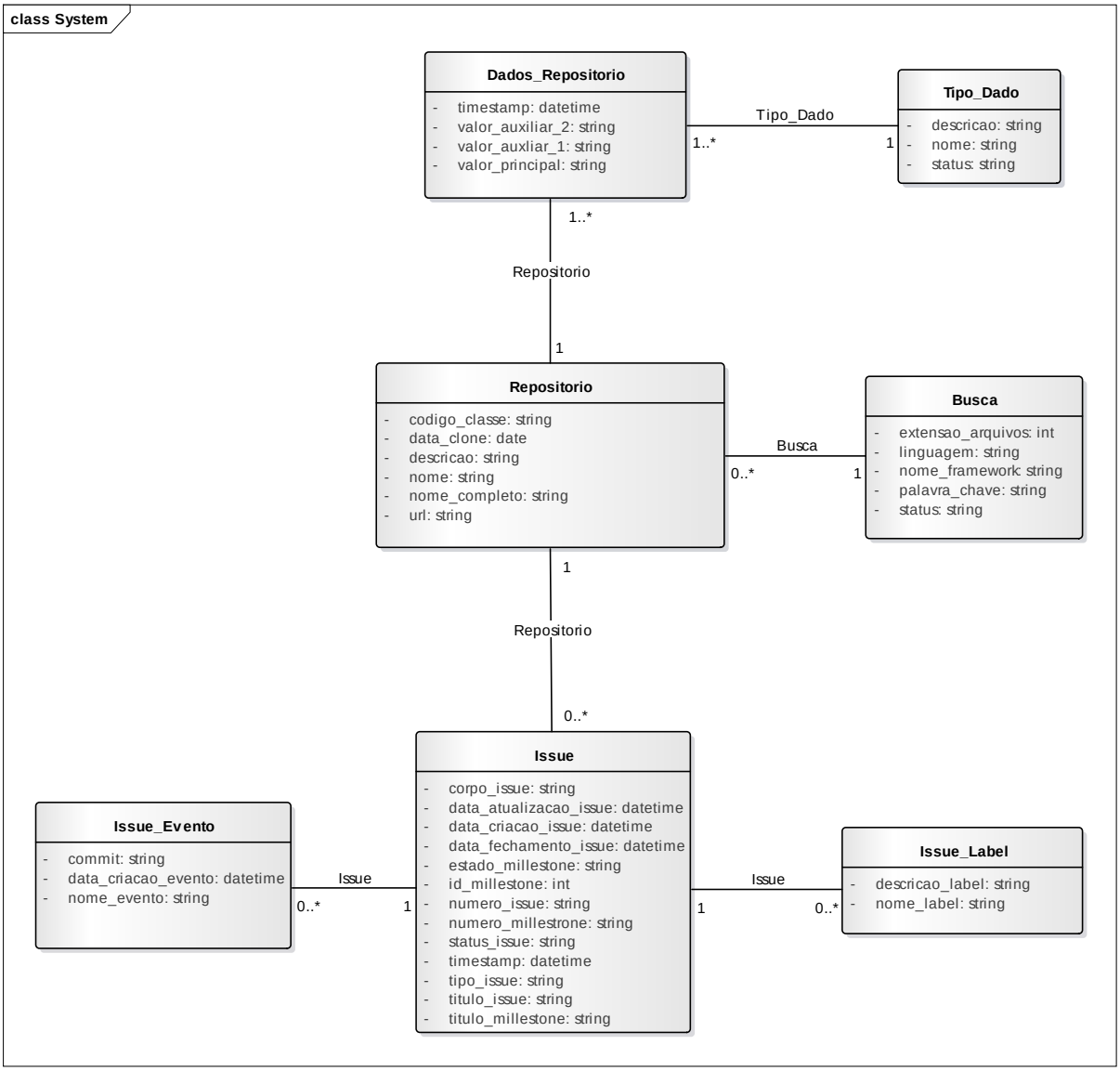
Título/ Referência	Ano	Etapa	Origem	Utilizado?
Feature Toggles. Blog. Disponível em: <http://enterprisedevops.org/feature-toggle-frameworks-list/>. Acesso em: 24 nov. 2018.	2018	E4	Google	SIM
BAKER, J. Is it a feature flag or a feature toggle?LaunchDarkly Blog, 23 abr. 2016. Disponível em: <https://launchdarkly.com/blog/is-it-a-feature-flag-or-a-feature-toggle/>. Acesso em: 12 mar. 2018	2016	E4	Google	SIM
The Travis CI Blog: Using Feature Flags to Ship Changes with Confidence. Blog. Disponível em: <https://blog.travis-ci.com/2014-03-04-use-feature-flags-to-ship-changes-with-confidence/>. Acesso em: 4 ago. 2018.	2014	E4	Google	SIM

APÊNDICE B – *FRAMEWORK* DE *FEATURE TOGGLE*

Framework	**Repositório Git**	**Linguagem**	**Versão Atual**	**Data Última *Release***	**Meio de Controle de *toggles***	**Aplicação**
Switcheroo	rhanekom/Switcheroo	C# (*.cs)	0.3	15/02/2013	Arquivo config	A/B
FeatureSwitcher	mexx/FeatureSwitcher	C# (*.cs)	2.1.0	16/02/2018	Arquivo config	A/B
FeatureToggle	jason-roberts/FeatureToggle	C# (*.cs)	4.0.2	24/10/2017	Arquivo config	A/B
Togglz	togglz/togglz	Java (*.java)	2.6.1	25/07/2018	Painel de Controle	A/B C.R.
FF4J	ff4j/ff4j	Java (*.java)	1.7.1	24/01/2018	Painel de Controle	A/B C.R.
Ericelliot/feature-toggle	Unleash/unleash	JavaScript (*.js)	3.1.4	17/12/2018	Painel de Controle	A/B C.R.
angular-toggle-switch	cgarvis/angular-toggle-switch	JavaScript (*.js) Angular	1.3.0	06/05/2015	Arquivo config	A/B
fflip	FredKSchott/fflip	JavaScript (*.js) Node JS	4.0.0	12/12/2017	Arquivo config	A/B C.R.
ember-feature-flags	kategengler/ember-feature-flags	JavaScript (*.js)	5.0.0	27/02/2018	Arquivo config	A/B C.R.
Qandidate\Toggle	qandidate-labs/qandidate-toggle	PHP	1.1.1	20/11/2017	Painel de Controle	A/B C.R.
Gutter	disqus/gutter e Gutter-Django	Python (*.py)	0.5.0	28/04/2015	Painel de Controle	A/B C.R.
Gargoyle	adamchainz/gargoyle	Python (*.py)	1.4.0	04/08/2018	Arquivo config	A/B
django-waffle	django-waffle/django-waffle	Python (*.py)	0.15.0	05/11/2018	Arquivo config	A/B
Flask-FeatureFlags	trustrachel/Flask-FeatureFlags	Python (*.py)	0.6	09/06/2015	Arquivo config	A/B
Rollout	fetlife/rollout fiverr/rollout_dashboard	Ruby (*.rb)	2.4.3	30/06/2017	Painel de Controle	A/B C.R.
featureflags	ministryofjustice/featureflags	Ruby (*.rb)	0.2.0	01/09/2015	Arquivo config	A/B
feature_flipper	theflow/feature_flipper	Ruby (*.rb)	2.0.0	09/05/2017	Arquivo config	A/B

A/B – Testes A/B e C.R. – Suporte a *Canary Releases*

APÊNDICE C – DIAGRAMA DE CLASSES



Descrição das Classes:

- Classe Repositorio: Referencia cada repositório extraído.
- Classe Dados_Repositorio: Referencia os diversos tipos de dados referente a cada repositório, tais como: *hash commit*, data do *commit*, *hash commit de merge* etc.
- Classe Tipo_Dado: Tipifica o dado extraído.
- Classe Busca: Referencia os atributos de cada *framework* utilizado nas buscas realizadas.
- Classe Issue: Referencia uma ou mais *issues* ou *pull requests* de cada repositório.
- Classe Issue_Label: Referencia as *labels* de cada *issue* de cada repositório.
- Classe Issue_Evento: Referencia os eventos de cada *issue* de cada repositório.

APÊNDICE D – CLASSIFICAÇÃO DE DEFEITOS NO GITHUB

Classificação	
Bug	*type: bug*
kind/bug	*bug (open source)*
Priority: Critical	*error*
Priority: Medium	*contrib: good first bug*
Type - Bug	*contrib: maybe good first bug*
install-bug	*hotfix*
404	*incorrect*
403	*mistake*

APÊNDICE E – PALAVRAS-CHAVE PARA IDENTIFICAÇÃO DE DEFEITOS

Palavras-chave		
fix	*bug*	*not work*
erro	*resolve*	*not respond*
problem	*does not*	*unabled to*
invalid	*exception thrown*	*falling*
defect	*not able*	*failure*
500	*hotfix*	*502*
404	*incorrect*	*cannot*
403	*mistake*	*troubleshooting*
exception	*broken*	*wrong*