Universidade Federal de Pernambuco
Centro de Informártica

Graduação em Ciência da Computação

A vida privada dos conflitos de merge: replicação e análise qualitativa

Matheus Luiz Borba Alves da Silva

Trabalho de Graduação

Recife
20 de Outubro de 2022

Universidade Federal de Pernambuco
Centro de Informártica

Matheus Luiz Borba Alves da Silva

A vida privada dos conflitos de merge: replicação e análise qualitativa

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informártica da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Dr. Paulo Henrique Monteiro Borba*

Recife
20 de Outubro de 2022

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Borba Alves, Matheus Luiz.
A vida privada dos conflitos de merge: replicação e análise qualitativa / Matheus Luiz Borba Alves. - Recife, 2022.
33 : il., tab.

Orientador(a): Paulo Henrique Monteiro Borba
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2022.
Inclui referências, apêndices, anexos.

1. Conflitos de integração. 2. Desenvolvimento colaborativo de software. 3. Git. I. Monteiro Borba, Paulo Henrique. (Orientação). II. Título.

000 CDD (22.ed.)

Agradecimentos

Mais de uma vez questionei se um dia seria capaz de estar escrevendo essa seção de agradecimentos, que loucura é finalmente estar aqui, já consigo sentir um peso gigante sendo retirado das minhas costas. Agora mais leve, não posso deixar de refletir o quão pesado tudo isso foi. Fico feliz por finalmente estar concluindo minha graduação, e ao mesmo tempo triste, não era pra ser tão pesado assim, né?

Quero deixar explícito que os agradecimentos não seguem uma ordem específica. Dito isso, quero começar pela minha família, eu simplesmente amo todos eles, amo como durante todo esse tempo eles andaram numa linha tênue em demonstrar interesse pelo que eu fazia e queriam sempre demonstrar apoio, ao mesmo tempo que eles não entendiam nada do que eu falava kkkkkkkkkkkk, quero agradecer muito a minhas duas irmãs, Bibi e Ray, a meus pais, Quelli e Jadiael, a família Borba Alves. Em nenhum momento nesses anos de faculdade eu me senti pressionado por eles a nada, ao mesmo tempo que eles deixavam claro o quanto eles queriam que esse momento chegasse. Adendo especial ao meu vô Joca, em breve fará 100 anos, a única pessoa que eu escondo minhas tatuagens pra ir visitar, não quero te deixar triste vô. E a minha vó Carmelita, analfabeta, que trabalhou muito, muito, muito, e agora tem mais um neto com curso superior. Vocês são sem dúvidas, inspirações pra mim.

Que sorte a minha de ter os amigos que tenho. Jullyo e Allyson, a gente se conheceu no ensino médio e tamo junto até hoje, muito foda! E a todos amigos que estiveram na luta comigo no busão pra Carpina, amo vocês. Mário, o primeiro amigo que fiz na faculdade, ainda bem que o PC do grad estava quebrado naquele dia, não consigo imaginar como eu estaria hoje sem você, obrigado por me introduzir aos outros dois melhores amigos que fiz, Claudinho e Guila, vocês literalmente me salvaram, sem vocês, eu garanto que não estaria aqui.

Queria agradecer ao fato de eu ter reprovado em cálculo I logo no primeiro período, eu não teria conhecido grandes amigos se eu não tivesse atrasado meu curso, E que amigos que encontrei, são vários, mas em especial, vou citar Éden pois é isso que ele é, especial. Queria agradecer por eu não ter passado na Apple Academy, assim, eu consegui entrar no PET Informática e conheci várias pessoas incríveis, pqp amo todos vocês, valia a pena perder o busão pra ficar nas reuniões. Aos que participaram do programa antes de mim, comigo e depois de mim, amo vocês, o PET salvou minha graduação, direta e indiretamente. Obrigado Éden, Edjan, Claudinho, Cardoso, Luan, Tato, Lari, Ullayne, Malu, Marcela, Lucas Santana, Higor, Pêu, Rods, Valdemiro, Basi, Rossi, todos e claro, Simone Santos que além de tutora do PET foi orientadora do meu primeiro artigo publicado. Obrigado.

Na graduação eu conheci pessoas incríveis que me ajudaram muito, cada virada de noite foi mais legal graças a vocês("Helou Pessoas"e afins) e tantos outros que acabei conhecendo de forma indireta e hoje fico muito feliz de ser próximo, obrigado amigos.

Queria agradecer a quem me ajudou mais de perto nesse tcc. O apoio do meu orientador, Paulo Borba, junto ao auxilio de Marcela Cunha foram triviais pra eu conseguir chegar aqui. Aproveitar pra agradecer a alguns professores que foram muito importantes pra minha graduação, Kiev Gama, Leopoldo Teixeira e Paulo Borba, além de muito bons educadores, a empatia que vocês tiveram comigo no decorrer dessa graduação foi incrível, e muito importante principalmente no período da pandemia da covid-19. Obrigado.

Não posso deixar de agradecer a minha terapeuta, Thais Bastos, que grande ajuda você foi e ainda é pra mim. A todos que encontrei nesta caminhada, aos funcionários do RU, da UFPE como um todo, em específico aos tios e tias da limpeza do CIn, que sempre limpavam o grad que eu estava dormindo por último. Obrigado.

Aos meus professores da ETEMERB, que me introduziram a área de TI e me preparam para estar aqui, tive sorte de ter seguido esse caminho, e que baita sorte a minha...

Conhecimento sem visão só te faz mais um burro convicto.

—CESAR MC

Resumo

Para a grande maioria dos projetos de software o sucesso está atrelado ao desenvolvimento colaborativo. Dito isso, conflitos de integração podem surgir quando um desenvolvedor decide integrar suas modificações com outros desenvolvedores em um repositório remoto. Conflitos podem acarretar na diminuição da produtividade, diminuição de qualidade de código e inserção de bugs em ambientes de produção.

Graças a estudos realizados previamente, a frequência de comandos de integração e conflitos já foram analisados. Porém, na maioria das análises, o foco tende a ser apenas em códigos disponibilizados em repositórios públicos. Cenários de integração de código podem ser perdidos no histórico remoto dos repositórios devido à existência de comandos como o git rebase, que reescreve o histórico de commits do Git. Portanto, esses estudos podem estar analisando apenas uma parte dos conflitos reais e casos de integração de código.

Através da análise de repositórios locais, podemos acessar cenários de integração de código que não seria possível caso o foco fosse apenas nos repositórios públicos do GitHub. O objetivo deste estudo é trazer mais visibilidade para a importância da análise local de repositórios para fins de investigar diversos cenários de integração de código e suas relações com a ocorrência de conflitos.

Examinamos um total de 35 arquivos de git reflog de 16 projetos diferentes pertencentes a duas organizações, no total foram coletados logs de 17 desenvolvedores. Foram conduzidas 8 entrevistas semi-estruturadas, 4 colaboradores de cada organização, com objetivo de entender mais a fundo a relação entre o uso dos comandos de integração, o fluxo de trabalho e diretrizes de cada projeto estabelecidos pelas empresas e a ocorrência de conflitos.

Foi detectado que o uso de comandos que ofuscam a integração de código são mais utilizados por desenvolvedores, próximo ou acima dos 3 anos de experiência. Além de conseguir apontar quais características dos projetos podem influenciar na ocorrência de conflitos, como por exemplo: o uso de testes automatizados. Vimos também que a demora para revisão de código está relacionada a ocorrência de conflitos, mais chances do código ter sido alterado, assim é de extrema importância que o processo de integração de código seja feito de forma rápida e efetiva. O planejamento prévio das tarefas que serão realizadas, a preocupação com a estrutura e tamanho dos PRs e o uso de testes automatizados ajudam a diminuir a ocorrência de conflitos pois agilizam o processo de revisão de código e integração de mudanças.

Palavras-chave: Conflitos de integração, Desenvolvimento colaborativo de software, Git

Abstract

For the vast majority of software projects, success is linked to collaborative development. That said, merge conflicts can arise when a developer decides to merge their modifications with other developers into a remote repository. Conflicts can lead to decreased productivity, decreased code quality and bugs in production environments.

Thanks to previous studies, the frequency of integration commands and conflicts have already been analyzed. However, in most analyses, the focus tends to be only on code available in public repositories. Code integration scenarios can be lost in the remote history of repositories due to the existence of commands like git rebase, which rewrites the history of Git commits. Therefore, these studies may be analyzing only a part of the actual conflicts and code integration cases.

By analyzing local repositories, we can access code integration scenarios that would not be possible if the focus was only on public GitHub repositories. The objective of this study is to bring more visibility to the importance of local analysis of repositories in order to investigate different scenarios of code integration and their relationship with the occurrence of conflicts.

We examined a total of 35 git reflog files from 16 different projects owned by two organizations, in total we collected logs from 17 developers. Eight semi-structured interviews were conducted, with 4 employees from each organization that was collected in the sample, to try to understand more deeply the relationship between the use of integration commands, the workflow and guidelines of each project established by the companies and the occurrence of conflicts.

It was detected that the use of commands that obfuscate the code integration are more used by developers, close to or above 3 years of experience. In addition to being able to point out which project characteristics can influence the occurrence of conflicts, such as: the use of automated tests. We also saw that the delay for code review is related to the occurrence of conflicts, the more chances of the code having been changed, so it is extremely important that the code integration process is done quickly and effectively. Prior planning of the tasks that will be performed, concern with the structure and size of PRs and the use of automated tests help to reduce the occurrence of conflicts as they speed up the process of code review and integration of changes.

Keywords: merge conflits, collaborative software development, Git

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

O desenvolvimento colaborativo é sem dúvidas um dos principais pilares para o sucesso de projetos de software. Isso só é possível graças a sistemas de controle de versão, permitindo com que os desenvolvedores trabalhem de forma simultânea em um mesmo projeto. Dentre várias opções, Git é um dos mais usados [1]. Nele existe o repositório remoto que é normalmente o principal, e cada desenvolvedor possui uma cópia local, seu repositório privado. À medida que mudanças são feitas em paralelo por mais de um desenvolvedor em seus respectivos repositórios, mais as chances de ocorrer algum problemas ao tentar integrar o código local ao remoto, esses são chamados conflitos de merge [2, 3, 4, 5].

Solucionar conflitos de integração de código não é necessariamente uma tarefa fácil, por mais que existam soluções triviais, alguns delas podem gerar dor e custo ao colaborador. O desenvolvedor pode também resolver os conflitos de forma equivocada, acarretando em outra série de problemas, como, por exemplo, a introdução de bugs no ambiente de produção. Conflitos de integração podem impactar além da produtividade, na qualidade geral do código [6]. Por conta disso, e também pela quantidade de vezes que conflitos desse tipo podem acontecer [2, 3, 4, 5], existem vários estudos que abrangem desde o aspecto da detecção proativa de conflitos [7, 8, 9] até propor ferramentas cujo objeto é a resolução eficaz deles [10, 11, 12].

Com o objetivo de mitigar o impacto negativo dos conflitos, esse tópico já foi e ainda é estudado na academia. Porém, a maioria dos estudos focam apenas na análise de conflitos que acontecem em repositorios remotos [13, 14]. O problema é que essa análise não conta com todos os cenarios feitos por comandos que reescrevem o histórico do Git, ou seja, todos os comandos que ofuscam integração de código feitos localmente, são ignorados. Assim faltam mais estudos que analisam o impacto dessas ações locais na ocorrência de conflitos [15, 16].

Por conta disso, o foco deste trabalho é estudar os casos ocultos de integração de código de equipes de desenvolvimento de software. O processo de análise segue o que foi feito anteriormente [15, 16], utilizando os históricos locais do Git com objetivo de identificar quando e como tais comandos são usados e o porquê. Comparando assim se algo mudou em relação aos resultados dos estudos anteriores, além de tentar entender qual impacto das diretrizes e características de cada projeto, junto ao fluxo de trabalho dos desenvolvedores, e como isso pode interferir na ocorrência de conflitos.

O trabalho está organizado da seguinte maneira. Na Seção 2, nós discutimos os motivos que estimulam a realização deste estudo. A metodologia aplicada será apresentada na Seção 3, seguida dos resultados, conclusão e trabalhos futuros nas Seções 4, 5 e 6 respectivamente.

CAPÍTULO 2

Motivação

Graças a sistemas de controle de versão descentralizados, é possível a existência de um repositório privado para cada desenvolvedor de maneira que seus *commits* não afetem terceiros diretamente. Assim é possível ter controle de qual parte do código irá ser compartilhada, em que momento e para qual repositório remoto ela vai. Além destas funcionalidades, *commits* podem ser editados, deletados, e até reordenados. Existem ferramentas de controle de versão que viabilizam um histórico completo de todas as ações feitas de maneira automática, como o Git. Devido a essas funcionalidades é importante entender os benefícios e os perigos do uso dessa ferramenta [17].

Cada mudança feita localmente por um desenvolvedor precisa ser integrada ao repositório remoto, normalmente o principal. Dito isso, nem todas as integrações são bem sucedidas, o que significa o surgimento de conflitos. A ocorrência frequente de conflitos em projetos de software pode afetar negativamente a produtividade dos desenvolvedores [3, 4, 18]. Isso acontece pois resolver um conflito exige que o colaborador descubra como aborda-lo e resolver-lo [3]. Vimos também que resolução de conflitos feitas da maneira errada podem comprometer a qualidade do projeto, além da introdução de bugs [6, 19].

A quantidade de conflitos pode variar de acordo com as práticas e diretrizes de cada projeto [14, 20]. Alguns estudos mostram zero conflitos resultantes de um comando de integração [14], outros podem chegar até 50%[7]. Em média estudos afirmam que a taxa de conflitos varia entre 10% e 20% [3, 7]. Porém a grande maioria parte desses estudos analisam apenas repositórios públicos hospedados no Github [3, 7, 10, 14].

Apesar de trazer uma quantidade significativa de evidências, esses estudos se baseiam apenas em repositórios remotos e focam em conflitos de integração de código proveniente apenas do uso do git merge [15, 16], o que é uma ameaça a validade [16]. Porém existem várias maneiras de integrar código usando git [21], algumas delas que não deixam registros no histórico remoto, são elas: (1) *rebase*, (2) *cherry-pick*, (3) *squash*, (4) *stash-apply*.

Antes de entrarmos em detalhes em cada comando, vale esclarecer como o merge funciona: Ao utilizá-lo, o git gera um novo commit que possui todas as mudanças feitas na branch que está sendo integrada. Ao registrar o novo commit, o histórico não permanece linear, deixando explícita a integração de código, um exemplo de merge é mostrado na Figura 2.1.

Ao usar o comando (1) de *rebase*, as modificações de uma branch são combinadas com as alterações de outra branch, servindo assim como a nova base para as alterações. O *rebase* gera novos *commits* para cada commit na branch original que está sendo integrada, diferente

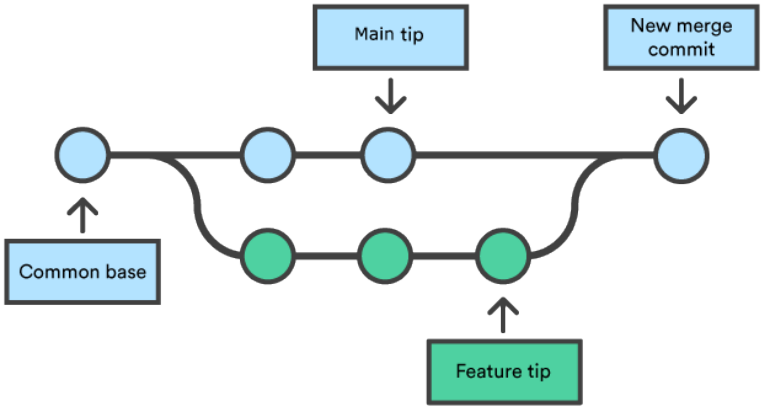

Figura 2.1 Cenário de merge [22]

do merge, que gera um commit adicional com o código integrado. Na Figura 2.2, podemos ver a modificação do histórico do repositório, tornando-o mais simples de examinar graças a linearidade.

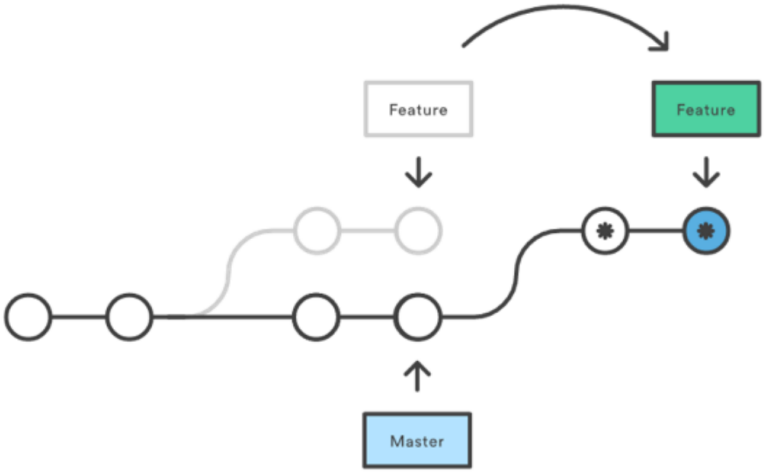

Figura 2.2 Cenário de *rebase* [23]

Para aplicar modificações específicas (*commits*) de uma branch para outra, use o comando (2) *cherry-pick*. O resultado final é um histórico linear, que ofusca o fato de que os *commits* reaplicados são duplicatas, pois foram produzidos separadamente numa branch diferente da que está sendo integrada. Como visto na Figura 2.3.

Já o (3) *squash*, pode combinar muitos *commits* em um enquanto apaga todas as evidências das contribuições anteriores. Quando um commit de merge é um dos *commits* que foi combinado pelo *squash*, é a situação especial em que o *squash* pode ofuscar a integração do código. Como

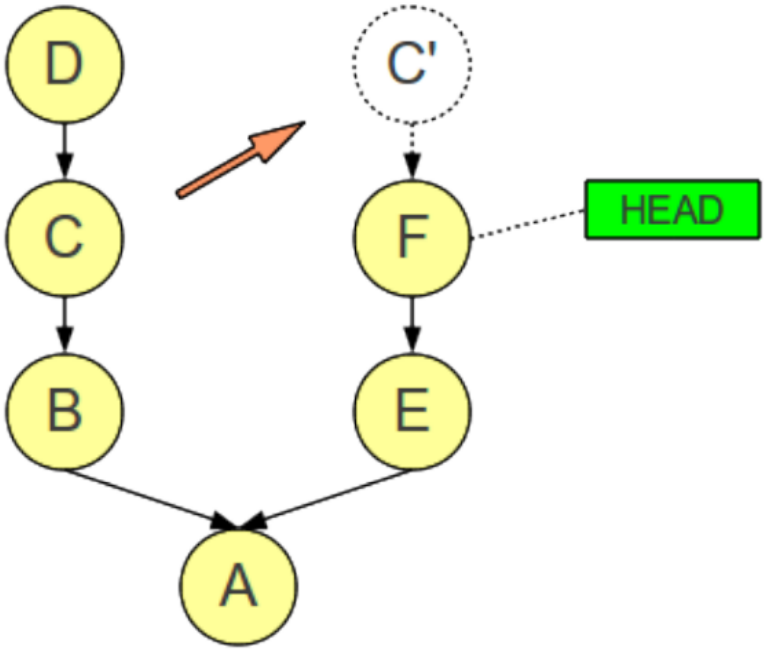

Figura 2.3 Cenário de *cherry-pick* [15]

resultado, o registro de confirmação de mesclagem ou a trilha de integração de código que foi preservada no histórico inicial é perdida, veja na Figura 2.4.

Por fim, o stash permite o armazenamento temporário numa pilha, que podem ser posteriormente aplicados a uma branch com o comando de (4) *stash-apply*, realizando assim a integração de código. Um uso comum do comando é quando se é necessário atualizar a branch local, assim é usado o stash para salvar as modificações, e após a branch ser atualizada, o *stash-apply* é usado.

Assim focar apenas para repositórios públicos e remotos é ignorar parte dos cenários de integração de código que acontecem no dia a dia de um projeto. Por conta disso estudos que analisam repositórios locais são de extrema importância e podem ajudar ainda mais a entender quais práticas são adotadas pelos desenvolvedores das equipes e quais delas influenciam ou não na ocorrência de conflitos [15, 16].

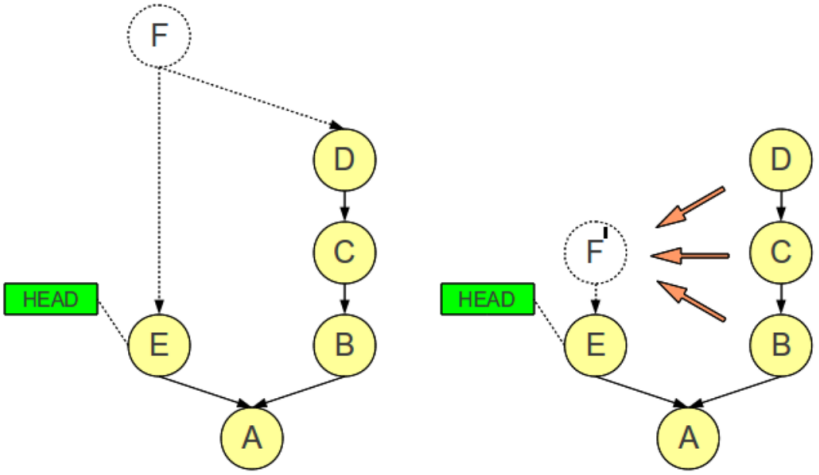

Figura 2.4 Cenário de *squash* [15]

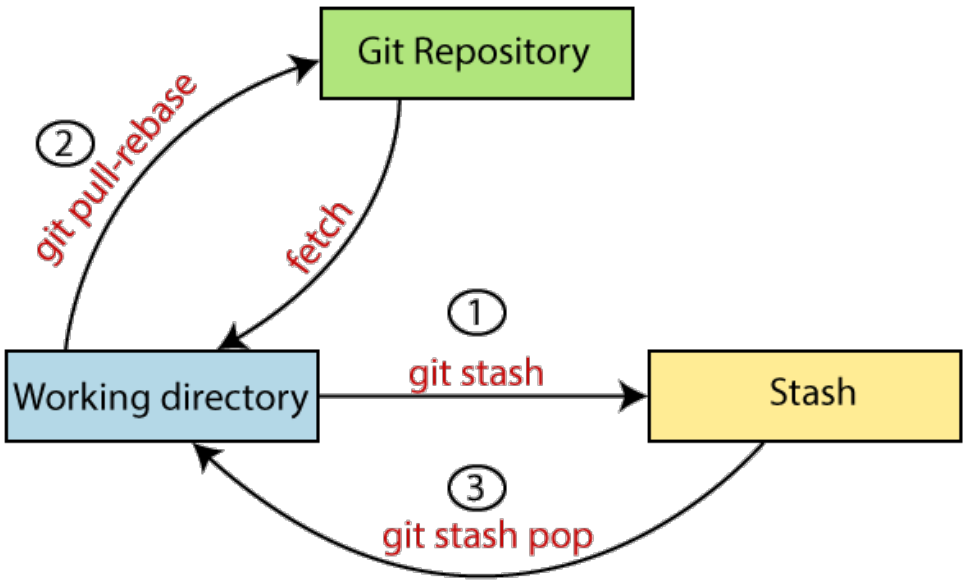

Figura 2.5 Cenário de *stash-apply* [24]

CAPÍTULO 3

Metodologia

O objetivo principal desta pesquisa é continuar trazendo visibilidade sobre a análise de repositórios privados e qual relação do uso de comandos que ofuscam integração de código com a ocorrência de conflitos. Assim como foi feito anteriormente [15, 16], tentar entender a relação entre a ocorrência e os comandos que o geram e identificar se estudos anteriores de integração de código estão focando apenas em uma parte dos casos de conflitos que ocorrem em projetos reais. Devido a quantidade de comandos que ofuscam a integração de código, foi necessário utilizar uma ferramenta feita por Marcela Cunha [15] que analisa cenários de integração local. A ideia vai além de comparar os estudos que focam apenas nos conflitos de repositórios públicos [10, 11, 12], mas também comparar aos resultados de estudos feitos focando nos repositórios privados [15, 16], e principalmente eles. Responderemos as seguintes perguntas:

- **RQ1:** Com que frequência são utilizados comandos que escondem integração de código?
- **RQ2:** Quais são os motivos da adoção ou não de tais comandos?
- **RQ3:** Quais os impactos dessas decisões na ocorrência de conflitos?

As perguntas RQ1 e RQ2 também foram exploradas por Marcela Cunha na sua análise em [16]. Com intuito de responder a RQ1, foi feito uma análise quantitativa reunindo os logs locais de desenvolvedores e aplicando a ferramenta de análise [16] para quantificar a frequência do uso de vários comandos do Git, e seus cenários de integração bem e mal sucedidos. Foi feito um estudo qualitativo cujo objetivo é obter as respostas para RQ2 e junto a análise incial responder a RQ3. Nesse estudo qualitativo foram realizadas uma série de estrevistas semi-estruturadas com o total de 8 desenvolvedores que participaram também na análise de logs. O foco era entender mais a fundo os casos que geraram mais conflitos e que usaram mais comandos que ofuscam integração de código.

3.1 Preparação do estudo

Amostra. Com o intuito de responder às perguntas da pesquisa usamos uma amostra de 35 arquivos de log(reflogs ou logs de referência) de 16 projetos diferentes pertencentes a duas organizações, no total foram coletados logs de 17 desenvolvedores distintos. O autor contactou diretamente líderes técnicos de duas empresas diferentes que ele trabalhou previamente, uma em

2020 e outra até metade de 2022, a fim de coletar os logs diretamente com os desenvolvedores. Por fim, ambas empresas de software concordaram em participar, auxiliando a comunicação e fazendo uma ponte direta entre o autor e os colaboradores.

Script para análise de logs. Como citado anteriormente, foi usado o mesmo script utilizado em outros estudos de análise de repositórios privados [15, 16] que identificam e calculam a frequência dos comandos que serão analisados nesta pesquisa, com o objetivo de ter uma análise automática de toda a base de dados dos 35 arquivos de logs. O script cria tabelas com as instâncias dos comandos e métricas associadas. Um desenvolvedor pode ter contribuído em vários projetos, portanto, o script compila todos os logs pertencentes a cada colaborador em específico, assim como visto em outros estudos [16]. Porém, graças a natureza da *amostra*, podemos além de analisar os logs com foco apenas nos colaboradores, podemos também agrega-los por projetos, agrupando pelos desenvolvedores que trabalharam juntos, tornando assim, mais fácil a visualização de comportamentos diferentes entre projetos dentro de uma mesma empresa, que podem ou não ter colaboradores em comum entre si.

Entrevistas. A fim de complementar a análise de logs, foi feita uma série de entrevistas semi-estruturadas com 8 colaboradores dos 17 que participaram da primeira etapa do estudo, 4 de uma empresa e 4 da outra. Por questões de disponibilidade tanto do autor, quanto dos colaboradores, as entrevistas aconteceram no decorrer de uma semana completa. Uma das empresas entrevistadas é uma empresa de inovação, que não possui um produto próprio e trabalha com ideação e desenvolvimento de produtos para clientes, assim a maioria de seus projetos tem prazos curtos, 14 dos 16 repositórios coletados pertencem a ela. A outra empresa é uma startup que possui um produto principal, e todos os times precisam trabalhar em conjunto para prover novas funcionalidades além da manutenção de código, assim o foco a longo prazo é essencial tanto para os projetos quanto para os colaboradores.

Apesar de ainda ser uma base de dados pequena, ainda é válida dado que o nosso objetivo é entender melhor os resultados da análise dos logs. De forma similar a estudos anteriores [15, 16] as entrevistas se iniciam com perguntas cujo intuito é determinar se o desenvolvedor conhece e usa cada um dos comandos analisados na pesquisa, e o motivo ou não de sua aderência. Em seguida são feitas perguntas sobre de que maneira eles integram o código com o repositório remoto, como e com que frequência eles lidam com conflitos. É de interesse também entender se existe alguma definição de quais diretrizes devem ser seguidas em relação ao uso do git no projeto.

Diferente dos estudos anteriores [15, 16], são feitas perguntas sobre diretrizes do uso do Git/Github, uso de Pull Requests, processo de deploy, uso de testes unitários e de integração, a fim de entender como isso pode interferir na ocorrência de conflitos. Todas as entrevistas foram gravadas e feitas de forma individual e duraram em média 15 minutos. A seguir, uma amostra de algumas das perguntas feitas nas entrevistas. Todo o conjunto de perguntas pode ser encontrado no apêndice A:

- Tem experiência ou conhece o comando de *rebase*?
- Existe alguma regra sobre quais comandos de integração devem ser usados no projeto?
- Com que frequência acontecem conflitos ao integrar código?
- Após um PR de uma feature ser aberto, quanto tempo leva para ser revisado, aprovado e finalmente mergido?
- Qual seu fluxo de trabalho?

3.2 Análise dos logs

Para responder as perguntas estabelecidas se fez necessário automatizar a identificação do uso desses comandos dos arquivos locais de log, para isso vamos usar o mesmo script construido por Marcela [15], cujo etapas são: *Identificação de Comandos*, *Identificação de Integração*, *Agregação de resultados*. Onde apenas na última etapa vamos acrescentar o agrupamento por projetos para facilitar a visualiação e ajudar na condução das entrevistas.

Identificação de Comandos. A abordagem é observar o histórico local do projeto conforme registrado no arquivo reflog do repositório local. Este arquivo contém detalhes de todas as atividades git do desenvolvedor executadas localmente em seu repositório. O Git salva e atualiza automaticamente esse arquivo conforme o usuário emite comandos para o Git. Ele é acessível através do diretório local do projeto, que fica no caminho ".git/logs/HEAD", que é uma lista de todos os comandos que foram referidos no repositório. A configuração padrão para expiração do reflog é de 90 dias, mas pode ser configurada. Porém para esse estudo, não foi questionado aos colaboradores se eles haviam configurado essa janela de tempo, porém a grande maioria deles não sabia da existência do arquivo de log, ou seja, é pouco provável que algum arquivo não possua a configuração padrão.

Cada comando feito pelo desenvolvedor num repositório local é gerado automaticamente uma chave única, SHA-1 atrelada a ele [21], mantendo assim o histórico local de modificações. HEAD é um ponteiro orquestrado pelo Git que informa o último comando executado, qual branch atual e é atualizado continuamente de forma automática à medida que mais comandos são executados [16]. O arquivo em si, reflog, consiste em uma sequência de linhas, cada uma representa uma operação realizada, como mostrado na Figura 3.1 cada linha possui: (1) chave única do comando anterior ao HEAD, (2) a chave única do comando atual, HEAD, (3) Nome do autor do comando, (4) Email do autor do comando, (5) Comando utilizado, (6) Mensagem atrelada ao comando.

Observe que na Figura 3.1, primeiro o desenvolvedor clona o repositório localmente, fez um commit, muda da branch *master* para *test* e faz um novo commit. Contudo nem todos os comandos de integração são representados de forma tão clara, e para construção do script foi

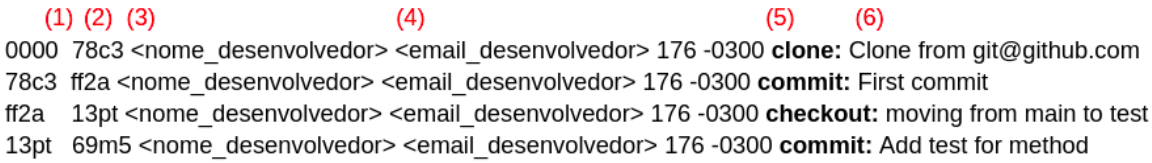

Figura 3.1 Reflog

preciso ser feito uma análise manual de vários arquivos de log, a fim de entender como cada cenário de integração é representado e como ele pode ser detectado de forma automática pela ferramenta [16].

Com ela é possível identificar: (a) *merge*, (b) *rebase*, (c) *cherry-pick*, (d) *squash*. O comando de *stash-apply* não deixa rastros no reflog, pois apesar do git registrar quando um *stash* é feito, o momento do *apply* não deixa rastros no arquivo de reflog, por conta disso esse comando não vai ser analisado no estudo. O *merge* é mostrado na figura 3.2, e nas figuras 3.3, 3.4 vemos ambos casos de uso do *rebase*.

Figura 3.2 Merge

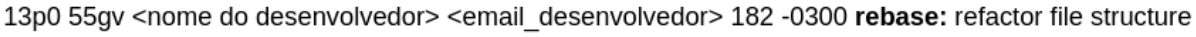

Figura 3.3 Rebase

O comando de *cherry-pick* é representado na figura 3.5, já o *squash*, não é possível identificar em um comando apenas, mas é possível perceber seu uso através de um *rebase* interativo, onde o desenvolvedor pode escolher a ação de *squash* numa lista de commits, como mostrado na figura 3.6.

Identificação de Integração. Com os comandos do Git já identificados podemos seguir para os cenários de integração de código. Similar ao que já foi feito, nem todos os cenários de integração são interessantes para o estudo, alguns comandos podem ser usados porém não necessariamente ocorrem integração de código [15, 16], esses não iremos utilizar no estudo. Por exemplo, ao tentar integrar um commit com outro commit da mesma branch que faz parte do mesmo histórico linear, o que o Git faz é apenas mover o HEAD, o histórico se mantém o mesmo e não há de fato integração de código, chamado de *fast-forward* [21].

A ação de integrar código pode não ser bem sucedida. Caso uma integração ocorra sem gerar conflitos e não seja uma ação de *fast-forward*, nós contabilizamos como um cenário de sucesso. Quando conflitos de integração ocorrem, o usuário tem três opções: *Resolver* os conflitos gerados e seguir com a integração; *Pular* o commit que causou o conflito, ignorando-o;

```
13p0 55gv <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 rebase -i (start): checkout 867c2
55gv 3914 <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 rebase -i (reword): updating HEAD
3914 928d <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 rebase -i (pick): group test
928d 928d <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 rebase -i (finish): returning to HEAD
```

Figura 3.4 Rebase Interativo

```
13p0 55gv <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 cherry-pick: add Method
```

Figura 3.5 Cherry-Pick

Abortar a operação, deixando para trás tudo o que já foi feito. Dito isso, independente do que o desenvolvedor optar por fazer, todas as três opções são contabilizadas como falhas. Assim analisando o arquivo completo, conseguimos além da frequência de uso de cada comando de integração, também a quantidade de falhas e sucessos ao realizar a operação, processo igual ao feito anteriormente [16].

Essa contagem não é precisa para nenhum dos lados. Existem casos que não é possível identificar a diferença entre uma integração bem sucedida e uma integração de *fast-forward*, devido às características iguais para ambos cenários, ou seja, a contagem de sucessos pode estar superestimada. A lógica se aplica também para falhas, quando uma integração gera conflito, os cenários de *Pular* e *Abortar* não deixam registros no arquivo de logs, assim a contagem de falhas pode estar subestimada. Isso já foi citado no estudo anterior [16] e está representado na tabela 3.1.

Comando Git	Sucesso	Falha
Merge	=	↑
Cherry-Pick	↓	↑
Squash	↓	↑
Rebase		
- Normal	=	↑
- Interativo	=	=
- Pull - rebase	=	↑

Tabela 3.1 Os cenários de integração identificados dos comandos Git. As setas indicam se o número está subestimado (↑) ou superestimado (↓), o que significa que os números devem ser maiores ou menores respectivamente [15]

Agregação de resultados. Diferente do que foi feito nos estudos anteriores [16], é necessário que exista uma agregação de dados também voltada a projetos. Essa visualização pode ajudar a identificar comportamentos diferentes dentro dentro da mesma organização, e servir como apoio para a etapa da análise qualitativa ao questionar sobre as diretrizes adotadas em cada projeto.

13p0 55gv <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 **rebase -i (start):** checkout 867c2
55gv 3914 <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 **rebase -i (reword):** updating HEAD
3914 928d <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 **rebase -i (pick):** group test
928d 13a8 <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 **rebase -i (squash):** refact
13a8 13a8 <nome do desenvolvedor> <email_desenvolvedor> 182 -0300 **rebase -i (finish):** returning to HEAD

Figura 3.6 Squash

CAPÍTULO 4

Resultados

Foram analisados 35 arquivos de logs, em comparação aos 95 do estudo anterior [16], pertencentes a 17 desenvolvedores diferentes, de um total de 16 repositórios, como aponta a tabela 4.1. Foram coletados 8540 comandos de Git, valor 4 vezes menor que no estudo anterior [16], dados sobre organizações e quais projetos cada log pertencem não foram registrados.

Coleta de logs	Estudo Anterior	Estudo Atual
Total	95	35
Arquivos que possuem integração de código	77	26
Projetos	-	16
Organizações	-	2
Desenvolvedores	61	17

Tabela 4.1 Comparação da coleta coleta de logs entre o estudo atual e o estudo anterior [16]

Coleta de logs	Total	Org A	Org B
Total	35	29	6
Arquivos que possuem integração de código	26	22	4
Projetos	16	14	2
Desenvolvedores	17	12	5

Tabela 4.2 Distribuição de logs por organização

Além da análise de logs, foram feitas entrevistas semi-estruturadas com 8 desenvolvedores que participaram da primeira etapa do estudo, a fim de construir um estudo qualitativo. Como mostrado na Tabela 4.3, foram 4 entrevistados de cada organização, onde para organização A, foram entrevistados 4 colaboradores de 4 projetos distintos, e para organização B, tivemos 3 de um projeto e 1 de outro.

Vale destacar algumas características sobre as organizações. A organização A é considerada uma fábrica de software, uma empresa de inovação focada em ideação e desenvolvimento, ela não possui um produto principal próprio, clientes a contratam para realizar o processo de ideação e implementação de soluções de inovação. Isso significa que temos normalmente projetos com prazo curto de desenvolvimento, e muitas vezes desenvolvedores que foram contratados apenas durante o tempo de implementação do projeto.

Já na organização B, temos uma startup que tem um produto específico, todos os times trabalham focado nele, assim temos um foco a longo prazo não apenas em relação ao projeto, mas também em relação aos colaboradores, uma empresa nova com expectativa de crescimento

é atrativa para desenvolvedores fazerem carreira. O projeto 1 é o *front-end*, projeto 2 é o *back-end*.

Projetos	Organização	Quantidade de entevistados
Projeto 1	B	3
Projeto 2	B	1
Projeto 3	A	1
Projeto 4	A	1
Projeto 5	A	1
Projeto 6	A	1

Tabela 4.3 Distribuição de desenvolvedores entrevistados no estudo qualitativo por projeto

4.1 RQ1: Qual frequência de comandos que ofuscam integração de código?

Comandos Git	Estudo Anterior	Estudo Atual
Total	34504	8540
Comandos de Integração	4131(12%)	241(2.8%)
Comandos visíveis	548(1.6%)	160(1.87%)
Comandos invisíveis	3583(10.4%)	81(0.94%)

Tabela 4.4 Comparação da quantidade de comandos Git entre o estudo atual e o estudo anterior [16]

Comandos Git	Estudo Anterior	Estudo Atual
Total	4131	241
Comandos visíveis	548(13.3%)	160(66,4%)
Comandos invisíveis	3583(86.7%)	81(33,6%)

Tabela 4.5 Comparação da quantidade de comandos de integração entre o estudo atual e o estudo anterior [16]

A ferramenta de análise identifica as ocorrências de todos os comandos de integração de código. Para responder essa pergunta, precisamos calcular a quantidade de *rebase*, *squash* e *cherry-pick*. Essa frequência é a soma dos casos de sucesso e falha para cada um dos comandos de integração estudados. Contudo, como citado na Seção 3 (Metodologia), cenários de conflitos que foram *abortados*, *pulados* ou cenários de *fast-forward* não são detectados, assim temos uma estimativa da frequência em comparação ao valor real.

Dos 8540 comandos de Git, apenas 241 foram cenários de integração de código, um valor de aproximadamente 2.8% do total, como mostra tabela 4.4. Levando em conta apenas os cenários de integração, cenários que ofuscam integração de código totalizam 33.6%, como apresentado na tabela 4.5, valor bem diferente dos 86.7% apresentados no estudo anterior [16],

essa diferença tão alta pode estar relacionada ao fato que 14 dos 16 repositórios, ou 82.9% dos logs coletados fazem parte da mesma organização como mostrado na tabela 4.2, e graças ao estudo qualitativo descobrimos que o padrão é o uso do comando de *merge*, que não ofusca integração de código.

Dando foco apenas nos cenários que ofuscam a integração, temos o *rebase* como o comando mais utilizado com 79%, valor um pouco maior que os 65.7% detectados anteriormente [16], seguido do *cherry-pick* com 21%, que antes foi de 30.5% e por fim o *squash*, que anteriormente ocorreu em 3.85% dos casos e no estudo atual não foi encontrado nenhuma ocorrência [16].

Vimos também que a taxa de falhas no uso do comando *merge* foi de 35.4%, valor próximo aos 37.2% encontrados anteriormente [16], ja em relação aos comandos que ofuscam integração de código, tivemos no estudo anterior [16] um valor de 11.3%, já nesse foi encontrado 21.8% de falhas.

Resultado 1: De acordo com a amostra analisada, comandos que ofuscam integração de código são menos usados, com 33.6% valor substancialmente menor comparado aos 86.7% encontrados anteriormente [16]. Essa diferença pode ser explicada pela característica da amostra analisada, que tem 82.9% dos logs pertencentes a uma única organização que define como padrão o uso de *merge* como comando de integração, que é um comando que não ofusca a integração. Tivemos também uma taxa similar de falhas no uso do comando *git merge*, porém um valor bem maior quando focado nos comandos que ofuscam integração de código, 21.8% contra 11.3% [16], uma diferença grande, porém é preciso ser levado em conta que a frequência desses comandos em ambos estudos é bem diferentes. Ainda sim, mesmo em proporção menor, ainda temos vários cenários que ofuscam integração de código que estão sendo ignorados por estudos que focam apenas em repositórios públicos e remotos.

4.2 RQ2: Quais são os motivos da adoção ou não de tais comandos?

O análise de logs serviu como base para guiar o estudo qualitativo e, para entender os motivos do uso de cada um dos comandos, foi necessário entrevistar 8 desenvolvedores que participaram da primeira etapa da pesquisa, sendo metade pertencente a cada organização estudada. Como mostrado na tabela 4.2, a distribuição de logs por organização não está equilibrada.

Apesar de que todos os entrevistados tinham conhecimentos de todos os comandos que estão sendo estudados nessa pesquisa, o grande uso de merge é devido ao fato de que, na org A, detentora de 82.9% dos logs da amostra, foi reportado através das entrevistas, que os colaboradores são aconselhados a usar como método padrão de integração de código o comando *git merge*. Um dos entrevistados atua como gerente de projetos e, apesar de atuar pouco como desenvolvedor, é o colaborador com mais experiência dentre todos entrevistados. Segundo ele, a média de tempo de experiência dos desenvolvedores na empresa é de 1-2 anos, onde ele está próximo dos 4 anos.

Segundo ele, o uso do *merge* foi escolhido pois era o mais simples, que todos conheciam ao chegar na empresa e, como muitas vezes eles lidam com desenvolvedores em incio de carreira, é o menos suscetível a erros no processo de integração, independente de acarretar ou não conflitos, a familiaridade prévia da equipe com o comando fez com que o *git merge* se tornasse padrão. Ele foi o único da empresa que fez uso do *git rebase*, e explicou: *"Eu uso rebase quando preciso integrar código mas ainda não subi um PR, não gosto de subir um PR com commits de merge, só faço isso quando sei que já estão revisando o PR. Daí não quero dar push force."*

Na organização B, o projeto 1 tentou o uso do *git rebase* como padrão, mas desistiram como relata um dos entrevistados: *"A gente tentou, mas como os PRs tem muitos commits demorava muito pra conseguir fazer, sempre gerava muitos conflitos, sempre algo dava errado tinha que voltar, desistimos"*. Relato que é possível validar nos logs, onde, para esse projeto, foi encontrado uma taxa de 70% de *rebase* falhos, sendo metade deles, abortados. Ou seja, existiu uma tentativa do uso de um comando que ofusca integração de código, mas nesse caso a grande quantidade de commits atrapalhou de forma direta na sua utilização.

Já no projeto 2, ainda na organização B, tivemos a maior proporção de comandos que ofuscam integração de código, chegando a 67.8%, valor mais próximo à proporção encontrada no estudo feito por Marcela Cunha[16] que foi de 86.7%. Desses comandos, tivemos 57.5% de *rebase* e 42.5% de *cherry-pick*, sendo o único projeto que usou o comando em toda a amostra. Esse uso vem da necessidade do controle de quais mudanças vão ser entregue no ambiente de produção, e essa seleção específica de mudanças é feita através do *cherry-pick*. Eles prezam por uma linha do tempo linear, para facilitar esse controle, e explicou o uso: *"Começamos a usar pois o front-end sempre acabava atrasando as features e nem sempre por questões de agilidade a gente conseguia fazer mudanças retrocompatíveis. Por conta disso, dado que temos uma linha do tempo linear e limpa usando rebase e squash, a gente usa cherry-pick para escolher quais mudanças vamos mover para produção"*.

Resultado 2: Temos duas organizações com características bem distintas. Por um lado, temos o uso massivo do comando de *git merge* que além da familiaridade com o comando, há uma maior facilidade para lidar com conflitos caso aconteçam, se comparado com outros comando de integração, como o *rebase* interativo que exige uma possível resolução de conflitos a cada iteração do comando. E do outro temos o uso em massa devido uma definição explícita, quase como regra, definida pelos gerentes de projetos. Ou seja, o uso pela familiaridade e maior facilidade de resolução de conflitos. Contudo, tivemos um caso evidente onde ter um histórico limpo e linear é necessário e de extrema importância para o desenvolvimento contínuo e manutenção de um produto real. Possuir o controle de quais mudanças vão para o ambiente de produção, através do *cherry-pick* se mostrou essencial para o projeto 2.

4.3 RQ3: Quais os impactos dessas decisões na ocorrência de conflitos?

Para entender os impactos, foi necessário conhecer mais sobre as organizações, padrões de projetos adotados, e como funciona o fluxo de trabalho de cada um dos entrevistados.

Se levarmos em conta estudos anteriores que apontam que entre 10% a 20% de todos os comandos de integração falham [3, 4, 5, 6, 7], a organização A se manteve dentro da média. O que ficou explícito com as entrevistas é que, ao iniciar o projeto, a taxa de conflitos aumenta drasticamente, chegando a quase 50%, valor que também já foi encontrado em outros estudos [7]. Essa frequência tende a diminuir e voltar à média no decorrer do projeto, onde a estrutura já está bem definida e não é necessário a mudança frequente de configurações globais.

Já na organização B tivemos casos bem diferentes, no projeto de back-end (2) foi relatado que já existiam poucos conflitos. O time era pequeno, apenas 2 pessoas, ambos possuíam mais de 3 anos de experiência, tanto de Git quando de mercado de trabalho, e dentro desse projeto sempre trabalhavam juntos, numa espécie de pair programming. Mesmo quando trabalhando em tarefas diferentes, um sempre sabia do contexto do outro. O projeto foi o único do estudo que possuía testes unitários e de integração automatizados e segundo o desenvolvedor isso reduzia muito o tempo de revisão, conseguindo realizar o deploy das mudanças em produção de maneira mais rápida.

Existia um esforço ativo por parte do time para manter PRs pequenos, com poucos commits e, para isso, eles costumavam usar a estratégia de *Stacked PRs*, como mostra na figura 4.1. A ideia é de abrir um PR de uma feature para ser revisado e suas mudanças integradas numa *branch* que pertence a outra feature que ainda está em desenvolvimento. O ato de dividir uma feature grande em várias pequenas e subir em etapas acarreta em PRs pequenos, mais fáceis de revisar. Quanto menos tempo um PR fica em aberto, menos chances de gerar conflitos, além de ajudar bastante no tempo de revisão de PRs, melhorou bastante a agilidade do time e a capacidade de entregar novas tarefas. Porém, acontecia de, por engano, a ordem que os PRs deveriam ser integrados ser confundida, e como o projeto faz uso do *squash and merge* isso fazia com que o histórico de commits fosse reescrito, quebrando e gerando conflitos em todas as *branch* que foram construídas a partir da *branch* atual, que deveriam ter sido integradas primeiro. Essa foi a única causa de conflitos relatadas pelo entrevistado que disse que a média de conflitos no projeto chegava no máximo em 5%.

Ainda na organização B, mas agora focando no projeto de *front-end* (1), temos uma média de experiência entre 1-3 anos, parecido com a org A, mas foi relatado nas entrevistas que a quantidade de conflitos que acontecem é muito alta: *"Eu não lembro a última vez que fiz um merge que não gerou conflitos, inclusive quando a gente vai mergir um PR e não tem que resolver conflito a gente acha que tem algo errado, e vai investigar."*, um dos três 3 desenvolvedores que trabalham no projeto atua hoje como gerente de projetos, e só escreve código para resolver algum bug mais importante. Já os outros dois que lidam com todas as cargas de features, ambos não tiveram nenhum comando de integração que não gerou conflitos

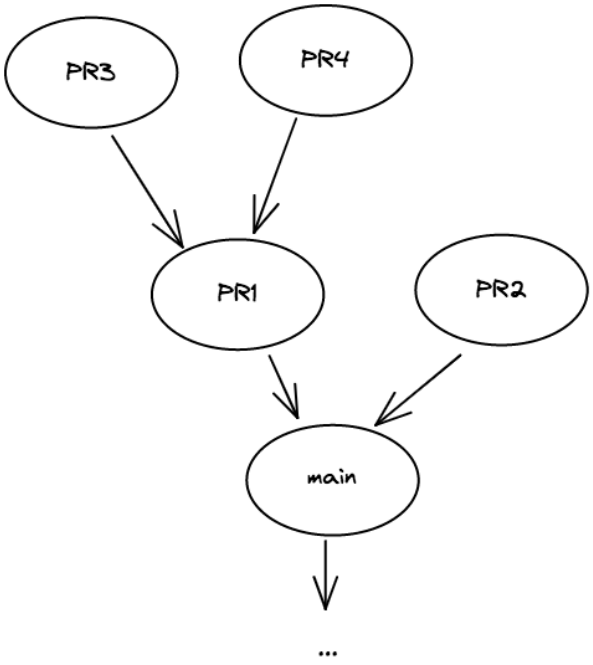

Figura 4.1 Stacked PRs [24]

na análise dos logs, comportamento confirmado por eles nas entrevistas, segundo eles a taxa de conflitos era de 98%.

Algumas características do *front-end* (1) valem ser destacadas, PRs de features demoram cerca de 2 semanas para serem revisados e diferente do *back-end* (2) não existem testes unitários e de integração automatizados, é trabalho do revisor acessar o ambiente de teste e tentar quebrar na mão o front-end com a mudança, além de revisar o código em si. Features muito grandes, que geram PRs longos, complexos, com muitos commits, e que demoram para serem revisados definitivamente influenciam na ocorrência de conflitos.

Na entrevista foi constatado que a solução pra isso está sendo investir mais tempo no planejamento, para ter tarefas menores e mais bem definidas e foi dado início o uso de Stacked PRs, um padrão que já é usado no time de back-end e que começou a ser usado no time de front-end e que segundo eles tem ajudado a manter os PRs menores, e até então não sofreram com o problema ocorrido no back-end de confundir a ordem dos PRs na hora do *merge*.

	Projeto em estado inicial	PRs abertos por muito tempo	Falta de planejamento das tasks	Uso de Stacked PRs	Possuir mais de 2 pessoas no projeto	Grau de senioridade baixo
Influencia	X	X	X	X	X	X
Não Influencia				X		

Tabela 4.6 Consolidação de pontos coletados que influenciam ou não na ocorrencia de conflitos.

Resultado 3: Analisamos perspectivas e maneiras diferentes de como lidar com integração de código, até mesmo dentro de uma mesma organização. Na org B, ambos times acabam se adaptando e testando novas formas de integrar código, e a experiência de um é compartilhada entre eles. Tivemos o uso do *cherry-pick* apenas pelo projeto de back-end, enquanto o front-end ainda usava *git merge*, porém o uso de Stacked PRs que se iniciou no back-end, está sendo integrada no fluxo do front-end.

Como consolidado na tabela 4.6, vemos que a ocorrência de conflitos pode ser influenciada devido ao fato do projeto estar ainda no início de seu ciclo de desenvolvimento, requerendo muitas configurações e mudanças na estrutura global. A falta de planejamento nas tarefas, acarreta normalmente em PRs grandes, que demoram na revisão, aumentando a ocorrência de conflitos. O uso de Stacked PRs se provou eficiente para diminuição do tamanho dos PRs, mas exige cuidado no uso podendo acarretar em conflitos caso seja usado de maneira indevida. A quantidade de pessoas trabalhando juntas está diretamente relacionada às chances de conflitos acontecerem, junto com o baixo grau de *senioridade*. Times mais **experientes** investem mais tempo em documentação e planejamento de tarefas.

CAPÍTULO 5

Conclusão

Nessa pesquisa foram realizados estudos quantitativos e qualitativos sobre a frequência do uso de comandos de integração em repositórios locais e de que maneira isso se relaciona com a ocorrência de conflitos. Diferente da maioria de estudos feitos anteriormente na área, esse tem como foco olhar apenas para os logs locais dos desenvolvedores.

Foram coletados 35 arquivos de log, de 17 desenvolvedores distintos, que foram analisados através de uma ferramenta desenvolvida em outros estudos na área que tem como foco também os logs locais, mas foi encontrado um resultado diferente, devido a características distintas da amostra, apontando um uso muito menor de comandos que ofuscam integração de código do que encontrado anteriormente [16].

Além da análise de logs, foram realizadas 8 entrevistas semi-estruturadas, a fim de entender mais a fundo a relação entre o uso dos comandos de integração, fluxo de trabalho e a ocorrência de conflitos. Encontramos uma proporção de uso de comandos de integração próxima a estudos anteriores [16] quando focado apenas nos desenvolvedores com mais de 3 anos de experiência de mercado e uso do Git. A relação entre comandos de integração e o surgimento de conflitos não demonstrou relevância direta, porém algumas características de fluxos de trabalho e diretrizes de projeto sim tiveram grande impacto.

Foram destacados de acordo com a entrevistas que junto ao baixo grau de experiência do time, o maior causador de conflitos é devido a quanto tempo um PR fica em processo de revisão. De forma indireta, a falta de planejamento e definição de tasks se mostrou como maior causador dessa demora, PRs longos, complexos podem demorar até 2 semanas para serem revisados, acarretando em grande quantidade de conflitos gerados. Na análise de logs para esse projeto em específico nenhum comando de integração foi bem sucedido, comportamento que foi confirmado nas entevistas pelos desenvolvedores: *"Eu não lembro a última vez que fiz um merge que não gerou conflitos"*. De forma análoga, projetos que possuem PRs pequenos, organizados, com testes automatizados que facilitam o trabalho do revisor foram encontrados baixa ocorrência de conflitos, chegando ao máximo em 5%.

Com isso, além de trazer mais à tona, estudos e análises que focam em repositórios locais, podemos perceber como o uso de comandos de integração, junto ao fluxo de trabalho e diretrizes dos projetos influenciam a ocorrência de conflitos.

CAPÍTULO 6

Trabalhos Futuros

Em ambientes de desenvolvimento colaborativo de software, desenvolvedores trabalham em paralelo de forma independente, podendo ocasionar em conflitos na integração de código com o repositório principal. Conflitos como esses são frequentes e já foram apontados em estudos que podem impactar negativamente a produtividade, qualidade e corretude do código. Neste trabalho o foco foi o estudo de comandos de integração de código que acontecem nos repositórios locais dos colaboradores, e como a relação desses comandos junto a características dos projetos podem acarretar na ocorrência de conflitos.

Devido a diferença nas características da amostra coletada em comparação com estudos anteriores, os resultados sobre a frequência de comandos foram diferentes, propomos um novo estudo agora com foco em desenvolvedores mais experientes, acima dos 3 anos de experiência, tanto com o uso de Git, tanto no mercado de trabalho, pois foi nesse subconjunto pequeno da nossa amostra que conseguimos achar resultados próximos aos encontrados antes.

Como dentro da nossa amostra tivemos apenas um projeto que possuía testes automatizados, tanto de integração quanto unitário, é de interesse estudar mais projetos com essas características para compreender o real impacto a existência de testes com a ocorrência de conflitos, seja de maneira direta ou indireta.

APÊNDICE A

Lista de Perguntas das entrevistas

- Quanto tempo de experiência no mercado de trabalho?
- Quanto tempo como desenvolvedor no projeto?
- Quanto tempo de experiência usando git?
- Usa alguma ferramenta para auxiliar o uso do Git/GitHub?
  - Qual(is)?
  - O que ela te ajuda?
- Participou da definição do que seriam as diretrizes adotadas pelo projeto sobre Git/GitHub, ou seja, estrutura de commits, uso de comandos de integração, estrutura de PRs? Tem liberdade pra mudar?
- Quantas pessoas trabalham no projeto hoje?
- Tem experiência com o comando de *rebase*? *squash*? *cherry-pick*?
- Levando em conta a quantidade de vezes que você realiza integração de código, com que frequência acontecem conflitos? 50%, 20%, todas as vezes?
- Você costuma resolver conflitos? Se sim, sozinho ou pede ajuda a alguém do time?
- Os PRs do projetos você considera de qual tamanho?
  - pequeno: consigo trocar de contexto para revisar sem muitos problemas;
  - médio: preciso tirar um tempo pra revisar com atenção;
  - grande: nunca consigo revisar tudo de uma vez, sempre aparece outra demanda, ou acabo me distraindo pelo tamanho.
- Baseado na definição de tamanho acima, quanto tempo leva para um PR de tamanho médio, de uma nova feature, ser revisado, aprovado, e finalmente mergido? Ou seja, após abrir o PR, quanto tempo até mergi-lo?
- Ja lidou com Stacked PRs? Um PR que vai ser mergido em outra branch de feature/fix, que não é a main, criando essa interdependência.
- Qual seu fluxo de trabalho? Como você gostaria que fosse? Porque?

Referências Bibliográficas

[1] Stack-Overflow, "Version control survey," *https://survey.stackoverflow.co/2022/version-control-systems*.

[2] D. Perry, H. Siy, and L. Votta, "Parallel changes in large scale software development: an observational case study,"

[3] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," May 2013.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1358–1375, Oct. 2013.

[5] W. Mahmood, M. Chagama, T. Berger, and R. Hebig, "Causes of merge conflicts: A case study of elasticsearch," Feb. 2020.

[6] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," Sept. 2017.

[7] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," 2011.

[8] A. van der Hoek and A. Sarma, "Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts," 2008.

[9] M. L. Guimaraes and A. R. Silva, "Improving early detection of software merge conflicts," June 2012.

[10] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1–27, Oct. 2017.

[11] Y. Nishimura and K. Maruyama, "Supporting merge conflict resolution by using fine-grained code change history," vol. 1, pp. 661–664, 2016.

[12] J. Clementino, P. Borba, and G. Cavalcanti, "Textual merge based on language-specific syntactic separators," 2021.

[13] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge," 2011.

[14] P. Accioly, P. Borba, and G. Cavalcanti, “Understanding semi-structured merge conflict characteristics in open-source java projects (journal-first abstract),” Sept. 2018.

[15] M. Cunha, “Entendendo o uso do git em equipes de desenvolvimento de software,” pp. 1–29, Dec. 2018.

[16] M. Cunha, P. Accioly, and P. Borba, “The private life of merge conflicts,” Oct. 2022.

[17] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, “The promises and perils of mining git,” May 2009.

[18] E. Shihab, C. Bird, and T. Zimmermann, “The effect of branching strategies on software quality,” pp. 301–310, 2012.

[19] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, “Awareness and merge conflicts in distributed software development,” pp. 26–35, 2014.

[20] S. Apel, O. Leßenich, and C. Lengauer, “Structured merge with auto-tuning: balancing precision and performance,” pp. 120–129, 2012.

[21] S. CHACON and B. STRAUB, “Pro git,” vol. 2nd edition, 2014.

[22] Atlassian, “Git merge,” *https://www.atlassian.com/git/tutorials/using-branches/git-merge*.

[23] Atlassian, “Git rebase,” *shorturl.at/dtx23*.

[24] Atlassian, “Git stash apply,” *https://static.javatpoint.com/tutorial/git/images/git-stash.png*.

[25] B. Congdon, “Stacked prs,” *shorturl.at/ENQV0*.