

# Entendendo o Uso do Git em Equipes de Desenvolvimento de Software

Marcela Bandeira Cunha



CENTRO DE INFORMÁTICA (CIN)  
UNIVERSIDADE FEDERAL DE PERNAMBUCO (UFPE)

Recife, 2018

Marcela Bandeira Cunha

# Entendendo o Uso do Git em Equipes de Desenvolvimento de Software

Monografia apresentada ao curso Engenharia da Computação  
do Centro de Informática, da Universidade Federal de Pernambuco,  
como requisito para a obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Paulo Henrique Monteiro Borba

Dezembro de 2018



CENTRO DE INFORMÁTICA  
UNIVERSIDADE FEDERAL DE PERNAMBUCO

Trabalho de Conclusão de Curso de Engenharia da Computação intitulado ***Entendendo o Uso do Git em Equipes de Desenvolvimento de Software*** de autoria de Marcela Bandeira Cunha, aprovada pela banca examinadora constituída pelos seguintes professores:

---

Prof. Dr. Paulo Henrique Monteiro Borba  
Universidade Federal de Pernambuco

---

Prof. Dr. Fernando José Castor de Lima Filho  
Universidade Federal de Pernambuco

---

Coordenador do Curso de Engenharia da Computação Centro de Informática  
Prof. Dr. Renato Mariz de Moraes  
CIn/UFPE

Recife, 3 de Dezembro de 2018

Centro de Informática, Universidade Federal de Pernambuco  
Av. Jorn. Aníbal Fernandes, s/n, Cidade Universitária, Recife, Pernambuco, Brasil CEP: 50740-560  
Fone: +55 (81) 2126-8430

## AGRADECIMENTOS

Agradeço, primeiramente, a meus pais, Paulo Cunha e Mônica Bandeira, pelo amor, incentivo e apoio incondicional. À minha família, que sempre acreditou no meu potencial acadêmico, profissional e humano.

A meu orientador, Prof. Paulo Borba, pela orientação, pela confiança depositada no meu trabalho e pela paciência ao lidar com minhas dificuldades. Agradeço em especial a Paola Accioly, pelo auxílio na orientação do trabalho e por ter sido peça importante no desenvolvimento deste.

Agradeço à Universidade Federal de Pernambuco, em especial aos professores de graduação e seus monitores por todo o conhecimento transmitido.

A minha amiga Alice Zloccowick, minha companheira de todo o curso até nos momentos mais desafiadores, sempre me incentivando e me apoiando durante esta jornada. Também gostaria de agradecer ao meu amigo Gabriel Albuquerque pelo amparo e pelos conselhos.

A todos que ajudaram a construir, direta ou indiretamente, a pessoa que sou hoje. Muito Obrigada.

## RESUMO

O desenvolvimento de projetos de software tem como característica principal o desenvolvimento paralelo entre os desenvolvedores da equipe. Conflitos de integração de código podem ocorrer quando os desenvolvedores atualizam o repositório principal com seus trabalhos individuais. Pesquisadores têm estudado a prevenção e resolução desses conflitos, mas não consideraram as ações locais e nem os cenários de integração omitidos do histórico dos repositórios privados dos desenvolvedores. Sem esse conhecimento, estudos podem estar examinando apenas uma parte dos cenários de integração. Por este motivo, este trabalho apresenta um estudo empírico realizado nos repositórios privados de desenvolvedores com o objetivo de identificar cenários de integração, com o foco nos cenários ocultos do histórico. Foram analisados 4 projetos com um total de 14 repositórios privados de desenvolvedores. Pode-se concluir que a frequência do uso desses comandos ainda é menor em relação a quantidade de merge realizado, maneira tradicional de integrar código nos repositórios.

**Palavras-chave:** Desenvolvimento Colaborativo de Software, Sistema de Controle de Versão, Conflitos de Integração de Código

## LISTA DE FIGURAS

1	Cenário de merge. . . . .	13
2	Cenário de rebase. . . . .	14
3	Cenário de cherry-pick. . . . .	15
4	Cenário de squash. . . . .	16
5	Linhas do arquivo log. . . . .	17
6	Linha de log com o uso do rebase. . . . .	17
7	Linhas de log com o uso do rebase interativo. . . . .	17
8	Linha de log com o uso do cherry-pick. . . . .	18
9	Linhas de log com o uso do squash. . . . .	18
10	Ilustração do Experimento. . . . .	20

## LISTA DE TABELAS

1	Informações sobre os projetos. . . . .	20
2	Resultado total do Script por projeto. . . . .	21
3	Resultado do Script para o Projeto 1. . . . .	28
4	Resultado do Script para o Projeto 2. . . . .	28
5	Resultado do Script para o Projeto 3. . . . .	28
6	Resultado do Script para o Projeto 4. . . . .	28

## **Conteúdo**

<b>1</b>	<b>INTRODUÇÃO</b>	<b>9</b>
<b>2</b>	<b>MOTIVAÇÃO</b>	<b>11</b>
<b>3</b>	<b>ESTUDO DOS REPOSITÓRIOS PRIVADOS</b>	<b>13</b>
A	Comandos do Git que omitem integração de código . . . . .	13
B	Análise do Log . . . . .	16
<b>4</b>	<b>ESTUDO EMPÍRICO</b>	<b>19</b>
A	Definição do Experimento . . . . .	19
B	Projetos Analisados . . . . .	19
C	Estratégia do Experimento . . . . .	20
D	Resultados do Experimento . . . . .	21
E	Ameaças a Validade . . . . .	24
<b>5</b>	<b>TRABALHOS FUTUROS</b>	<b>25</b>
	<b>REFERÊNCIAS</b>	<b>25</b>
<b>A</b>	<b>TABELAS DO RESULTADO DO EXPERIMENTO</b>	<b>28</b>
<b>B</b>	<b>LISTA DAS PERGUNTAS DAS ENTREVISTAS</b>	<b>29</b>



# 1 INTRODUÇÃO

Desenvolvimento colaborativo é essencial para o sucesso de projetos de software. Isso é possível através de sistemas de controle de versão, os quais permitem o trabalho simultâneo de desenvolvedores nas equipes de projetos. Um dos mais populares é o Git, que permite que os desenvolvedores compartilhem código de uma forma gerenciada sem fazer uso de um repositório principal.

No Git, desenvolvedores trabalham em seu repositório privado e precisam sincronizar suas mudanças com o repositório principal. Enquanto trabalham em suas atividades, mudanças paralelas podem ocorrer, resultando em conflitos ao integrar o código. Resolver estes conflitos não é uma tarefa trivial, podendo se tornar custosa ao desenvolvedor[1]. Estes tipos de problemas não possuem uma solução exata, pois existe mais de uma maneira de solucioná-los.

Por conta desses problemas, estudos dedicaram suas pesquisas para examinar diferentes mecanismos para detecção proativa de conflitos [2, 3, 4] e ferramentas propostas para resolver conflitos de integração de código [5, 6]. Estes estudos focaram na situação em que ocorre a integração no repositório principal, ou seja, não levaram em consideração o estudo dos repositórios individuais de cada desenvolvedor. Por este motivo, algumas integrações ocorridas localmente no ambiente dos desenvolvedores não foram analisadas e nem avaliaram a influência que essas ações locais tiveram para a ocorrência de um conflito no repositório central do projeto. Além disso, existem comandos do Git que executam a integração de código, mas não deixam rastros no histórico do repositório. Por não indicar explicitamente que ocorreu uma integração, não foram considerados nos estudos anteriores. Esta exclusão pode impactar negativamente, pois a frequência de integração pode aumentar em relação ao número atual do estudo, além dos casos de conflito que não estão sendo tratados.

Com a finalidade de tratar os casos ocultos de integração do histórico do repositório, o objetivo deste trabalho é o estudo dos repositórios privados dos desenvolvedores de equipes de software a fim de entender suas práticas diárias de trabalho. Assim, poderemos especular a periodicidade destes comandos nas equipes de software e, no futuro, compreender se existe ou não uma influência nos conflitos que ocorreram no repositório principal. Dessa maneira, vai ser avaliado a frequência que os desenvolvedores utilizam dos comandos que omitem integrações de código do histórico do Git e, por fim, um foco maior em entender os motivos deles utilizarem ou não estes comandos.

Para a realização deste trabalho, utilizamos os históricos locais do Git dos desenvolvedores a fim de identificar os cenários que esses comandos acontecem e se ocorrem frequentemente. Com intuito de entender se existe um incentivo para esta prática, entrevistamos os desenvolvedores.

O restante deste trabalho está organizado da seguinte forma. Na Seção 2, nós discutimos os motivos que estimularam a realização deste estudo. A proposta de solução sugerida para este tipo de problema foi abordada na Seção 3. A Seção 4 descreve detalhadamente tanto como o experimento foi realizado, quanto o seu resultado e sua análise. Finalizando o trabalho, na Seção 5, alguns trabalhos futuros são propostos.

## 2 MOTIVAÇÃO

Com a criação de sistemas de controle de versões descentralizados, foi permitido a colaboração de maneira recíproca entre os desenvolvedores, em vez dos sistemas centralizados que a permitia apenas por meio de um repositório central. Essa forma descentralizada possibilitou a existência de um repositório privado aos desenvolvedores sem que seus commits afetassem terceiros. Dessa maneira, era possível controlar qual parte do código iria ser compartilhada, com que repositório e em que momento. Além dessas utilidades, commits poderiam ser editados, reordenados ou deletados.

Um sistema descentralizado que se tornou muito popular é o Git. Fora os requisitos promovidos por este novo gerenciamento, o Git viabiliza um histórico de todas as ações feitas pelo contribuinte de maneira automática. Com esse mundo de possibilidades, é necessário entender os riscos de compreender de maneira incorreta o uso desses novos dados. Por este motivo, Bird realizou um estudo com o objetivo de entender os benefícios e os perigos ao utilizar o Git [7].

Como o trabalho concorrente é fundamental para o desenvolvimento de software em grande escala, vários projetos adotaram o uso do Git como seu sistema de controle de versões pela sua rapidez e eficiência. Projetos de desenvolvimento de software costumam possuir um repositório central tanto para administrar os versionamentos do produto como também a inclusão de novos requisitos e melhorias. Os desenvolvedores envolvidos ativamente na produção do software dos projetos costumam possuir uma cópia local do repositório em seu ambiente de trabalho para realizar suas atividades de maneira independente. O resultado é um projeto sendo alterado continuamente em diversos aspectos por vários colaboradores ao mesmo tempo.

Cada mudança feita pelo desenvolvedor tem que ser integrada ao conteúdo do repositório central do projeto para se tornar visível ao restante dos contribuintes. A integração é feita através de ferramentas de merge, que auxiliam o processo para que aconteça de maneira rápida e que possibilite um desenvolvimento paralelo efetivo [6, 8].

Entretanto, nem toda integração ocorre de maneira bem-sucedida. Conflitos podem surgir no momento de comparação do código da versão base com as mudanças provenientes das versões dos colaboradores. Segundo o estudo empírico de Zimmermann, os conflitos ocorrem a uma taxa de 23% a 46% dos cenários totais de integração de código [4].

Com respeito aos cenários de conflitos, estudos anteriores comprovam a ocorrência frequente deles em diversos projetos, como também mostram que podem afetar negativamente a produtividade dos desenvolvedores[9, 10, 1, 11]. Isso ocorre já que resolver um conflito manualmente é uma tarefa considerada monótona, pois o colaborador terá que

descobrir como abordá-la e solucioná-la [1]. Além disso, se as soluções dos conflitos forem mal resolvidas, podem comprometer a qualidade do projeto [11, 12].

Visto que conflitos de cenário de integração acontecem regularmente, estudos foram criados para analisar os fatores que mais contribuem para este acontecimento [13, 14, 15, 16]. Estes estudos analisaram repositórios públicos de diversos projetos. O motivo de coletar esses dados através desse tipo de repositório deve-se a disponibilidade e facilidade de reter tais informações para investigação.

Por mais que estes estudos apresentam resultados significativos para a melhoria de práticas entre os colaboradores de um projeto de software, os dados coletados a partir de um repositório principal não apresentam todas as ações realizadas pelos desenvolvedores. O motivo é que todo colaborador possui um repositório local e privado para realizar suas atividades de maneira independente. Todas as ações realizadas localmente por qualquer desenvolvedor não são gravadas no histórico do repositório principal.

Por não ter coletado informações de repositórios privados, cenários de conflitos locais não foram avaliados nas pesquisas passadas. Além de não ter sido explorados, as pesquisas não informaram a influência que estes cenários, omitidos do histórico principal, podem ter nos conflitos em relação ao código base.

Por isto, além do repositório principal, seria de grande importância estudar os repositórios locais de todos os desenvolvedores envolvidos no projeto. Com todos esses dados, seria possível investigar quais os cenários de integração mais frequentes no projeto e quais ações foram realizadas por cada colaborador individualmente. Dessa maneira, seria capaz de entender com mais detalhes as práticas realizadas pela equipe de desenvolvimento e suas motivações para as abordagens escolhidas no momento de solucionar os problemas de integrações. Por fim, existiria uma possibilidade de entender de maneira mais completa os fatores que influenciam ou não os conflitos atuais e de criar novas soluções para combatê-los.

### 3 ESTUDO DOS REPOSITÓRIOS PRIVADOS

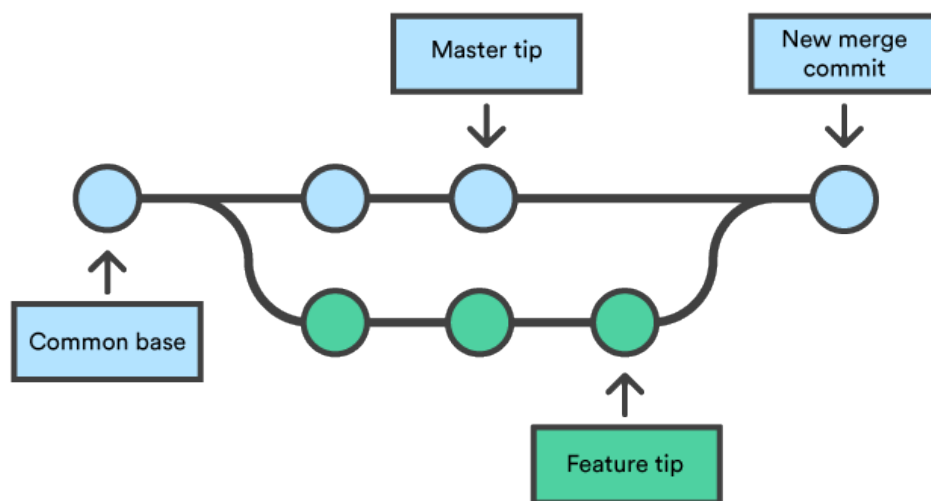
#### A Comandos do Git que omitem integração de código

Para entender o uso dos comandos do Git que escondem integração de código, esse trabalho tem como objetivo analisar repositórios privados de desenvolvedores. O estudo deve focar em identificar cenários de integração de código no histórico local, ou seja, contabilizar os cenários omitidos do histórico do repositório principal. Estes cenários podem acontecer de maneira explícita no histórico, como no comando merge do Git, mas também podem ocorrer de maneira implícita através de outros comandos.

Os estudos anteriores atuaram somente nos conflitos resultantes do comando merge do Git. Este comando é responsável por incorporar modificações feitas por uma branch secundária na branch atual de trabalho. Isso significa que o Git vai replicar as mudanças ocorridas e registrar o resultado em um novo commit.

Ao registrar este novo commit, o histórico da branch não permanece de maneira linear, tornando visível a ocorrência de integração de código. Isso acontece porque o novo commit gerado aponta para outros dois, um é o último commit da branch atual e o outro é o último commit da branch que contém as modificações. Um exemplo de merge é mostrado na Figura 1, o qual a branch master realiza o merge da branch feature.

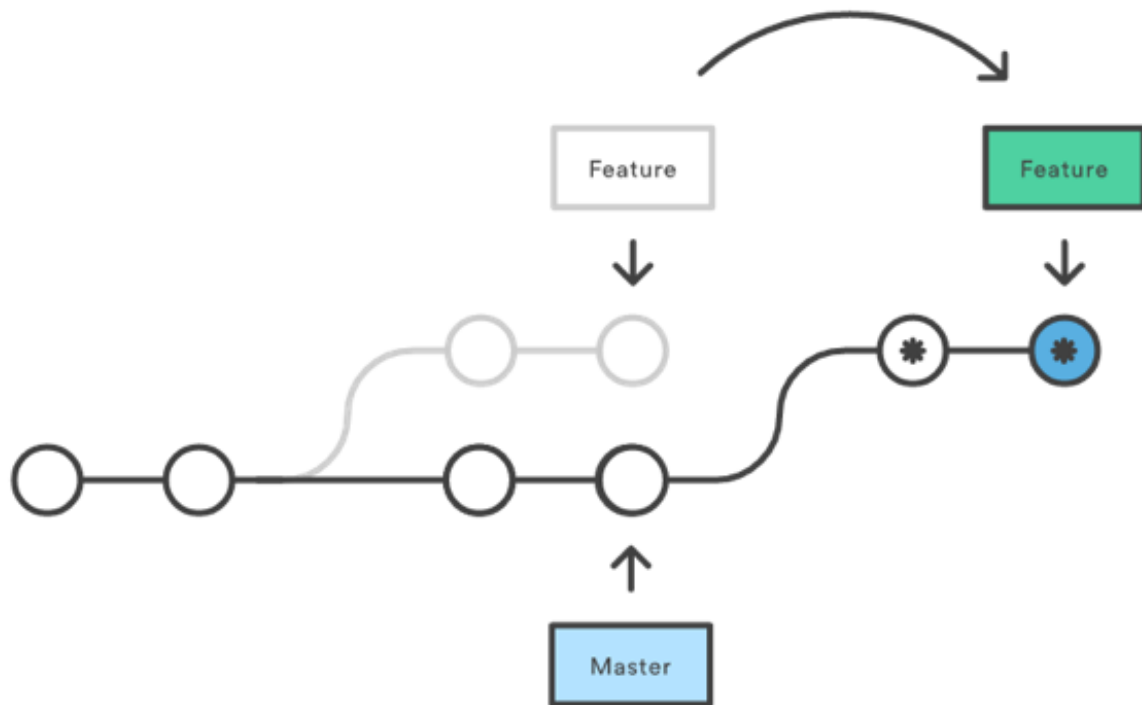
**Figura 1 - Cenário de merge.**



Fonte: [17]

Diferente do comando merge, outros comandos podem omitir a ocorrência de uma integração de código, são eles: (1) rebase, (2) cherry-pick, (3) squash, (4) stash-apply. O comando (1) rebase serve como maneira alternativa de realizar o comando merge. Ele funciona da seguinte maneira, por exemplo: o desenvolvedor termina seu trabalho em um repositório secundário, mas percebe que novas mudanças foram feitas no repositório principal e deseja atualizar sua branch atual. Então, ao efetuar o comando rebase, a primeira ação é incorporar as mudanças provenientes do repositório base e, em seguida, aplicar as modificações locais já feitas na branch atual. Este cenário pode ser visto na Figura 2.

**Figura 2 - Cenário de rebase.**



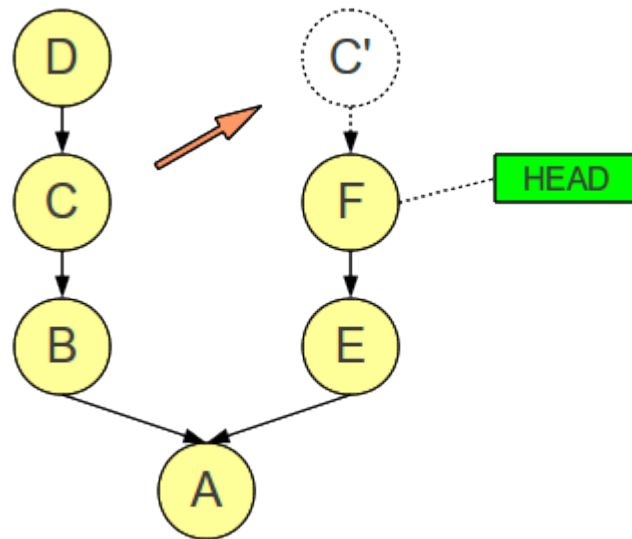
Fonte: [18]

Ao contrário do comando merge, que gera um único commit adicional interligado as mudanças, a ação de rebase cria novos commits para cada commit da ramificação original. Dessa forma, o comando rebase reescreve o histórico do repositório para deixá-lo de maneira linear. A consequência do uso deste comando é não conseguir identificar quando as alterações foram incorporadas na funcionalidade.

Já o comando (2) cherry-pick é utilizado no Git como uma maneira de reaplicar mudanças de uma branch para outra. Pode-se dizer que este comando é uma forma de realizar rebase, mas com a limitação de aplicar um único commit por vez. O processo ocorre através da cópia do conteúdo pertencente ao commit de outra branch e, logo em

seguida, é realizada uma tentativa de reaplicar o conteúdo copiado na branch atual. Por fim, é criado um novo commit para cada mudança introduzida na branch atual. O resultado da ação desse comando é uma história linear no repositório atual do desenvolvedor, sem deixar rastros que aquele commit foi de fato uma cópia de outro. Um exemplo é mostrado na Figura 3.

**Figura 3 - Cenário de cherry-pick.**

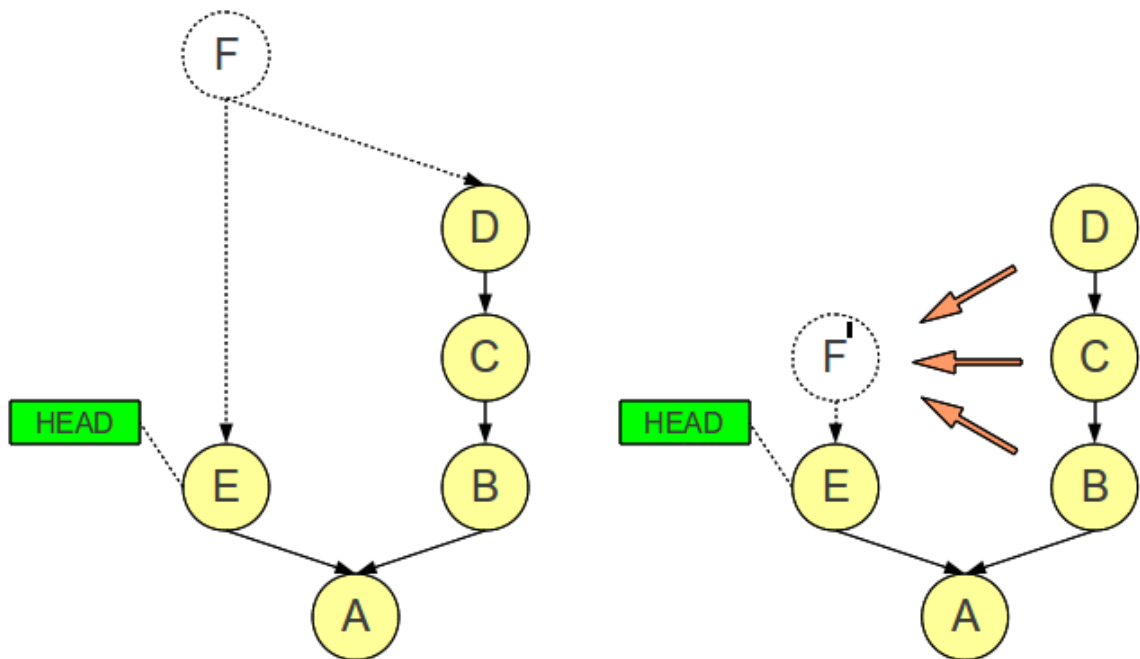


Fonte: [19]

O terceiro comando que pode esconder cenários de integração é o (3) squash. Com este comando, é possível reescrever o histórico local de um repositório porque ele permite que o desenvolvedor associe diferentes commits como se fosse um único. Além disso, se incluir um commit merge entre os commits escolhidos para realizar a ação de squash, pode-se perder qualquer rastro que aquela integração de código uma vez aconteceu. Este cenário é exibido na Figura 4, a qual mostra à esquerda o commit merge F referenciando os dois commits anteriores a ele, mas, após o squash, a referência foi perdida como é mostrado à direita na figura.

Por último, existe uma funcionalidade no Git, o stash, que permite que o usuário salve suas modificações em uma pilha para caso as necessite no futuro. Para aplicar as mudanças guardadas no código atual, é necessário utilizar o comando (4) stash-apply. Este comando não se limita a utilizá-lo na mesma branch que foi aplicado o stash, ou seja, é permitido que salve conteúdo de uma branch e o reuse em uma outra. Um exemplo muito comum desse comando é o armazenamento das modificações não comitadas a fim de atualizar o repositório através do rebase. Após a atualização, o desenvolvedor é livre para aplicar suas mudanças armazenadas através do stash-apply.

Figura 4 - Cenário de squash.



Fonte: [20]

## B Análise do Log

Com o conhecimento dos possíveis comandos que podem gerar omissões de integração de código, é necessário encontrar uma maneira de identificá-los no repositório privado do desenvolvedor. Para isto, deve-se analisar o histórico local do projeto, ou seja, o arquivo que é composto por todas as ações realizadas pelo desenvolvedor. Este arquivo é salvo e atualizado automaticamente pelo Git, de maneira simultânea a cada comando Git realizado. Conhecido como ‘log’, o arquivo é localizado no diretório Git do repositório, o qual salva diversas informações sobre o projeto.

A cada comando do Git realizado pelo desenvolvedor em seu repositório privado, uma chave única é criada para referenciá-lo [21]. Além disso, o comando atual também contém a informação de qual foi o comando anterior, ou seja, armazena a chave gerada da ação que antecedeu a atual. Dessa maneira, o Git não perde o rastro do histórico do desenvolvedor.

Para ser capaz de informar qual foi o commit mais recente, o Git mantém um ponteiro conhecido como ‘HEAD’ [21]. Este ponteiro serve para indicar tanto em que ‘branch’ que o desenvolvedor se encontra quanto o último comando realizado. O HEAD é sempre atualizado automaticamente pelo Git, sem a necessidade de intervenção manual.

O arquivo de log é preenchido por várias linhas, cada uma representando uma ação diferente realizada pelo desenvolvedor. Por linha, é possível coletar os seguintes



dados: (1) a chave única do HEAD anterior (primeira sequência de número), (2) a chave única do HEAD atual (segunda sequência de número), (3) nome cadastrado no Git do desenvolvedor, (4) e-mail cadastrado no Git do desenvolvedor, (4) o comando realizado e a (5) mensagem vinculada. Um exemplo de arquivo de log é mostrado na Figura 5:

**Figura 5 - Linhas do arquivo log.**

```
(1) (2) (3) (4) (5) (6)
0000 94c3 nome_desenvolvedor <email_desenvolvedor> 153 -0300 clone: Clone from git@github.com
94c3 ee7a nome_desenvolvedor <email_desenvolvedor> 153 -0300 commit: First commit
ee7a 49cc nome_desenvolvedor <email_desenvolvedor> 153 -0300 checkout: moving from master to test
49cc c046 nome_desenvolvedor <email_desenvolvedor> 153 -0300 commit: Add class
```

Neste exemplo, podemos perceber que o desenvolvedor realizou primeiro a ação de clonar algum repositório para sua máquina local. Após isto, executou seu primeiro commit. Por fim, criou uma nova branch e realizou mudanças nessa branch secundária.

Como é necessário estudar as ações dos desenvolvedores nos seus repositórios locais, cada arquivo de log será analisado com o objetivo de obter os cenários de integração omitidos do histórico principal do projeto. Os comandos explorados foram os citados anteriormente nesse estudo. A primeira tentativa de identificação foi feita manualmente através de uma leitura do arquivo log a fim de diferenciar os vários tipos de comandos do Git. Logo após, foi realizado testes em repositórios locais na tentativa de reproduzir os comandos citados anteriormente com intuito de reconhecer como são gravados no arquivo log.

Feito este experimento com o foco em cenários de integração, foi possível identificar a ocorrência dos comandos: (1) rebase, (2) cherry-pick e (3) squash. Respectivamente nas Figuras 6 e 7, contém exemplo de como o rebase é gravado no arquivo log, tanto na sua forma direta quanto de maneira interativa. Enquanto que o comando cherry-pick é salvo como mostrado na Figura 8.

**Figura 6 - Linha de log com o uso do rebase.**

```
1c57 3914 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase: refactor project structure
```

**Figura 7 - Linhas de log com o uso do rebase interativo.**

```
173f 9119 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (start): checkout 91194
9119 e26d nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (reword): updating HEAD
e26d 928d nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (reword): Adding parameter
928d 54f5 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (pick): Regrouping test
54f5 6938 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (pick): Adding test
6938 6938 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (finish):returning to head
```

**Figura 8 - Linha de log com o uso do cherry-pick.**

```
161b 3f38 nome_desenvolvedor <email_desenvolvedor> 153 -0300 cherry-pick: Add method
```

Diferente da maneira como o cherry-pick é exibido no log, o squash não é registrado com uma linha única e, sim, identificado através do rebase iterativo. No momento do rebase iterativo, o desenvolvedor tem a possibilidade de escolher uma lista de commits para realizar a junção. Um exemplo de squash é mostrado na Figura 9.

**Figura 9 - Linhas de log com o uso do squash.**

```
ea380 fb38b nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (start): Checkout  
fb38 ca32 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (pick): Delete test  
ca32 49d4 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (pick): Add folder  
49d4 b5d8 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (squash): Refact  
b5d8 b5d8 nome_desenvolvedor <email_desenvolvedor> 153 -0300 rebase -i (finish): Returning to head
```

Em relação ao comando (4) stash-apply, não foi possível o seu reconhecimento no histórico de log. O Git apenas registra no log o momento de criação do stash e não faz referência alguma ao aplicar o código salvo da pilha de volta na branch atual. Por este motivo, os dados relacionados a este comando não serão estudados neste trabalho.

Todo este estudo realizado envolvendo a análise de comandos do Git que possibilitam a omissão de integração de código e como identificá-los através do arquivo do histórico do repositório, tem como objetivo investigar se o desenvolvedor utiliza dessas práticas com frequência e entender por que as usa. Sendo capaz de mapear as ações do contribuinte e entender como ele atua na solução de conflitos, pode ajudar também a investigação de como conflitos locais podem afetar a produtividade do desenvolvimento de software. Além disso, realizar este experimento utilizando repositórios privados de desenvolvedores de projetos de software possibilita o estudo sobre como essas ações feitas localmente podem impactar o repositório principal do projeto.

## 4 ESTUDO EMPÍRICO

Neste trabalho, foi avaliado a frequência com que desenvolvedores utilizam comandos Git que favorecem a omissão de integração de código do histórico do projeto. Para isso, foi utilizado uma análise retrospectiva baseada no histórico do log do repositório local de cada desenvolvedor. Além dessa análise, foram realizadas entrevistas com os contribuintes envolvidos nos projetos estudados a fim de entender melhor o motivo do uso de tais comandos em ambiente de desenvolvimento de software.

As subseções seguintes descrevem a definição do experimento, os projetos analisados, a estratégia do experimento e os resultados obtidos.

### A Definição do Experimento

O objetivo do experimento é analisar o quão frequente é o uso de comandos que escondem integração de código do histórico de desenvolvimento. Foi utilizado como amostra um conjunto de arquivos de log correspondentes a cada colaborador dos projetos selecionados para o estudo.

O experimento tem como questões de pesquisa:

- **RQ1:** Com que frequência os desenvolvedores utilizam comandos que escondem integração de código?
- **RQ2:** Quais são os motivos deles terem adotado ou não esta prática?

A métrica utilizada foi a quantidade de vezes que tais comandos foram usados de acordo com o histórico individual dos desenvolvedores. Nesta avaliação, foi considerado diferentes cenários em que os comandos foram aplicados. Tudo isso para conseguir identificar o maior número de casos possíveis.

### B Projetos Analisados

O estudo foi baseado em 4 projetos. Uma empresa de inovação disponibilizou arquivos de log de 3 projetos internos e privados. Enquanto que o último projeto é de uma pesquisa acadêmica de um projeto open-source que tem como objetivo desenvolver uma ferramenta capaz de resolver conflitos de merge utilizando uma abordagem semiestruturada [22] para os projetos de software que utilizam o Git.

Em cada projeto cedido pela empresa, foi disponibilizado 4 arquivos de log, referentes a 4 desenvolvedores escolhidos de forma aleatória. Dentre os desenvolvedores selecionados, 2 deles trabalharam em mais de um dos projetos analisados no experimento. Um dos desenvolvedores trabalhou em todos os 3 projetos e o outro somente em

**Tabela 1 - Informações sobre os projetos.**

Projetos	N.º de Desenvolvedores	N.º de Desenvolvedores Únicos	Qtde de ações de Git (linhas de logs)
Projeto A	4	3	1598
Projeto B	4	2	388
Projeto C	4	2	458
Projeto D	2	2	445

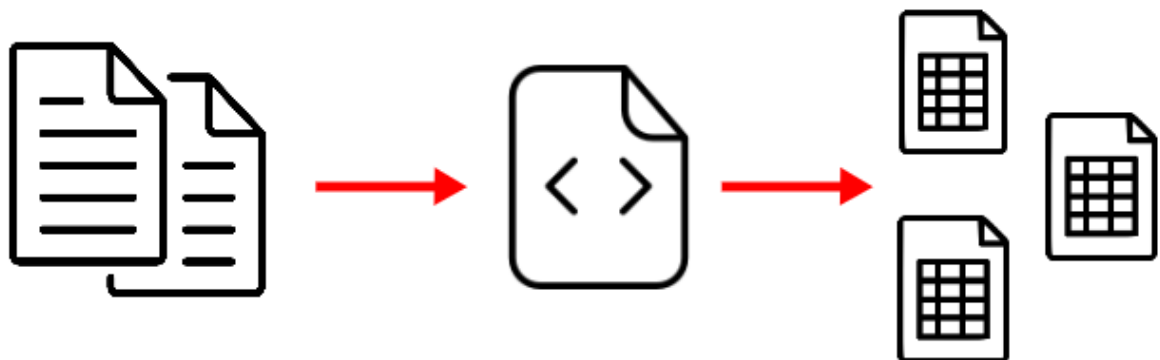
2. Enquanto que no projeto cedido pela pesquisa acadêmica possuíam 2 arquivos de log, correspondendo a 2 desenvolvedores. Por fim, foi contabilizado 14 arquivos de log, que equivalem ao histórico local do ambiente de trabalho do desenvolvedor.

A Tabela 1 contém informações sobre os projetos analisados: o número total de desenvolvedores, o número de desenvolvedores que trabalham exclusivamente naquele projeto e o total o total de ações presentes nos arquivos de log do Git realizados por esses desenvolvedores.

## C Estratégia do Experimento

O experimento consiste em identificar e contabilizar através da análise das ações nos arquivos de log os comandos do Git que podem omitir integração de código. Como o experimento envolvia 14 arquivos de log, foi criado um script para automatizar toda essa leitura. O script tinha como entrada todos os arquivos de log e gerava tabelas com os resultados das ocorrências dos comandos, como ilustrado na Figura 10.

**Figura 10 - Ilustração do Experimento.**



Para cada arquivo de log, o script era responsável por extrair diversas informações. Cada linha de log é o resultado de uma ação efetuada através de algum comando Git. Com isso, o script mapeava a cada linha de log se a ocorrência de algum dos cenários procurados pela pesquisa foi executada.

Por cada desenvolvedor, o script contabilizava seu resultado separadamente numa visão de colaborador como também contabilizava o cenário na visão por projeto, somando todas as ocorrências dos envolvidos. Por fim, todas as informações dos projetos eram somadas para gerar um cenário de visão total.

Com o resultado da frequência do uso dos comandos estudados, criamos um roteiro de perguntas para serem feitas com cada desenvolvedor do experimento. As perguntas, em geral, eram para saber se o desenvolvedor realizava ou não cada comando Git analisado no script e questionar sua resposta a fim de entender se existiam fatores positivos ou negativos no uso e descobrir se sua prática era influenciada por alguma política ou regra de uso do Git pela sua equipe de desenvolvimento. As entrevistas, realizadas de maneira individual, foram gravadas e transcritas. A seguir, um exemplo de algumas das perguntas que fizemos na entrevista. Todo o conjunto de perguntas pode ser encontrado no Apêndice B.

- Tem experiência ou conhece o comando rebase?
- Ao utilizá-lo, lidou com algum conflito?
- Já teve que lidar com conflitos resultantes do merge? Como os resolveu?
- Existe algum padrão para o fluxo de trabalho?

## D Resultados do Experimento

Nesta subseção, nós respondemos às questões de pesquisa através dos resultados obtidos neste experimento. Na Tabela 2, podemos observar o resultado da análise dos repositórios privados por visão total de projeto, enquanto que os resultados individuais de cada colaborador, pode ser visto no Apêndice A.

**Tabela 2 - Resultado total do Script por projeto.**

Projetos	Linhas de Log	Cherry-pick	Squash	Merge	Rebase	Rebase Failed	Rebase Aborted
Projeto A	1598	0	0	34	16	0	0
Projeto B	388	0	0	5	0	0	0
Projeto C	458	0	0	16	0	0	0
Projeto D	445	0	1	6	21	5	2

**RQ1: Com que frequência os desenvolvedores utilizam comandos que escondem integração de código?**

Os resultados do script mostram a frequência do uso dos comandos que omitem integração de código, exceto do stash-apply, e do comando merge através de tabelas

separadas por projeto. Cada tabela apresenta informações a respeito das ações feitas por cada colaborador separadamente. Essas ações foram contabilizadas e, em seguida, mapeadas para identificar a frequência dos cenários alvo dessa pesquisa. As Tabelas 3, 4, 5, 6 mostram o resultado dos desenvolvedores individualmente referente aos projetos A-D do estudo, enquanto que a Tabela 2 exibe o resultado do experimento por uma visão total, somando as ações de todos os colaboradores por projeto.

Os resultados obtidos mostraram que a frequência do uso de comandos que escondem integração de código em relação ao total de comandos registrados no log é muito baixa, chegando ao máximo de 5% no projeto D. Devido a isso, é comprovado que esses tipos de comandos são utilizados em projetos, como também foi explicado na seção anterior a forma que eles realizam a integração sem deixar rastros no histórico do Git.

De acordo com os dados exibidos, nenhum desenvolvedor usou o comando `cherry-pick`. Em relação ao comando `squash`, somente o desenvolvedor 10 o utilizou. Já em relação ao `merge`, os desenvolvedores 4, 7 e 8 não utilizaram. Sobre o comando `rebase`, os desenvolvedores 1,3,10 e 11 o usaram, mas só houve falha ou abandono nas execuções do `rebase` dos desenvolvedores 10 e 11, participantes do mesmo projeto. Essa falha ou aborto da execução do comando aconteceu, no pior caso, na metade das vezes que o comando foi utilizado. Isso mostra que conflitos, frutos de uma integração, aconteceram e foram resolvidos localmente.

Em relação ao projeto D, a Tabela 6 mostra que a abordagem adotada para integração de código foi através do `rebase`, excedendo o total das demais ações. Em contrapartida, os projetos A-C contabilizaram o uso `merge` em maior quantidade em relação ao uso de qualquer outro comando analisado. Embora no projeto A tenha totalizado o `merge` em maior quantidade, deve-se atentar ao detalhe que o desenvolvedor 1 foi responsável por quase todas as ações de `rebase` por parte do projeto, ultrapassando a quantidade de `merge` aplicado por ele mesmo.

Se compararmos o uso do comando `rebase`, o comando mais utilizado dentre os estudados que omitem integração, com o comando `merge`, pode-se observar na Tabela 3 que o `rebase` teve uma ocorrência de 47% em relação ao número total do uso do `merge`. Enquanto que na Tabela 6, a ocorrência do `rebase` foi mais que o triplo de ocorrência do comando `merge`. Ou seja, o uso dos comandos investigados neste trabalho poderia afetar estudos com 3 dos 14 repositórios analisados neste experimento.

O uso desses comandos pode impactar estudos de mineração, pois eles podem estar analisando um cenário incompleto sobre integração de código e, consequentemente, representando um limite inferior ao número real de conflitos resultantes do processo de integração de código. O risco de impacto é maior quanto maior for o uso desses comandos pela equipe de desenvolvimento. Um fator que contribui para uma maior taxa de

frequência de uso é se o desenvolvedor ou a equipe prezar por um histórico linear, pois o uso do comando rebase irá ser mais frequente.

Mesmo analisando e incorporando as ocorrências desses tipos de comandos que omitem o momento de integração de código, faltaria os dados do comando stash-apply. Lembrando que este comando não deixa rastros no arquivo de log, o uso dele pode resultar em conflitos e não seria possível relacioná-los a causa correta. Sem conseguir identificá-lo em um estudo, não seria possível analisar: sua frequência, sua taxa de conflitos ao aplicá-lo e como impactaria os conflitos de integração de código através do merge.

## **RQ2: Quais são os motivos deles terem adotado ou não esta prática?**

Conforme as entrevistas com os desenvolvedores, o único comando estudado neste trabalho que todos já executaram no projeto atual foi o comando merge. Em contrapartida, o comando que todos afirmaram não terem experiência ou conhecem, mas nunca o utilizaram no projeto atual, foi o comando cherry-pick. Sobre o comando squash, os desenvolvedores 10 e 11 disseram que o conheciam, mas nunca tinha utilizado, enquanto que o restante dos entrevistados admitiu não conhecer o comando.

Em relação ao comando rebase, somente os desenvolvedores 1,10 e 11 afirmaram que já o utilizaram no projeto atual. O motivo por usá-lo foi com intuito de manter o histórico linear no projeto. Contudo, o desenvolvedor 1 admitiu que o considerava mais custoso que benéfico, retornando ao uso mais frequente do merge. Em contrapartida, os desenvolvedores 10 e 11 afirmaram o uso frequente do rebase mesmo lidando com os conflitos resultantes, pois prezam por um histórico linear no projeto. Também foi citado nas entrevistas o uso do rebase por conta de um padrão adotado por uma outra empresa em um emprego anterior.

Fora os comandos possíveis de se mapear nos arquivos log, 6 dos 11 desenvolvedores entrevistados nesses projetos afirmaram utilizar o stash-apply com frequência, o que resulta em informações perdidas pelo nosso estudo. Também foi questionado sobre a existência de algum padrão estabelecido pelo projeto para adoção e uso dos comandos do Git, mas todos afirmaram utilizar um fluxo de comandos baseados na sua própria experiência profissional.

O resultado da frequência do uso desses comandos é influenciado pelos integrantes da equipe de desenvolvimento, como também pode ser influenciado por um padrão adotado de colaboração do projeto. Como citado nas entrevistas, empresas podem prezar por um histórico linear de trabalho e impor o uso desses comandos nas equipes. Por este motivo, a escolha de um padrão de desenvolvimento e do uso do Git pode influenciar nos resultados de pesquisa.

Caso não exista uma exigência a respeito do uso dos comandos do Git, cabe somente ao desenvolvedor a preferência por um histórico linear ou não. De acordo com as entrevistas, a experiência profissional foi um fator decisivo para entender o motivo dos desenvolvedores utilizarem ou não esses comandos. Colaboradores com pouca experiência ou desconhecimento não os utilizam devido à complexidade de realizá-los sem falhas ou por falta de conhecimento do que o próprio comando é capaz. Já os desenvolvedores com mais experiência têm mais facilidade de identificar cenários ideais para utilizar esses comandos e de usá-los corretamente sem muito esforço.

## **E Ameaças a Validade**

Esta seção descreve possíveis aspectos que devem ser levados em consideração para futuras replicações deste estudo. Como primeiro ponto, deve-se atentar ao fato que o histórico contido no arquivo de log não registra todas as ações feitas pelo desenvolvedor naquele ambiente. Por padrão, o Git deleta commits que não são mais referenciados após 90 dias do dia que foi realizado. Então, este estudo é limitado a analisar as ações locais de desenvolvedores feitas em um período de 90 dias. Além disso, se o desenvolvedor deletar o repositório privado, o histórico também é perdido.

A respeito dos comandos que omitem integração de código, o único que não foi possível de rastrear pela análise retrospectiva deste trabalho foi o comando `stash-apply`. Por conta disso, dados estão sendo coletados de forma incompleta e conclusões não estão tratando todos os casos. Como solução para este problema, é encontrar uma maneira de coletar essa informação de maneira simultânea ao momento que o desenvolvedor a realiza.

Além desses fatores, a escolha de projeto deve ser realizada de maneira cautelosa. Deve-se atentar ao fato de a equipe de desenvolvimento seguir um fluxo padrão do uso dos comandos do Git para não ter seus dados viciados a uma rotina fixa preestabelecida. Caso contrário, ter atenção aos colaboradores da equipe de desenvolvimento do projeto escolhido, pois pessoas com pouca experiência profissional ou pouco tempo de uso de Git podem não trazer nenhuma conclusão nova, já que não utilizam nenhum dos comandos estudados neste trabalho.



## 5 TRABALHOS FUTUROS

Em ambientes de desenvolvimento colaborativos, desenvolvedores trabalham simultaneamente em diversas tarefas de maneira independente do outro. Como resultado, podem se deparar com conflitos no momento de integração de código com o repositório principal. Estudos anteriores indicam que esses conflitos ocorrem frequentemente e pode impactar negativamente a produtividade do desenvolvedor. Neste trabalho, apresentamos um estudo com foco nas ações realizadas em repositórios privados dos desenvolvedores, principalmente nas ações de integração de código com intuito de descobrir o quão frequente é o uso de comandos que omitem a integração do histórico do repositório.

Ao realizar o experimento, existia a limitação de não mapear o comando `stash apply` no histórico do Git, mas, ao realizar as entrevistas, pode-se perceber que o uso era comum pela maioria dos desenvolvedores dos projetos analisados. Além do uso frequente deste comando, o cenário mais comentado foi que ele era executado após realizar o comando `pull` no repositório de trabalho. Por esta razão, como trabalho futuro, propomos um estudo mais detalhado em relação a este comando e como relacioná-lo ao uso de outros, como no uso do `pull`.

Também é de grande interesse estudar como que os comandos que omitem integração de código podem estar contribuindo para os conflitos de integração do repositório principal e realizar uma comparação com a frequência deles com o uso do comando `merge`. Além disso, analisar em quanto aumentaria a ocorrência de conflitos se esses comandos fossem considerados nos estudos.

## REFERÊNCIAS

- [1] KASI, BK.; SARMA, A. **Cassandra: proactive conflict minimization through optimized task scheduling**. In *Proceedings of the 2013 international conference on software engineering*. IEEE, 2013.
- [2] BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. **Proactive detection of collaboration conflicts**. In *International Symposium and European Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 168–178. IEEE, 2011.
- [3] SARMA, A.;. PhD thesis.
- [4] GUIMARÃES, L.; SILVA, A. R. **Improving early detection of software merge conflicts**. In *International Conference on Software Engineering (ICSE)*, page 342–352. IEEE, 2012.
- [5] NISHIMURA, Y.; MARUYAMA, K. **Supporting merge conflict resolution by using fine-grained code change history**. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page 661–664. IEEE, 2016.
- [6] CARVALHO, F.B.S. DE; LOPES, W.T.A.; ALENCAR, MARCELO S. **A state-of-the-art survey on software merging**. *IEEE Transactions on Software Engineering (TSE)*, 28(5):449–462, 2002.
- [7] BIRD, C. ET AL. **The promises and perils of mining git**. In *MSR '09 Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, page 1–10. IEEE, 2009.
- [8] LESSENICH, O.; APEL, S.; LENGAUER, C. **Balancing precision and performance in structured merge**. *Autom. Softw. Eng. ASE*, 22(3):367–397, 2014.
- [9] ZIMMERMANN, T. **Mining Workspace Updates in CVS**. In *MSR '07 Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE, 2007.
- [10] BRUN, Y.; HOLMES, R.; ERNST, M.D.; NOTKIN, D. **Early detection of collaboration conflicts and risks**. *IEEE Trans Softw Eng*, 39:1358–1375, 2013.
- [11] YING, X. ET AL. **The effect of branching strategies on software quality**. In *Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement*. ACM, 2012.
- [12] ESTLER, H.C.; NORDIO, M.; FURIA, C.A.; MEYER, B. **Awareness and merge conflicts in distributed software development**. In *International Conference on Global Software Engineering*. ACM/IEEE, 2014.

- [13] GHIOTTO, G.; MURTA, L.; BARROS, M.; HOEK, A. **On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub.** In *IEEE Transactions on Software Engineering*. IEEE, 2018.
- [14] NGUYEN, H.L.; IGNAT, C. **An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects.** *Computer Supported Cooperative Work*, 27:741–765, 2018.
- [15] ACCIOLY, P.; BORBA, P.; SILVA, L.; CAVALCANTI, G. **Analyzing conflict predictors in open-source Java projects.** In *MSR '18 Proceedings of the 15th International Conference on Mining Software Repositories*, pages 576–586. ACM, 2018.
- [16] MCKEE, S.; NELSON, N.; SARMA, A. **Software Practitioner Perspectives on Merge Conflicts and Resolutions.** In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017.
- [17] **Git merge.** <https://cs.atlassian.com/git/tutorials/using-branches/git-merge>. Acessado Em: 30/11/2018.
- [18] **To Squash or Not to Squash?** [https://seesparkbox.com/foundry/to\\_squash\\_or\\_not\\_to\\_squash](https://seesparkbox.com/foundry/to_squash_or_not_to_squash). Acessado Em: 30/11/2018.
- [19] **How to perform a cherry-pick.** <http://www.syntevo.com/doc/display/SG/How+to+perform+a+cherry-pick>. Acessado Em: 30/11/2018.
- [20] **How to perform normal merges and squash merges.** <https://www.syntevo.com/doc/display/SG/How+to+perform+normal+merges+and+squash+merges>. Acessado Em: 30/11/2018.
- [21] CHACON, S.; STRAUB, B. **Pro Git.** Apress, 2nd edition edition, 2014.
- [22] APEL, S. ET AL. **Semistructured Merge in Revision Control Systems.** In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010.

## A TABELAS DO RESULTADO DO EXPERIMENTO

Tabela 3 - Resultado do Script para o Projeto 1.

Desenvolvedores	Linhas de Log	Cherry-pick	Squash	Merge	Rebase	Rebase Failed	Rebase Aborted
Desenvolvedor 1	1048	0	0	15	15	0	0
Desenvolvedor 2	93	0	0	3	0	0	0
Desenvolvedor 3	294	0	0	12	1	0	0
Desenvolvedor 4	163	0	0	4	0	0	0

Tabela 4 - Resultado do Script para o Projeto 2.

Desenvolvedores	Linhas de Log	Cherry-pick	Squash	Merge	Rebase	Rebase Failed	Rebase Aborted
Desenvolvedor 5	42	0	0	3	0	0	0
Desenvolvedor 6	248	0	0	2	0	0	0
Desenvolvedor 7	38	0	0	0	0	0	0
Desenvolvedor 4	60	0	0	0	0	0	0

Tabela 5 - Resultado do Script para o Projeto 3.

Desenvolvedores	Linhas de Log	Cherry-pick	Squash	Merge	Rebase	Rebase Failed	Rebase Aborted
Desenvolvedor 8	16	0	0	0	0	0	0
Desenvolvedor 7	31	0	0	1	0	0	0
Desenvolvedor 9	41	0	0	2	0	0	0
Desenvolvedor 4	370	0	0	13	0	0	0

Tabela 6 - Resultado do Script para o Projeto 4.

Desenvolvedores	Linhas de Log	Cherry-pick	Squash	Merge	Rebase	Rebase Failed	Rebase Aborted
Desenvolvedor 10	104	0	1	2	6	3	1
Desenvolvedor 11	341	0	0	4	16	2	1

## B LISTA DAS PERGUNTAS DAS ENTREVISTAS

- Quanto tempo como desenvolvedor no projeto?
- Tem experiência ou conhece o comando rebase? Ao utilizá-lo, lidou com algum conflito?
- Tem experiência ou conhece o comando squash? Ao utilizá-lo, lidou com algum conflito?
- Tem experiência ou conhece o comando cherry-pick? Ao utilizá-lo, lidou com algum conflito?
- Costuma utilizar o stash-apply? Em que situações utilizou este comando?
- Acontece muitos conflitos de integração durante o desenvolvimento?
- Esses conflitos são resolvidos localmente no seu repositório antes de integrar o código com o repositório principal?
- Na sua opinião, qual comando costuma resultar em conflitos de integração de código?
- No seu projeto, existe algum padrão adotado por todos os membros? Ou é uma escolha pessoal o uso dos comandos de git?
- Já teve que lidar com conflitos resultantes do merge? Como os resolveu?
- Como você costuma resolver os conflitos de merge?
- Você costuma consultar o time para resolver os conflitos de merge?
- Existe algum padrão para o fluxo de trabalho?