ARTHUR ROBERTO MARCONDES

UMA ABORDAGEM PARA ATUALIZAÇÃO DE FORKS FRENTE AO PROJETO ORIGINAL

LAVRAS – MG

2021

ARTHUR ROBERTO MARCONDES

UMA ABORDAGEM PARA ATUALIZAÇÃO DE FORKS FRENTE AO PROJETO ORIGINAL

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre em Ciência da Computação.

Prof. DSc. Ricardo Terra Nunes Bueno Villela

Orientador

LAVRAS – MG

2021

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Marcondes, Arthur Roberto
Uma abordagem para atualização de forks frente ao projeto original / Arthur Roberto Marcondes. – Lavras : UFLA, 2021.
103 p. : il.

Dissertação (mestrado acadêmico)–Universidade Federal de Lavras, 2021.
Orientador: Prof. DSc. Ricardo Terra Nunes Bueno Villela.
Bibliografia.

ARTHUR ROBERTO MARCONDES

UMA ABORDAGEM PARA ATUALIZAÇÃO DE FORKS FRENTE AO PROJETO ORIGINAL
AN APPROACH FOR UPDATING FORKS AGAINST THE ORIGINAL PROJECT

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre em Ciência da Computação.

APROVADA em 22 de Janeiro de 2021.

Prof. DSc. ANDRÉ GUSTAVO DUARTE DE ALMEIDA	IFRN
Prof. DSc. HEITOR AUGUSTUS XAVIER COSTA	UFLA
Prof. DSc. MAURICIO RONNY DE ALMEIDA SOUZA	UFLA

Prof. DSc. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2021

Dedico esta dissertação à minha família, em especial à minha esposa Josiane e aos meus filhos Matheus e Jonathas, razões do meu viver.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela saúde, força e sabedoria a mim concedidas. Sem isso nada seria possível.

Agradeço à minha esposa Josiane pelo apoio e incentivo, cruciais na decisão de começar o mestrado e prosseguir com esse até o final.

Agradeço ao professor Terra pela paciência em me orientar, pelo tempo dedicado, pelo comprometimento com os resultados e pela empolgação com o projeto em momentos em que nem mesmo eu acreditava no sucesso. Obrigado!

Agradeço a todos os meus colegas de mestrado pelo apoio, ajuda e companheirismo nas horas difíceis. Foi um desafio ficar longe da família e retomar os estudos depois de 13 anos longe da vida acadêmica.

Agradeço especialmente ao IFSULDEMINAS pela afastamento integral que me foi concedido. Sem esse seria impossível dedicar o tempo e esforço necessários à conclusão do mestrado.

Agradeço por fim à UFLA pela oportunidade. Como diz o ditado popular "O bom filho a casa torna". Foi na UFLA que iniciei minha carreira acadêmica e foi nela que subi mais esse degrau.

A todos meu muito obrigado!

Aqueles que passam por nós, não vão sós, não nos deixam sós. Deixam um pouco de si, levam um pouco de nós.
(Antoine de Saint-Exupéry)

RESUMO

Diversos projetos de software se iniciam a partir de um projeto já existente. Essa prática, no ecossistema de Sistemas de Controle de Versão (*Version Control Systems* – VCS), é denominada *fork*. Por exemplo, o projeto Bootstrap, inicialmente desenvolvido no Twitter, possui em dezembro de 2020 mais de 71 mil *forks*, o que indica que vários projetos se iniciaram a partir do código-fonte do Bootstrap em um certo instante e estão sendo customizados. O problema se dá quando tais projetos customizados querem obter as atualizações do projeto original, i.e., novas *features*, correção de *bugs*, etc. Essa mesclagem do código-fonte do projeto original com o projeto customizado normalmente gera conflitos que necessitam de intervenção humana para resolução. Mais importante, a resolução desses conflitos pode não ser trivial e representar uma tarefa árdua para desenvolvedores. Esta dissertação de mestrado, portanto, propõe uma abordagem para atualização de *forks* frente ao projeto original onde *features* são modularizadas, documentadas, rastreáveis e podem ser reutilizadas. Afirma-se que essa tarefa não pode mais ser realizada de uma forma *ad hoc*. A abordagem permite a atualização desses sistemas de uma forma não invasiva, preservando a independência do projeto customizado e não exigindo intervenções diretas no projeto original. Para tanto, baseia-se na descrição de *features* em alto nível através de uma DSL (*Domain Specific Language* – Linguagem de Domínio Específico). De forma sucinta, ao invés de alterar o corpo de um método `foo` do projeto original, o desenvolvedor o implementa localmente e especifica, por meio de uma das onze instruções da DSL proposta, algo como "substitua o método `foo` pela implementação local". Uma ferramenta que automatiza a abordagem proposta foi desenvolvida para conduzir uma avaliação real em um projeto de software frequentemente atualizado frente ao seu projeto original. Por meio dessa avaliação constatou-se que a abordagem aplica-se a cenários reais e evita conflitos de mesclagem. Avaliou-se também a perspectiva de desenvolvedores desse projeto quanto à abordagem proposta, que se mostrou positiva e contribuiu no planejamento de trabalhos futuros.

Palavras-chave: Evolução de software. Atualização de *fork*. Conflitos de mesclagem.

ABSTRACT

Several software projects start from an existing project. This practice, in the VCS ecosystem, is called fork. For instance, the Bootstrap project, initially developed on Twitter, has in December 2020 more than 68,000 forks, which indicates that several projects started from the Bootstrap source code at a certain moment and are being customized. The problem occurs when customized projects want to obtain updates from the original project, i.e., new features, bug fixes, etc. The merge of the source code between the original and the customized projects usually generates conflicts that need human resolution. More important, the resolution of those conflicts might not be trivial and poses an arduous task for developers. This dissertation, therefore, proposes an approach for updating forks against the original project where features are modularized, documented, traceable, and can be reused. We claim that the such task can no longer be carried out on an ad hoc basis. The approach updates those systems in a non-invasive way, which preserves the independence of the customized project does not requires direct interventions in the original project. For this purpose, it specifies features on a high level using a DSL (Domain Specific Language). In a nutshell, instead of modify the method `foo` from the original project, the developer implements it locally and specifies, using one of the eleven instructions of the proposed DSL, something like "replace the `foo` method with local implementation". We developed a tool that automates the approach and conduced an evaluation on a real-world project that is regularly updated against your original project. Through this evaluation we found that the approach applies to real scenarios and avoids merging conflicts. We also evaluated the perspective of the developers of this project regarding the proposed approach, which proved to be positive and contributed to the planning of future work.

Keywords: Software evolution. Fork update. Merge Conflicts.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ANTLR	*Another Tool for Language Recognition*
AST	*Abstract Sintax Tree*
CVCS	*Centralized Version Control System*
DSL	*Domain Specific Language*
DVCS	*Distributed Version Control System*
IDE	*Integrated Development Environment*
IFRN	Instituto Federal do Rio Grande de Norte
IFSULDEMINAS	Instituto Federal do Sul de Minas
JSON	*JavaScript Object Notation*
LPS	Linhas de Produtos de Software
SUAP	Sistema Unificado de Administração Pública
VCS	*Version Control Systems*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo está organizado como a seguir. A Seção 1.1 fornece uma visão geral do problema abordado nesta dissertação de mestrado. A Seção 1.2 descreve a solução proposta. A Seção 1.3 apresenta limitações de escopo para o estudo realizado. A Seção 1.4 apresenta uma visão geral da estrutura da dissertação. Por fim, a Seção 1.5 apresenta a publicação gerada por este trabalho.

1.1 Problema

Os Sistemas de Controle de Versão (*Version Control Systems* – VCS) proveem aos desenvolvedores técnicas e ferramentas para lidar com a criação e a evolução de sistemas de software (MENS, 2002; NGUYEN; IGNAT, 2018). Dentre essas destaca-se a prática de *fork*, a qual cria novos projetos (a.k.a., *projetos customizados*) a partir da cópia de projetos existentes (a.k.a., *projetos originais*) (JIANG et al., 2017; LI et al., 2017; STĂNCIULESCU; SCHULZE; WĄSOWSKI, 2015). Modernos VCS, como `Git`[1] e `Subversion`[2] , aliados ao surgimento de plataformas web de desenvolvimento como o `GitHub`[3], fomentaram o surgimento de novos sistemas de software independentes baseados em *forks* de sistemas populares (BORGES; HORA; VALENTE, 2016b; ZHOU et al., 2018). Por exemplo, o projeto Bootstrap[4], inicialmente desenvolvido no Twitter e disponível no GitHub, possui em dezembro de 2020 mais de 71 mil *forks*.

Apesar dos projetos customizados possuírem evolução própria, ou seja, terem liberdade e independência para criar ou modificar funcionalidades, o *problema* abordado nesta dissertação é quando esses sistemas customizados querem se manter atualizados frente aos projetos originais, i.e., querem obter novas *features*, correções de *bugs*, melhorias de desempenho, etc. A mesclagem do código do projeto original (*PO*) no projeto customizado (*PC*) pode gerar inúmeros conflitos (KASI; SARMA, 2013; MAHMOOD et al., 2020; MCKEE et al., 2017; NISHIMURA; MARUYAMA, 2016) em decorrência, por exemplo: (i) o PC criou uma nova classe de domínio em uma tela, porém o PO também criou só que de forma diferente; (ii) o PC alterou o tamanho de um campo, porém o PO também alterou mas para um valor diferente e (iii) o PC corrigiu um *bug*, porém o PO também corrigiu só que de forma diferente.

[1] Disponível em <https://git-scm.com/>.
[2] Disponível em <https://subversion.apache.org/>.
[3] Disponível em <https://github.com/>.
[4] Disponível em <https://github.com/twbs/bootstrap>.

Embora seja algo rotineiro no desenvolvimento de software, solucionar conflitos de mesclagem de código pode ser considerada uma tarefa árdua para desenvolvedores (NISHIMURA; MARUYAMA, 2016; MCKEE et al., 2017; CAVALCANTI; ACCIOLY; BORBA, 2015). Além de custosa, quando a resolução dos conflitos é mal executada, leva a erros de integração e interrupção do fluxo de trabalho, comprometendo a eficiência e o cronograma do projeto (MCKEE et al., 2017). Estudos apontam que essa realidade está presente em aproximadamente 19% das mesclagens de código (MCKEE et al., 2017), podendo chegar a percentuais maiores (34% a 54%) em função das características dos projetos (KASI; SARMA, 2013).

No cenário desta dissertação, acredita-se em um alto percentual de conflitos, com base (i) na literatura da área, (ii) em resultados apurados através de uma avaliação histórica conduzida sobre um projeto real de software, e (iii) na experiência em resolução de conflitos de mesclagem do autor desta dissertação, que também é desenvolvedor do projeto real analisado.

Como exemplo da literatura, cita-se o trabalho de Sung et al. (2020), onde evidenciou-se a maior ocorrência de conflitos de mesclagem em cenários onde dois projetos independentes, com diferentes equipes, trabalham de forma paralela e, mais importante, a equipe do projeto original sequer tem conhecimento do que vem sendo customizado em seus *forks*, da mesma forma que as mudanças na origem ocorrem sem conhecimento dos projetos customizados.

Em uma avaliação histórica, que abrangeu um período de quatro anos de desenvolvimento de um grande projeto de software, constatou-se que o sistema analisado atualizava-se frente à sua origem em média uma vez a cada um mês e cinco dias. Além disso, observou-se um número crescente de conflitos de mesclagem, elevado à medida que o volume de customizações locais aumentava. Uma análise detalhada dessa avaliação é apresentada no Capítulo 5.

1.2 Solução Proposta

Esta dissertação tem como *objetivo geral* propor uma abordagem sistemática que busca eliminar os conflitos de mesclagem de código e reduzir a necessidade de intervenção humana, conduzindo o desenvolvimento de *forks* que devem ser atualizados frente ao projeto original de uma forma **não *ad hoc***, onde *features* são modularizadas, documentadas, rastreáveis e podem ser reutilizadas. Cada customização é composta de um arquivo descritor – o qual fomenta modularidade, documentação e rastreabilidade – e classes locais que implementam as customizações de forma não invasiva. Os conflitos de mesclagem são evitados e, em contrapartida, *conflitos de customização*, para os quais acredita-se que a resolução é mais simples (ver Seção 3.5) pas-

sam a existir. De modo a alcançar o *objetivo geral* definido nesta dissertação, quatro *objetivos específicos* são definidos e discutidos no decorrer desta seção.

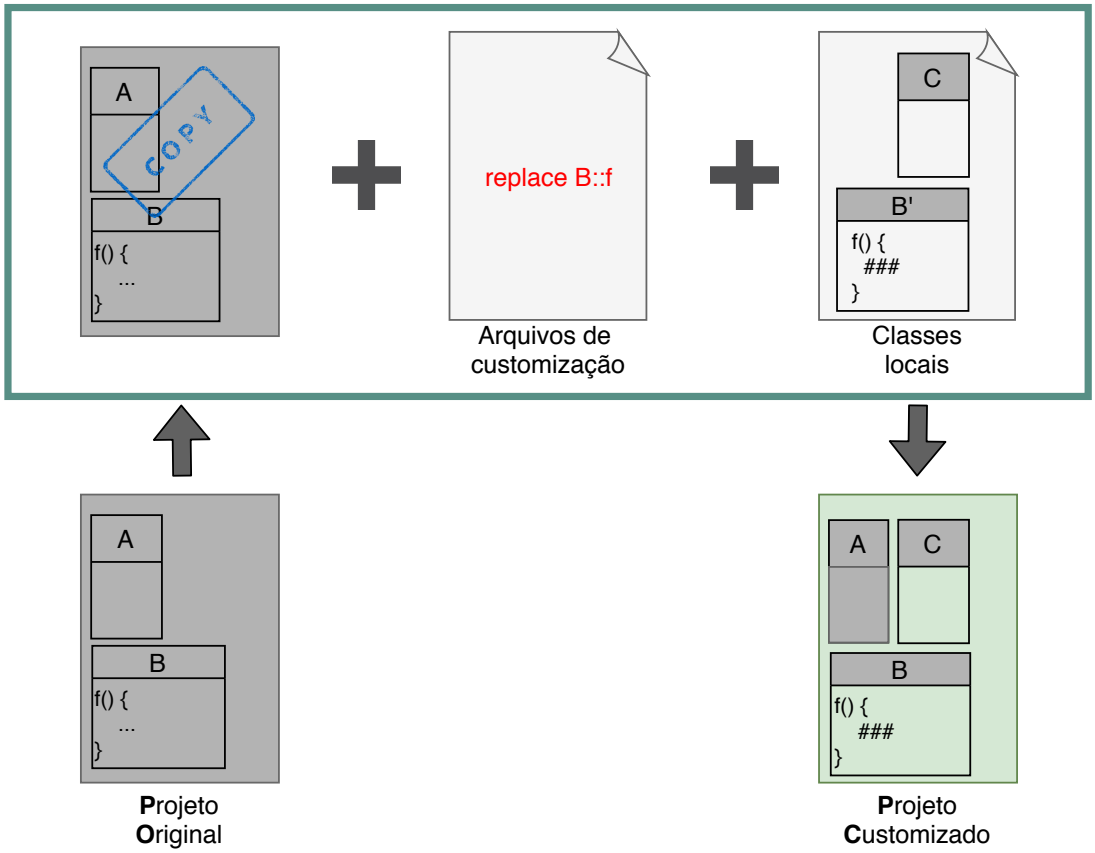
A ideia central da abordagem é o projeto customizado ser desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original", i.e, PC = PO + customizações. Ao invés de alterar o corpo de um método `foo` do projeto original, o desenvolvedor o implementa localmente e especifica, por meio de uma das onze instruções da DSL proposta, algo como "substitua o método `foo` pela implementação local". Isso permite que as atualizações sejam realizadas de forma não invasiva, preservando a independência do projeto customizado e não exigindo intervenções diretas no projeto original. Para tanto, a solução baseia-se na descrição de *features* em alto nível através de uma DSL (*Domain Specific Language* – Linguagem de Domínio Específico), criada de forma empírica através de um estudo realizado sobre um sistema real de software. A definição dessa DSL é o primeiro *objetivo específico* desta dissertação (ver Seção 3.4.1.).

Outras opções de solução foram consideradas como alternativas ao uso de uma DSL: (i) compilação condicional e (ii) LPS (Linhas de Produtos de Software). Uma solução baseada em compilação condicional exige intervenções no projeto original, de modo a viabilizar as customizações realizadas no projeto customizado. Tal solução é invasiva e cria uma dependência não desejada entre o projeto customizado e sua origem. O uso de uma LPS não se encaixa ao cenário do problema tratado nesta dissertação. Produtos de software personalizados por meio de uma LPS são obtidos a partir da seleção automática de recursos (características) desenvolvidos sob uma base de código comum (MARTINS, 2019). No cenário desta dissertação, a base de código do PC não é a mesma do PO. Além disso, apesar de dar origem ao PC, a base de código do PO também muda ao longo do tempo. Por fim, as bases de código do PO e do PC não são organizadas em "recursos" e não existe forma consistente e sistemática capaz de combiná-las de modo a obter o código do projeto customizado.

Conforme ilustrado na Figura 1.1, o processo de atualização proposto nesta dissertação de mestrado busca ser simples. Realiza-se uma cópia exata do código fonte atual do PO. Em seguida, realiza-se o *parsing* das instruções de customização obtidas em arquivos de customização. No exemplo, o método `f` da classe `B` deve ser trocado. Novas implementações para unidades de código (classes, métodos e funções) são recuperadas em arquivos de classes locais. No exemplo, uma nova implementação do método `f` é obtido na implementação local da classe `B'`. São recuperados ainda em arquivos locais implementações completas de novas unidades de

código, inexistentes em PO, como a classe **C**. Ao final do processo de atualização é obtido o código fonte do sistema PC.

Figura 1.1 – Abordagem para atualização de *forks* frente ao projeto original.



Fonte: Do autor (2021).

Uma ferramenta que automatiza a abordagem proposta foi desenvolvida para conduzir uma avaliação em um projeto real de software frequentemente atualizado frente ao seu projeto original. Nesta dissertação de mestrado a criação dessa ferramenta é o segundo *objetivo específico* (ver Capítulo 4) e a realização da avaliação é o terceiro *objetivo específico* (ver Capítulo 5).

Como resultado mais relevante, a abordagem conseguiu reduzir 752 conflitos que ocorreram durante 42 atualizações em um período de quatro anos (ver Seção 5.5). De forma complementar, outra avaliação foi conduzida com o objetivo de avaliar a perspectiva de desenvolvedores de um projeto real com relação à abordagem proposta. A avaliação junto aos desenvolvedores é o quarto *objetivo específico* definido nesta dissertação. Em suma, pôde-se concluir que a abordagem proposta apresentou boa aceitação pelos participantes e que é possível aplicá-la em cenários reais de desenvolvimento.

É importante destacar que a solução proposta se difere dos recursos existentes em sistemas de controle de versão como a opção *ours* no *git merge*, o *git rerere* e o *git patch*. O detalhamento das diferenças pode ser encontrado no Capítulo 7.

1.3 Limitações de Escopo

A abordagem de customização proposta nesta dissertação de mestrado é voltada exclusivamente a projetos customizados que querem se manter atualizados frente ao projeto original. Dentro desse cenário, foi idealizada como uma abordagem genérica, capaz de lidar com diferentes linguagens e diferentes sistemas de controle de versão. Contudo, existem limitações quanto ao escopo do estudo realizado. Acredita-se que essas limitações não impedem a adoção da abordagem em outros cenários, mas faz-se necessário apresentá-las.

O processo empírico de construção das instruções de customização da DSL proposta foi realizado por meio de análise de conflitos de mesclagem de um único sistema de software. O mesmo sistema foi utilizado nas avaliações realizadas, limitando assim a aplicação da abordagem durante o estudo a uma única linguagem (Python). Todavia, as instruções de customização foram elaboradas de modo a contemplar diversos cenários, independentemente da linguagem adotada. Além disso, não há impeditivos para que novas instruções sejam propostas no futuro.

No que diz respeito à ferramenta de customização, alterações são necessárias no módulo Execução, uma vez que esse lida apenas com código Python, em particular na construção de árvores de sintaxe abstrata utilizadas para manipulação do código. Propõe-se, como solução, a criação de diversos módulos, cada qual capaz de manipular código de uma linguagem específica. O uso de uma ferramenta como o ANTLR[5], que possui um analisador léxico e sintático que contempla diversas linguagens de programação[6], pode facilitar essa tarefa e é proposto como trabalho futuro.

Deve-se destacar que **não** há garantias diretas de qualidade ao se adotar a abordagem. Sua adoção não dispensa a realização de outras atividades comuns em cenários de resolução de conflitos, como a realização de inspeções e testes. Logo, as garantias de qualidade que a abordagem oferece são as mesmas que a resolução manual de conflitos, i.e., podem ocorrer problemas que precisam ser detectados por outras atividades do processo de desenvolvimento de software.

[5] Disponível em <https://www.antlr.org/>.

[6] Linguagens disponíveis em Janeiro de 2021: Java, C, Python (2 e 3), JavaScript, Go, C++, Swift, PHP e Dart.

Por fim, este estudo **não** foca em avaliar a questão de esforço. Aceita-se que o esforço de desenvolvimento pode ser até maior a curto prazo, mas o fato de modularizar – em 1 arquivo – *n* modificações que produzem uma nova *feature* ou alterem uma existente facilita a manutenibilidade a médio e longo prazo, além de evitar um código confuso que mistura customizações com o código original.

1.4 Estrutura da Dissertação

Esta dissertação enquadra-se em uma pesquisa de *design research* pois seu foco é resolver um problema (Seção 1.1) criando, para tanto, uma solução específica (Seção 3). Parte-se da análise de um universo micro (Seção 5.2) e cria-se uma solução para o problema apresentado. A solução é validada para esse universo (Capítulos 5 e 6) para então discutir-se como ela pode ser generalizada (Seção 8.4).

O restante desta dissertação está estruturado como a seguir. O Capítulo 2 introduz conceitos fundamentais, como `Git`, *branch*, *fork*, sistemas de controle de versão, análise estática e DSL. O Capítulo 3 descreve a abordagem proposta, apresentando as instruções de customização idealizadas. O Capítulo 4 apresenta a ferramenta implementada. O Capítulo 5 discute os resultados da avaliação da abordagem proposta aplicada ao longo de um período histórico de um sistema real frequentemente atualizado frente ao seu projeto original. O Capítulo 6 avalia a perspectiva dos desenvolvedores de um projeto real com relação à abordagem proposta. Por fim, o Capítulo 7 apresenta trabalhos relacionados e o Capítulo 8 conclui.

1.5 Publicações

Esta dissertação gerou o seguinte artigo relacionado:

- Arthur Roberto Marcondes and Ricardo Terra. An approach for updating forks against the original project. Em: *34th Simpósio Brasileiro de Engenharia de Software (SBES)*, páginas 1-10, 2020.

2 *BACKGROUND*

Este capítulo apresenta os conceitos fundamentais necessários para o entendimento desta dissertação de mestrado. A Seção 2.1 trata dos Sistemas de Controle de Versão, com enfoque a conceitos fundamentais para esta dissertação, como `Git`, *branch* e *fork*, além de outros conceitos relacionados. A Seção 2.2 apresenta uma definição para Análise Estática de Código, bem como o conceito de Árvore de Sintaxe Abstrata – AST *(Abstract Sintax Tree)*. A Seção 2.3 define o conceito de uma Linguagem de Domínio Específico – DSL *(Domain Specific Language)*. Por fim, a Seção 2.4 discute as considerações finais do capítulo.

2.1 Sistemas de Controle de Versão

Os VCS são ferramentas que suportam trabalho paralelo em projetos compartilhados, oferecendo meios para a sincronização de alterações simultâneas no código fonte de uma aplicação (NGUYEN; IGNAT, 2018). Tornaram-se amplamente adotados nas últimas décadas e atuam no gerenciamento de alterações em repositórios de software e em vários tipos de dados (XU et al., 2019). São exemplos softwares como `Dropbox`[1] e `SharePoint Online`[2], que adotam o VCS interno para o gerenciamento de documentos, incluindo imagens, vídeos etc. (XU et al., 2019).

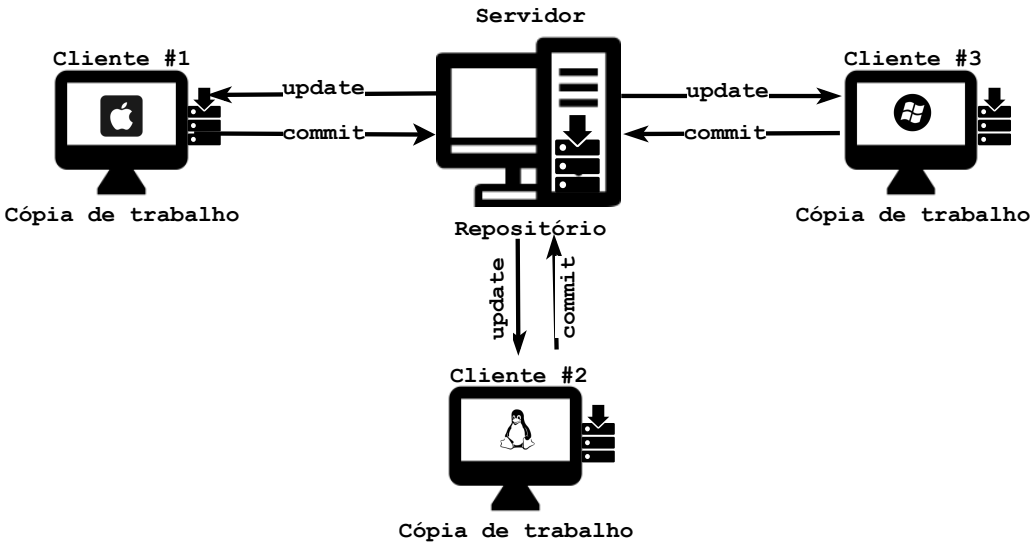
2.1.1 Classificação

Os Sistemas de Controle de Versão podem ser classificados em:

- CVCS (VCS centralizado): baseia-se na arquitetura cliente-servidor, na qual um servidor central fornece a versão de referência e cada cliente se comunica com o servidor para sincronizar suas modificações (Figura 2.1). O servidor mantém um histórico completo das versões, enquanto os clientes mantêm apenas uma cópia local dos documentos compartilhados. Os usuários podem modificar em paralelo sua cópia local e sincronizar com o servidor central para publicar suas contribuições e torná-las visíveis aos outros colaboradores (NGUYEN; IGNAT, 2018). São exemplos de CVCS: CVS[3] e `Subversion`;
- DVCS (VCS descentralizado): também denominado VCS distribuído, conta com uma arquitetura ponto a ponto em que cada cliente mantém o histórico de versões, além de

[1] Disponível em <https://www.dropbox.com/>.
[2] Disponível em <https://products.office.com>.
[3] Disponível em <https://www.nongnu.org/cvs/>.

Figura 2.1 – Sistema de Controle de Versão Centralizado.



Fonte: Do autor (2021).

uma cópia local dos documentos compartilhados. Os usuários podem trabalhar isoladamente em seus repositórios locais, sincronizando-os sempre que necessário com repositórios de outros colaboradores (Figura 2.2) por meio de uma troca de modificações ponto a ponto (XU et al., 2019). DVCS tornaram-se populares a partir de 2005 (NGUYEN; IGNAT, 2018) com destaque para o `Git`, `Mercurial`[4] e `Darcs`[5].

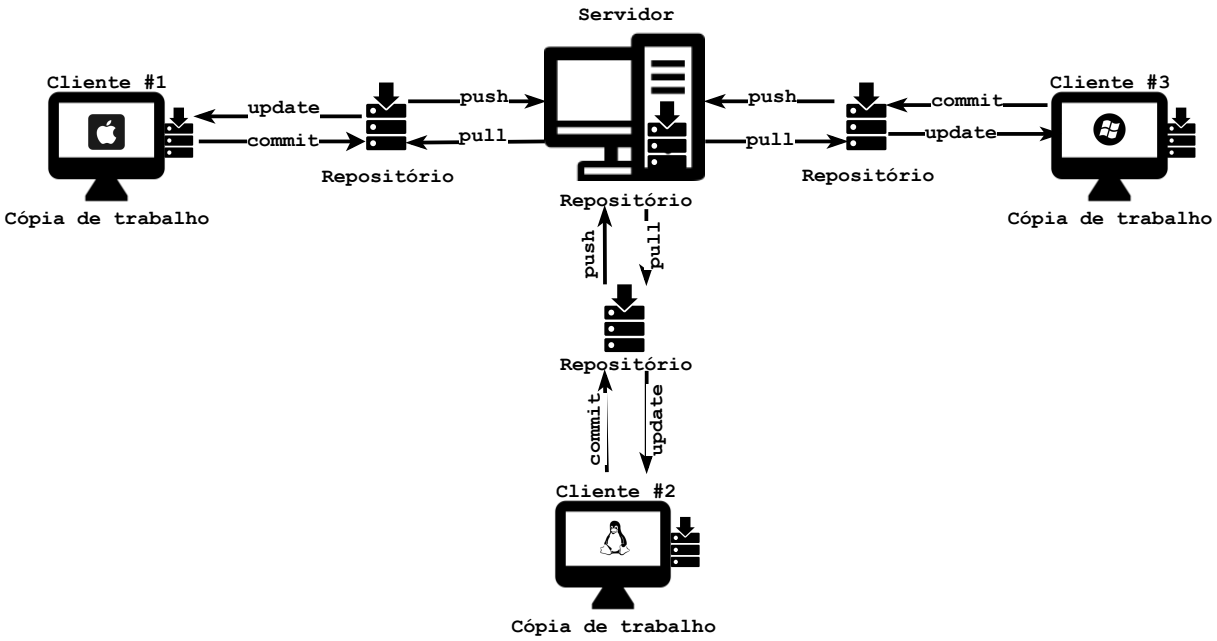
Mesmo sendo amplamente utilizados na Academia e na Indústria CVCS possuem limitações inerentes à sua arquitetura centralizada: limitam a escalabilidade e a tolerância a falhas, os custos de administração não são compartilhados e a centralização de dados nas mãos de um único ponto é uma ameaça à privacidade (NGUYEN; IGNAT, 2018). Por outro lado DVCS fornecem cooperação mais eficiente e flexível e evitam o ponto único de falha. Sua arquitetura ponto a ponto permite suportar grande quantidade de usuários e cada usuário mantém uma cópia do histórico e decide com quem compartilhá-lo sem armazená-lo em um servidor central. Esses recursos tornaram DVCS amplamente utilizados no domínio do desenvolvimento de software aberto, onde os projetos geralmente têm grande quantidade de colaboradores (XU et al., 2019).

2.1.2 Git e GitHub

O `Git` é um Sistema de Controle de Versão Distribuído disponível em todas as plataformas de desenvolvimento convencionais por meio de uma licença de software livre (BLAW,

[4] Disponível em <https://www.mercurial-scm.org>.
[5] Disponível em <http://darcs.net/>.

Figura 2.2 – Sistema de Controle de Versão Descentralizado.



Fonte: Do autor (2021).

2018). É usado ativamente em projetos de pequeno a grande porte, por grandes equipes de desenvolvimento a profissionais independentes. É uma solução de código aberto cuja comunidade contribui ativamente na melhoria do projeto. Sua principal vantagem é ser um sistema de ramificação leve e fácil de usar. Cada ramificação ou *branch* pode ser a linha principal de trabalho ou uma linha específica criada para desenvolvimento de uma nova funcionalidade, para correção de um *bug* ou para qualquer outro trabalho em andamento (BLAW, 2018).

O `Git` possui a capacidade de associar um repositório local a vários repositórios remotos, permitindo que desenvolvedores e gerentes de projetos criem uma variedade de fluxos de trabalho distribuídos, em sua maioria impossíveis de executar em um sistema tradicional de controle de versão centralizado (SPINELLIS, 2012). O uso do repositório local permite maior responsividade, maior facilidade de configuração e a capacidade de operação sem conectividade com a Internet.

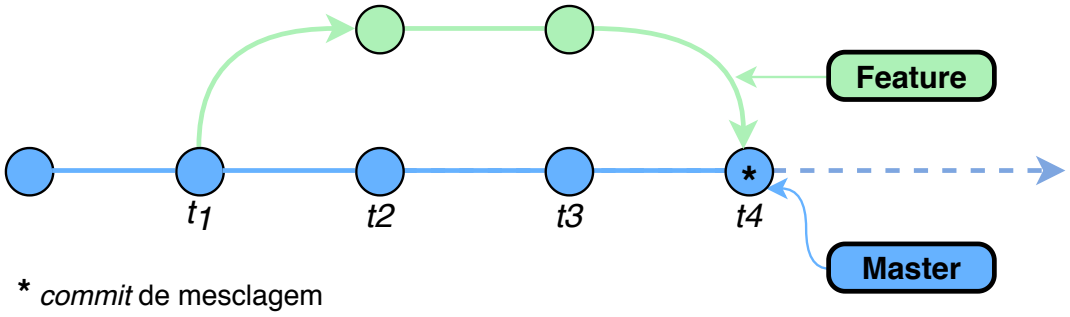
Com seu sistema de controle de versão baseado no `Git`, o `GitHub` representa uma nova geração de plataformas colaborativas *Web* que fornecem ferramentas para facilitar o desenvolvimento distribuído, inclusive para desenvolvimento de software de código aberto (COSENTINO; IZQUIERDO; CABOT, 2017). Possui recursos tradicionais, como hospedagem gratuita de código fonte, aliados a recursos sociais, destinados a facilitar a colaboração e as interações sociais em torno dos projetos (COSENTINO; IZQUIERDO; CABOT, 2017). Lançado em 2008, sua

quantidade de usuários cresceu exponencialmente, passando de 135.000 projetos em 2009 para mais de 35 milhões em 2015 (COSENTINO; IZQUIERDO; CABOT, 2017). Atualmente é o maior repositório de código fonte do mundo (BORGES; HORA; VALENTE, 2016a).

2.1.3 *Branch*

Um *branch* é uma ramificação do projeto de software em um VCS. Pode ser a ramificação principal de trabalho (*master*) ou uma ramificação criada para desenvolvimento de uma funcionalidade em paralelo. A Figura 2.3 apresenta uma visão simplificada de uma *branch* criada para implementação de uma *feature* qualquer.

Figura 2.3 – Um *branch* para desenvolvimento de uma *feature* em paralelo à ramificação principal.



Fonte: Do autor (2021).

Observe que a ramificação *master* refere-se à linha de desenvolvimento principal. No tempo t_1, uma nova ramificação *feature* é criada, trabalhada em paralelo e mesclada na ramificação principal somente no tempo t_4. Conforme Spinellis (SPINELLIS, 2012), essa é uma prática comum pois permite que seja criada, a qualquer momento, uma cópia privada completa e instantânea do repositório de trabalho atual.

2.1.4 *Fork*

Fork é o conceito de criar uma cópia completa de um projeto de software (STĂNCIULESCU; SCHULZE; WĄSOWSKI, 2015), normalmente com dois objetivos distintos: (i) possibilitar o trabalho local de um desenvolvedor que vai atuar na correção e na evolução daquele sistema ou (ii) criar um novo sistema de software a partir de um sistema existente. Popularmente e também nesta dissertação, o termo *fork* é empregado no segundo objetivo (RUBIN et al., 2012).

Um projeto de software (a.k.a., *projeto customizado*) criado a partir do *fork* de um *projeto original* tem total independência para evoluir conforme suas necessidades (ZHOU et al.,

2018). No entanto, em certos cenários, o projeto customizado necessita aproveitar novas características incorporadas ao projeto original, bem como evoluções e correções de *bugs* (STĂNCIULESCU; SCHULZE; WĄSOWSKI, 2015). Isso representa uma vantagem ao projeto independente, pois implica em economia de recursos.

Diante disso, deve-se realizar a mesclagem do código do projeto original no projeto customizado, o que pode resultar em inúmeros conflitos de mesclagem. Isso se justifica por causa da complexidade em mesclar um mesmo trecho de código alterado no projeto original e no projeto customizado. Se tais conflitos ocorrem rotineiramente em mesclagem de código dentro de um mesmo projeto, conjectura-se que tais conflitos ocorram com mais frequência quando se trata de *forks* que envolvem dois projetos independentes.

2.1.5 Mesclagem de Código

A mesclagem de código é uma atividade inerente ao desenvolvimento de software paralelo e colaborativo, sendo um aspecto essencial da manutenção e na evolução de sistemas de software de larga escala (MENS, 2002). Também denominada fusão de software, é a atividade que permite que linhas paralelas e independentes sejam fundidas em uma só de tal forma a gerar uma nova versão de um sistema (MENS, 2002). Dentre outras classificações, a mesclagem de código pode ser categorizada em (ACCIOLY; BORBA; CAVALCANTI, 2018):

- Textual: artefatos de software são tratados como arquivos de texto. Linhas são unidades indivisíveis, que podem ser inseridas, excluídas, modificadas ou movidas. Possui desvantagens, mas ainda é utilizada por causa da sua eficiência, escalabilidade e precisão. DVCS populares, como o `Git`, trabalham dessa forma;

- Sintática: mais poderosa que a textual porque também leva em conta a sintaxe de artefatos de software. As técnicas utilizam estruturas de dados subjacentes como grafos e árvores como apoio ao processo de fusão, o que acaba causando sobrecarga de desempenho durante a análise da mesclagem;

- Semântica: busca identificar conflitos semânticos não identificados por abordagens sintáticas. Mesmo sendo mais abrangente, ainda assim não consegue identificar todos os tipos de conflitos semânticos, pois alguns ocorrem somente durante a execução de um programa e só podem ser identificados por abordagens que utilizam formalismos matemáticos complexos;

2.1.6 Conflitos de Mesclagem

Os conflitos de mesclagem ocorrem durante o processo de fusão de linhas paralelas de desenvolvimento. Inúmeras técnicas de mesclagem foram propostas na literatura para dar suporte a essa tarefa (MENS, 2002), (GUIMARÃES; SILVA, 2012), (NISHIMURA; MARUYAMA, 2016), (COSTA et al., 2016), (ACCIOLY; BORBA; CAVALCANTI, 2018), mas automatizar completamente o processo ainda é um desafio, fazendo com que seja inevitável a intervenção dos desenvolvedores para conclusão do trabalho.

Estudos apontam que a ocorrência de conflitos de mesclagem está presente em aproximadamente 19% dos *merges* de código (MCKEE et al., 2017; NELSON et al., 2019). A resolução desses conflitos não é trivial e representa uma tarefa árdua para desenvolvedores (NISHIMURA; MARUYAMA, 2016; MCKEE et al., 2017). É um processo custoso que, quando mal executado, leva à erros de integração, interrupção de fluxo de trabalho e outros problemas, comprometendo a eficiência e cronogramas do projeto (MCKEE et al., 2017). LEßENICH et al. (2018) apontam em seus estudos que 38% dos desenvolvedores evitam as mesclagens de código temendo os conflitos, o que acaba fomentando a fusão tardia do código, agravando os problemas.

No cenário particular de projetos customizados que buscam atualizações em seus respectivos projetos originais, a complexidade e o custo de resolução desses conflitos tornam-se maiores. Sung et al. (2020) enunciam três razões: (i) as mudanças no projeto original ocorrem sem o conhecimento do projeto customizado; (ii) não é trivial encontrar a causa raiz de um conflito de mesclagem, pois o histórico de *commits* da origem pode conter vários milhares de *commits*; e (iii) uma quebra de compilação ou conflito causada pela mesclagem de código pode ocorrer por mudanças no projeto customizado e não no projeto original e ainda muitas vezes realizadas em *commits* antigos. Isso torna difícil encontrar o desenvolvedor correto para corrigir o problema, que pode ter deixado o projeto.

2.2 Análise Estática de Código

A análise estática de código é uma técnica que examina o código fonte de uma aplicação sem realizar entrada de dados e sem executá-lo. Ela pode detectar possíveis violações de segurança, como injeção de SQL, pode antecipar erros que ocorrerão em tempo de execução, como

desreferência de um objeto nulo, e indicar inconsistências lógicas, como um teste condicional que possivelmente nunca será verdadeiro (AYEWAH et al., 2008).

Em vez de tentar provar que o código fonte de uma aplicação atende às suas especificações, as ferramentas de análise estática procuram violações de princípios, práticas ou diretivas de programação. Assim, procuram por trechos do código que possam fazer desreferência de objetos nulos ou causar estouros no limite de arranjos, operações lógicas que nunca poderão ser verdadeiras e que levarão a um comportamento incorreto do programa, ou ainda sinalizam problemas de estilo de programação, como convenções de nomenclatura ou o uso de chaves em condicionais e estruturas de *loop* (AYEWAH et al., 2008).

Ferramentas baseadas em análise estática existem desde os anos 80. Ferramentas mais recentes conseguem fazer análises mais robustas descobrindo mais defeitos e produzindo menor quantidade de falsos positivos. Podem ainda lidar com grandes aplicações com milhões de linhas de código (EMANUELSSON; NILSSON, 2008).

Árvore de Sintaxe Abstrata: Uma árvore de sintaxe abstrata (AST) é uma maneira de representar a sintaxe de uma linguagem de programação como uma estrutura hierárquica semelhante a uma árvore. Cada nó da árvore indica uma construção que ocorre no código fonte (SANTOS, 2019). Uma AST é geralmente resultado de uma fase da análise de sintaxe realizada pelo compilador (PORFIRIO; PEREIRA; MASCHIO, 2017).

Muitas ferramentas de modificação e de geração de código fonte, como Ambientes de Desenvolvimento Integrado (*Integrated Development Environment* - IDE), baseiam-se em AST para manipulação do código fonte. O uso de árvores de sintaxe abstrata também é observado na literatura, como em trabalhos voltados ao estudo de conflitos de mesclagem de código (ACCIOLY et al., 2018; LEßENICH et al., 2018; MAHMOUDI; NADI, 2018; HATTORI; LANZA, 2010). Nesses trabalhos são propostas ferramentas que utilizam AST como parte do processo de análise, tal como Sung et al. (2020).

2.3 Linguagem de Domínio Específico

DSL (*Domain Specific Language* -– Linguagem de Domínio Específico) são linguagens criadas para resolução de problemas específicos de um dado domínio. Ao contrário das Linguagens de Propósito Geral, uma DSL possui expressividade bastante limitada, possibilitando ao usuário construir estruturas que modelem de forma clara e concisa funcionalidades

específicas de determinado domínio (FOWLER, 2010). DSL podem ser classificadas em duas categorias (FOWLER, 2019):

- Internas: utilizam a infraestrutura de uma linguagem de programação existente, chamada de linguagem nativa. São implementadas como bibliotecas e componentes que estendem a linguagem nativa criando novos tipos de dados, rotinas, procedimentos, macros etc.;
- Externas: são linguagens que possuem uma infraestrutura própria para as etapas de análise (*parsing*), interpretação, compilação e geração de código. Sendo assim, desenvolver uma DSL externa é como desenvolver uma linguagem de programação própria. É comum o uso de outras estruturas para representação de DSL externas, tais como estruturas da linguagem XML (*Extensible Markup Language*) ou JSON (*JavaScript Object Notation*).

DSL mais comuns são textuais (FOWLER, 2019) mas há uma gama crescente de DSL gráficas. Por exemplo, as *language workbenches* são ferramentas que funcionam como IDE de desenvolvimento, disponibilizando recursos gráficos que visam expressar o modelo de negócio de forma não textual, com uso de diagramas de diversos tipos, gráficos, estruturas de relacionamento e planilhas (TOMASSETTI, 2017). Populares nos dias atuais, permitem que muitos problemas de negócio sejam facilmente representados de forma gráfica, eliminando limitações e excessos de complexidade que as representações textuais oferecem.

2.4 Considerações Finais

Neste capítulo, foram apresentados os conceitos fundamentais para o entendimento desta dissertação de mestrado. Foram inicialmente introduzidas as definições de VCS, que provém as ferramentas necessárias para que desenvolvedores possam trabalhar de forma colaborativa.

Foram apresentados os conceitos sobre o VCS distribuído `Git` e sobre a plataforma *Web* com maior repositório de código fonte do mundo, o `GitHub`. Ambos popularizaram a prática do *forking* (ou clonagem em grande escala) utilizada na Indústria e em projetos de código aberto para criação de novos sistemas a partir de sistemas existentes.

O trabalho colaborativo e paralelo é sustentado pelos VCS que possibilitam a fusão de diferentes linhas de desenvolvimento em uma única linha, criando uma versão de um sistema. Essa ação, denominada mesclagem ou fusão de código, foi conceituada neste capítulo, bem como os problemas por ela gerados: conflitos de mesclagem. Mesmo com diversos estudos

na literatura tratando do assunto, ainda não foi possível automatizar plenamente o processo de mesclagem de código de tal forma que não ocorram conflitos. Esta dissertação propõe uma abordagem para evitar os conflitos de mesclagem, no caso particular de sistemas criados a partir do *forking* de outros sistemas que necessitam manter-se atualizados frente ao código fonte dos projetos que lhes deram origem.

Foram apresentados ainda conceitos sobre análise estática de código, técnica utilizada nesta dissertação para customização do sistema derivado, e sobre Árvore de Sintaxe Abstrata, utilizada na implementação da ferramenta de customização. Por fim, definiu-se DSL, utilizada nesta dissertação de mestrado para especificar regras de customização a serem aplicadas sobre o código fonte de um projeto original de tal forma a torná-lo um projeto customizado.

Uma vez apresentados os conceitos fundamentais para entendimento deste estudo, em especial o conceito de DSL, o próximo capítulo trata da abordagem de customização de sistemas, cujo ponto central é a DSL que descreve as regras de customização.

3 ABORDAGEM DE CUSTOMIZAÇÃO

A ideia central da abordagem proposta nesta dissertação é a de que um projeto customizado deve ser desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original". Dito isso, a abordagem é melhor aplicável a cenários de projetos recém criados, com baixo volume de customizações próprias. Não há restrição para sua adoção por projetos já em andamento, mas nesse caso todas as customizações já existentes no projeto customizado devem ser adaptadas à estrutura proposta pela abordagem. Logo, a dificuldade de adoção da abordagem por um projeto em andamento é diretamente proporcional ao volume de customizações próprias já realizadas.

Este capítulo indica como definir as *features* e como descrever as alterações por meio de instruções de customização. Mais importante, este capítulo provê evidências de que a abordagem proposta promove modularidade, documentação, rastreabilidade e reúso, além de evitar conflitos de mesclagem. A Seção 3.1 apresenta um sistema motivador, utilizado ao longo do capítulo para ilustrar a aplicação da abordagem. A Seção 3.2 define o conceito de *feature* no contexto da abordagem e as diretrizes existentes em um arquivo de customização. A Seção 3.3 apresenta um potencial *workflow* para aplicação da abordagem, discutindo-o com detalhes. A Seção 3.4 discorre sobre a etapa de levantamento das instruções de customização, e apresenta detalhes de cada uma das onze instruções criadas, como motivação, sintaxe e exemplo prático. Para algumas instruções é apresentado ainda um exemplo motivador, dentro do escopo do sistema motivador apresentado na primeira seção deste capítulo. A Seção 3.5 aborda os conflitos de customização, que são situações de conflito que podem ocorrer quando a abordagem é adotada. Os conflitos são categorizados em tipos e para cada tipo é apresentado um cenário, exemplo, instruções afetadas e ações a serem consideradas durante sua análise. Por fim, a Seção 3.6 discute as considerações finais do capítulo.

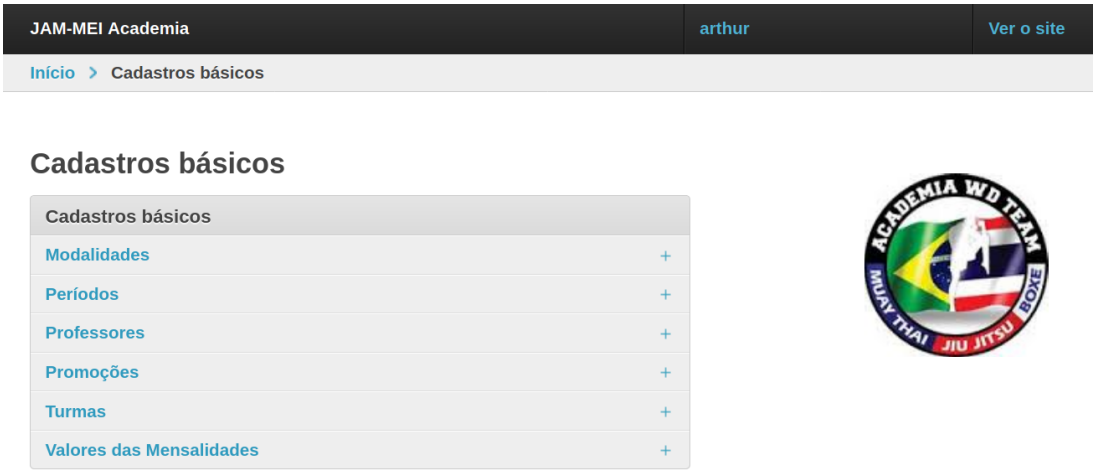
3.1 *Exemplo motivador*

A Figura 3.1 ilustra um sistema web de gerenciamento de academia desenvolvido na linguagem Python sob o *framework* `Django`[1] (a.k.a., *sistema motivador*) . Trata-se de um sistema desenvolvido pelo autor desta dissertação de mestrado em meados de 2015, para fins de

[1] Disponível em <https://www.djangoproject.com/>.

aprendizado das tecnologias envolvidas. O sistema foi comercializado durante algum tempo e depois foi descontinuado. Esse sistema – que é utilizado para ilustrar a abordagem no decorrer deste capítulo – possui cadastros essenciais para atender ao seu propósito, como cadastro de alunos, matrículas, turmas e mensalidades etc.

Figura 3.1 – Tela do módulo Cadastros básicos.



Fonte: Do autor (2021).

Para ilustrar a definição de *features*, foi criado um projeto customizado a partir do *fork* do sistema motivador. Nesse projeto customizado, idealizou-se três *features*:

*#F*1: *Customização do leiaute*: troca do logotipo da academia e cores de elementos em tela;

*#F*2: *Inclusão de recepcionista*: criação de um novo modelo de dados **`Recepcionista`**, com todas as funções de CRUD[2], dentro do módulo do sistema denominado "Cadastros básicos";

*#F*3: *Recepcionista em matrícula*: alteração do modelo de dados **`Matricula`**, dentro do módulo "Alunos", com a inclusão da informação sobre a recepcionista responsável pela efetivação de uma matrícula.

3.2 Definição de *features*

Uma *feature* pode ser definida como uma funcionalidade de um sistema de software que resolve algum problema do mundo real ou que entrega algum benefício aos seus usuários

[2] Na Indústria é o nome popularmente dado a uma tela que realiza as operações básicas *Create, Retrieve, Update e Delete* em uma classe de domínio.

(JUNIOR, 2018). Na abordagem proposta, customizações a serem realizadas em um sistema devem ser organizadas em *features*. Um *arquivo de customização* define uma única *feature* e deve possui as seguintes diretrizes:

- **`#feature`**: identificação única da *feature*;
- **`#description`**: informações sobre a customização, promovendo documentação;
- **`#dependsOn`**: identificação das *features* obrigatórias para essa *feature*, promovendo relações de dependência.

Além do supracitado, essa definição promove (i) modularidade, pois um arquivo de customização engloba todas as alterações que a *feature* requer, e (ii) reúso, pois esse arquivo pode ser utilizado em outros projetos customizados. Uma única *feature* definida em um único arquivo de customização promove também tomadas de decisão estratégicas. Por exemplo, se diversas customizações dependem de uma determinada customização `X`, modificá-la pode não ser uma boa ideia ou deve ser feita com cautela por meio de programação aos pares, por exemplo.

É importante observar que a abordagem não evita problemas com relação à modificação de trechos de código com grande impacto no sistema, tal como a criação ou alteração de modelos de dados, idealizados nas *features #F2* e *#F3* do exemplo motivador. Por outro lado, a abordagem ajuda a detectá-los, já que todas as modificações relacionadas a uma *feature*, como a alteração de um modelo de dados, devem estar contidas no mesmo arquivo de customização. Além disso, a abordagem permite a criação de um mecanismo de retroalimentação, onde um erro é identificado e sua correção utilizada como insumo para nova etapa de análise, dentro de um *workflow* de desenvolvimento, como aquele apresentado em detalhes no Seção 6.5.

Exemplo motivador: A Listagem 3.1 ilustra a definição da *feature* #*F*3. Como pode-se observar, além do nome e de uma descrição, existe a definição de uma dependência da *feature* #*F*3 com a #*F*2.

```
1 #feature 003-recepcionista-matricula
2 #description Altera a Matricula incluindo
3              recepcionista responsável
4 #dependsOn 002-recepcionista
```

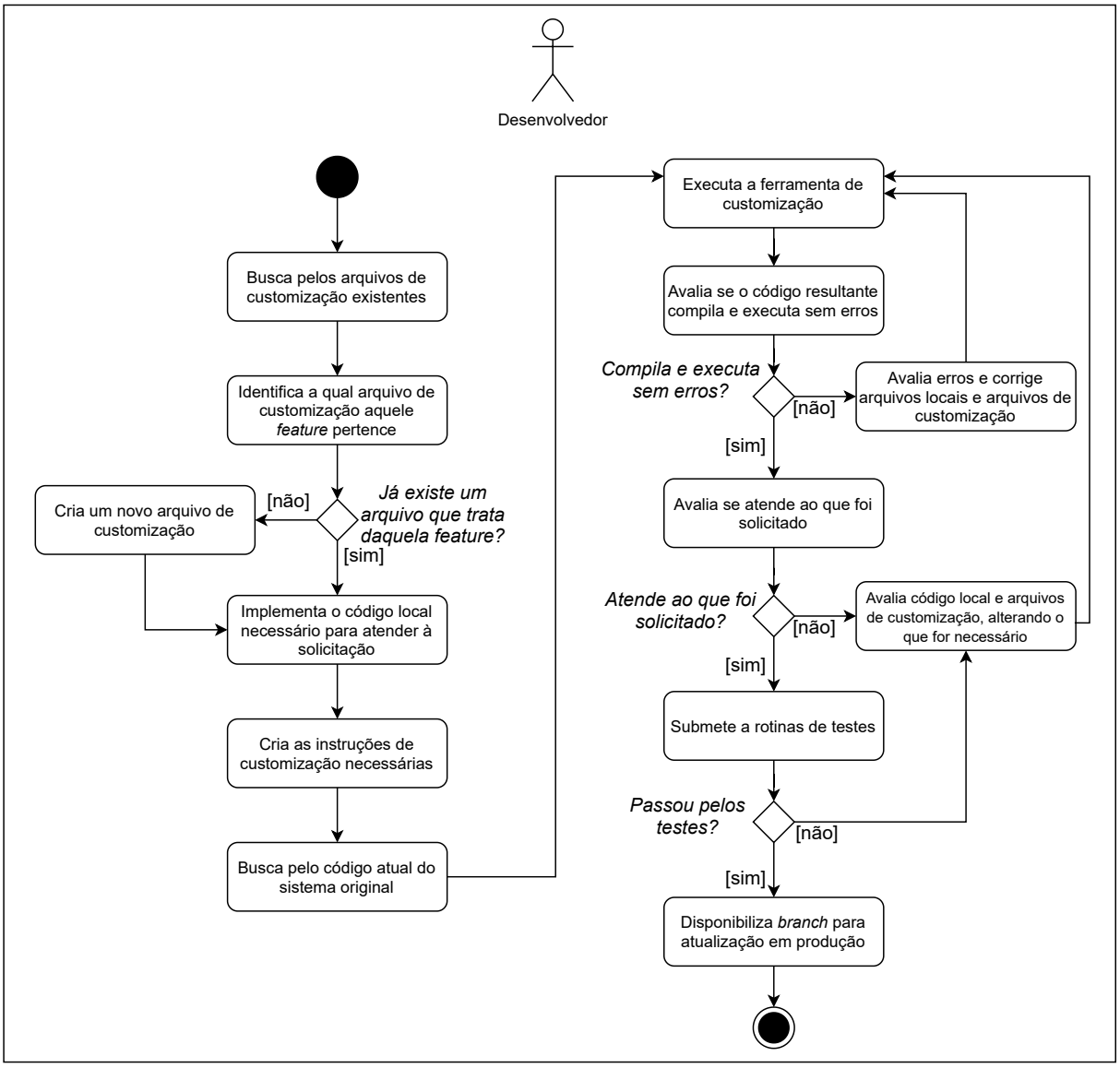
Listagem 3.1 – Fragmento do arquivo de customização F#3

A *feature* #*F*3 foi escolhida como exemplo motivador, pois as *features* #*F*1 e #*F*2 são simples e possuem apenas as diretivas **`#feature`** e **`#description`**.

3.3 *Workflow* de desenvolvimento

Um potencial *workflow* para aplicação da abordagem proposta em um cenário real de desenvolvimento de software é apresentado na Figura 3.2.

Figura 3.2 – Realizar desenvolvimento.



Fonte: Do autor (2021).

Dada uma solicitação de desenvolvimento, o desenvolvedor busca (ou cria) o arquivo de customização que trata da *feature* referente àquela solicitação. Em seguida, implementa o código local necessário para atender à solicitação (a ser integrado pela ferramenta) e cria as

instruções necessárias no arquivo de customização. Finalizada a primeira etapa de desenvolvimento, o desenvolvedor busca pelo código atual do sistema original persistindo-o em um novo *branch* e executa a ferramenta de customização a partir dos arquivos de customização por ele atualizados com o uso dos arquivos locais do projeto customizado.

A etapa seguinte consiste em avaliações do código fonte resultante da execução da ferramenta de customização. Avalia-se se o código resultante da customização compila e executa sem erros, se implementa a solicitação de desenvolvimento e se passa por rotinas de testes e inspeções já estabelecidas no processo de desenvolvimento. Havendo alguma divergência, novas implementações devem ser realizadas sobre os arquivos de customização e arquivos locais, voltando em seguida para a etapa de execução da ferramenta. Essa etapa se repete até que um código fonte correto seja obtido, onde o mesmo é disponibilizado em um *branch* para atualização em ambiente de produção.

Os arquivos de customização e os arquivos locais do projeto customizado devem estar sob o controle de versão, sendo possível modificá-los ao longo do tempo. As customizações são "codificadas" pelos desenvolvedores, ou seja, a abordagem não identifica e deriva as instruções de customização de forma automática.

3.4 Instruções de customização

Um arquivo de customização inclui pelo menos uma instrução de customização, tal como o arquivo apresentado na Listagem 3.1. Essas instruções especificam o que deve ser realizado sobre o código fonte do projeto original de tal forma a transformá-lo no projeto customizado.

3.4.1 Identificação de instruções

O conjunto de instruções foi identificado empiricamente pela análise de diversos conflitos de *merge* durante um ano de desenvolvimento de um *fork* de um sistema real de administração pública. Tal análise envolveu a simulação de conflitos de mesclagem e inspeção manual desses conflitos.

SUAP (Sistema Unificado de Administração Pública): Iniciado em 2007, é um projeto desenvolvido e mantido pelo IFRN (Instituto Federal do Rio Grande de Norte). Trata-se de um sistema com mais de 2.600 arquivos .py distribuídos em 80 módulos, com 330.000 linhas de

código (excluídas linhas em branco e linhas de comentário), além de outros arquivos típicos de uma aplicação web, como arquivos CSS, JavaScript e HTML. Em 2014, passou a ser utilizado e customizado pelo IFSULDEMINAS (Instituto Federal do Sul de Minas). Desde então, diversas customizações foram realizadas no código local desenvolvido pela equipe de desenvolvimento do IFSULDEMINAS, embora obtendo as atualizações realizadas a partir do código do ramo principal do projeto no IFRN.

Simulação de conflitos de merge: Adotou-se arbitrariamente a data de um ano antes do início dos experimentos (17/09/2018). Nessa data, existiam cerca de 460 arquivos com diferenças entre os projetos customizado e original. As *branches* de cada projeto foram retrocedidos para essa data e tentou-se uma operação de fusão do código original no código customizado. Como resultado, obteve-se mais de 200 trechos com conflitos.

Análise dos conflitos: A análise individual de cada um dos 200 conflitos expôs a dificuldade da resolução manual, pois foram despendidas aproximadamente 150 horas na tarefa de análise e resolução dos conflitos, distribuídas ao longo de vinte dias de trabalho. Cada diferença identificada foi utilizada para idealização, generalização e especificação de um conjunto de onze instruções em DSL (Tabela 3.1) que ajudassem a atingir o objetivo de transformar o código do projeto original no código customizado.

Tabela 3.1 – Instruções em linguagem DSL.

#	Instrução	Semântica
1	add @annotation	Adiciona uma *annotation* em determinada classe, função ou método.
2	add file	Adiciona determinado arquivo.
3	add unit	Cria determinada classe, função ou método em determinado arquivo fonte.
4	remove @annotation	Remove determinada *annotation* de determinada classe, função ou método.
5	remove file	Remove determinado arquivo.
6	remove string	Remove determinada *string* (ou cadeia de *strings*) de determinada classe, função, método ou arquivo.
7	remove unit	Remove determinada classe, função ou método de um arquivo fonte.
8	replace @annotation	Substitui determinada *annotation* de determinada classe, função ou método.
9	replace file	Substitui determinado arquivo.
10	replace string	Substitui determinada *string* (ou cadeia de *strings*) em determinada classe, função, método ou arquivo.
11	replace unit	Substitui o corpo de determinada classe, função ou método.

Fonte: Do autor (2021).

3.4.2 Descrição das instruções de customização

Esta seção detalha cada uma das onze instruções incluindo sintaxe e exemplos práticos. Para cada instrução é apresentada:

- sua motivação: o que motivou a criação daquela instrução;
- sua sintaxe: sintaxe que deve ser adotada ao empregar a instrução de customização;
- um exemplo prático: uma situação real de emprego da instrução, com sua explicação e a construção da instrução correspondente.

Opcionalmente, para algumas instruções é apresentado um "exemplo motivador" que ilustra o emprego daquela instrução para as *features* idealizadas na Seção 3.1. Por fim, deve-se notar a ausência de uma instrução "**`add string`**". Tal instrução não foi criada pela necessidade de trabalhar sobre o código fonte de forma assertiva. A manipulação do código de forma puramente textual levou a inúmeros erros na etapa de levantamento das instruções. Buscou-se, portanto, realizar as intervenções por meio de árvores de sintaxe abstrata e a instrução deixou de ser necessária pois seu uso foi suprido por outras instruções.

3.4.2.1 `add @annotation`

Motivação: Necessidade de adicionar uma anotação ao projeto customizado que não existe no projeto original.

Sintaxe:

```
<file> add @<annotation'> [<json>] to <unit>
[before|after @<annotation"">]
```

Adiciona a anotação **`annotation'`** – opcionalmente com os atributos e valores informados no **`json`** – na classe, função ou método **`unit`** existente dentro do arquivo fonte **`file`**. Em caso da existência de múltiplas anotações na unidade de compilação **`unit`**, o mantenedor opcionalmente pode informar que deve ser adicionada antes ou após a anotação **`annotation"`** de forma a preservar a legibilidade e a organização do código.

Exemplo prático: Essa instrução permite customizações simples, mas que possuem impacto na lógica do sistema customizado. Por exemplo, (i) pode-se customizar o método

clonar_objeto para que, diferentemente do que ocorre no projeto original, seja realizado de forma atômica no projeto customizado. (ii) Pode-se, ainda, informar que se trata de um método de classe, destacando essa característica através de uma anotação que antecede a primeira.

(i) `file.ext add @atomic to ClasseAlvo::clonar_objeto`

(ii) `file.ext add @classmethod to ClasseAlvo::clonar_objeto before @atomic`

3.4.2.2 add file

Motivação: Necessidade de criar todo um artefato de software.

Sintaxe:

```
<file> add
```

Cria todo o artefato de software **file** obtendo-o a partir da estrutura de arquivos auxiliares.

Exemplo prático: Aplica-se a casos onde as customizações necessitam da adição de novos arquivos ao projeto customizado, inexistentes no projeto original. Por exemplo, (i) arquivos *html* para renderização de uma nova funcionalidade, inexistente no projeto original e (ii) arquivos binários utilizados por essas novas funcionalidades, tal como imagens.

(i) `relatorio_notas_pdf.html add`

(ii) `logo_alternativo.jpg add`

3.4.2.3 add unit

Motivação: Necessidade de adicionar uma classe, função ou método ao projeto customizado que não existe no projeto original.

Sintaxe:

```
<file> add <unit'> before|after <unit"">
```

Adiciona a classe, função ou método **unit'** ao arquivo fonte **file**. O mantenedor deve, obrigatoriamente, informar a unidade de compilação **unit"** que antecede ou sucede a unidade de compilação a ser criada (**before|after**). Esse recurso preserva a organização semântica das unidades de compilação dentro do arquivo fonte. A implementação completa da nova unidade de compilação **unit'** é obtida a partir do arquivo fonte local **file'**, obtido na estrutura de arquivos auxiliares locais.

Exemplo prático: Útil em customizações em nível de módulo de um sistema, onde a criação de novas estruturas faz-se necessária. Por exemplo:

(i) **file.ext add ServidorAfastado after Servidor**

(ii) **file.ext add print_small_details after print_details**

(iii) **file.ext add EntradaPermanente::get_itens before EntradaPermanente::get_material**

(i) uma nova classe que representa uma nova entidade no sistema; (ii) uma nova função que implementa um novo *controller* em atendimento a novas regras de negócio; e (iii) um novo método que implementa um relatório específico no projeto customizado.

Exemplo motivador: A Listagem 3.2 ilustra uso de **add unit** no arquivo de customização da *feature* F#2:

```
1 #feature 002_criarModeloRecepcionista
2 #description Cria um novo modelo de dados - Recepcionista - dentro
      do módulo de cadastros básicos, e registra este modelo no
     admin do Django
3 cadastros_basicos/models.py add Recepcionista after OfertaDeTurma
4 cadastros_basicos/admin.py add RecepcionistaAdmin after
     OfertaDeTurmaAdmin
```

Listagem 3.2 – Arquivo de customização F#2

Assim como as instruções **replace file** e **replace unit**, a instrução **add unit** utiliza um arquivo fonte **file'** para realizar as transformações necessárias. Esse arquivo fonte é obtido em uma estrutura de arquivos locais, criados pelo desenvolvedor para atender às necessidades de customização. Nesse caso, a classe, função ou método **unit'** é obtida

nesse arquivo e inserida no arquivo fonte **file** antes ou após a unidade de código **unit"** já existente no mesmo.

3.4.2.4 remove @annotation

Motivação: Necessidade de remover uma anotação do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove @<annotation> from <unit>
```

Realiza a ação inversa da instrução **add_annotation** com o mesmo impacto na lógica do sistema, removendo **annotation** da classe, função ou método **unit** existente no arquivo fonte **file**.

Exemplo prático: Customização em que um método que realiza transações no banco de dados é "atômico" no projeto original, mas que no projeto customizado é um simples método de acesso:

```
file.ext remove @atomic from ClasseAlvo::clonar_objeto
```

3.4.2.5 remove file

Motivação: Necessidade de remover todo um artefato de software.

Sintaxe:

```
<file> remove
```

Remove todo o artefato de software **file**.

Exemplo prático: Aplica-se a casos onde arquivos oriundos do projeto original não são necessários e, por decisão do mantenedor, devem ser excluídos da estrutura diretórios do projeto customizado. Por exemplo, arquivos *html* que renderizam informações inexistentes no projeto customizado.

```
relacao_projetos_derivados_conhecidos.html remove
```

3.4.2.6 `remove string`

Motivação: Necessidade de remover uma *string* (ou uma cadeia de *strings*) de uma classe, função, método ou arquivo do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove [from <unit>] <str>
```

Remove a *string* **`str`** – opcionalmente da classe, função ou método **`unit`** – existente no arquivo fonte **`file`**.

Exemplo prático: Instrução criada para atender a situações onde a exclusão de uma única *string* ou de uma cadeia de *strings* representa a customização necessária para a instituição usuária do sistema.

`file.ext remove from send_alerts` "*enviar_sms()*"

3.4.2.7 `remove unit`

Motivação: Necessidade de remover uma classe, uma função ou um método do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove <unit>
```

Remove a classe, função ou método **`unit`** do arquivo fonte **`file`**.

Exemplo prático: Instrução utilizada em situações onde é interessante remover toda a unidade de compilação do arquivo fonte. Por exemplo:

(i) **`file.ext remove ServidorLicenciado`**

(ii) **`file.ext remove print_full_details`**

(iii) **`file.ext remove EntradaPermanente::get_itens_ordenado`**

(i) remover toda uma classe desnecessária no projeto customizado; (ii) remover uma função que implementa uma ação que não pode ocorrer no projeto customizado; e (iii) remover um método sem uso.

3.4.2.8 `replace @annotation`

Motivação: Necessidade de substituir uma anotação no projeto customizado que existe no projeto original.

Sintaxe:

```
<file> replace @<annotation> [<json>] from <unit>
```

Substitui a anotação **`annotation`** – opcionalmente com os atributos e valores informados no **`json`** – na classe, função ou método **`unit`** existente dentro do arquivo fonte **`file`**.

Exemplo prático: Trata-se de um "açúcar sintático" para as instruções **`remove @annotation`** e **`add @annotation`**, já que a substituição de uma anotação nada mais é que a remoção da anotação pré-existente seguida da sua adição com nova definição. Por exemplo, pode-se customizar o método **`get_matriculas`** para que, diferentemente do que ocorre no projeto original, seja realizada por mais de um grupo de usuários, como a seguir:

```
file.ext replace @group_required {"list_group":  "grupo1, grupo2"}
                    from Classe::get_matriculas
```

3.4.2.9 `replace file`

Motivação: Necessidade de substituição de todo um artefato de software.

Sintaxe:

```
<file> replace
```

Substitui todo o artefato de software **`file`** pelo artefato de software com mesmo nome obtido na estrutura de arquivos locais, que são aqueles criados localmente pelo desenvolvedor do projeto customizado e que dão apoio às necessidades locais de customização.

Exemplo prático: Aplica-se a casos onde as customizações necessárias são extensas, quando comparadas à quantidade de linhas de código do arquivo fonte customizado, e não é possível ou

não é viável especificá-las com o uso das demais instruções. Por exemplo, (i) arquivos *html* com customizações extensas e de difícil representação por outras instruções e (ii) arquivos binários que não podem ser manipulados como texto, tal como imagens.

(i) **infoAluno.html replace**

(ii) **logo.jpg replace**

Exemplo motivador: A Listagem 3.3 ilustra uma utilização da instrução de customização **replace file** no arquivo de customização da *feature* F#1:

```
1 #feature 001-leiaute
2 #description Altera o logotipo da academia e cores de elementos em
      tela
3 cb/static/grappelli/images/back/logo.jpg replace
```

Listagem 3.3 – Fragmento do arquivo de customização F#1

Nesse exemplo, o arquivo **logo.jpg** será substituído pelo arquivo com mesmo nome disponível na estrutura de arquivos locais exatamente no mesmo diretório do projeto original: **cb/static/grappelli/images/back**.

3.4.2.10 replace string

Motivação: Necessidade de substituir uma *string* (ou uma cadeia de *strings*) de uma classe, função, método ou arquivo do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> replace [from <unit'>] <str'> by <str"> 
```

Substitui a *string* **str'** – opcionalmente da classe, função ou método **unit'** – existente no arquivo fonte **file** pela *string* **str"**.

Exemplo prático: Instrução que se aplica a situações onde determinada *string* (ou cadeia de *strings*) de uma classe, função ou método é única no corpo dessa estrutura e necessita ser substituída por outra. Aplicável também a cenários onde é preciso substituir um termo fixo por outro, onde ambos são simples mas ocorrem mais de uma vez ao longo de um arquivo. Por exemplo,

(i) substituição de uma cadeia de *strings* que representa a chamada de outro método ou função e (ii) substituição de um e-mail de contato espalhado pelo código fonte.

(i) **file.ext replace from send_alerts** "*enviar_sms()*" **by** "*enviar_email()*"

(ii) **file.ext replace** "*contato@a.br*" **by** "*contado@b.ar*"

Exemplo motivador: A Listagem 3.4 ilustra o uso de **replace string** no arquivo de customização da *feature* F#1:

```
1 cb/.../screen.css replace "#4fb2d3" by "#A10305"
2 cb/.../screen.css replace "#309bbf" by "#A10305"
3 Cb/.../contentbase.css replace "#309bbf" by "#A10305"
4 cb/.../rtl.css replace "#309bbf" by "#A10305"
```

Listagem 3.4 – Fragmento do arquivo de customização F#1

Como não é informado o parâmetro opcional **unit'**, a instrução buscará por qualquer ocorrência de **str'** dentro do arquivo fonte **file**, substituindo a mesma por **str"**.

3.4.2.11 replace unit

Motivação: Necessidade de substituição de toda a implementação de uma classe, função ou método no projeto customizado, i.e., desconsidera a implementação do projeto original.

Sintaxe:

```
<file> replace <unit'>
```

Substitui a implementação da classe, da função ou do método **unit'** do arquivo fonte **file**, buscando a nova implementação de **unit'** no arquivo local (aquele com mesmo nome disponível da estrutura de arquivos locais no mesmo diretório do projeto original).

Exemplo prático: Útil em situações onde a customização é localizada e com abrangência mais restrita, com relação ao módulo de um sistema como um todo. Por exemplo:

(i) **file.ext replace MatriculaForm**

(ii) **file.ext replace realizar_matricula**

(iii) **`file.ext replace Matricula::validar`**

(i) customização de uma classe responsável pela criação de objetos que renderizam um formulário, com características específicas da instituição usuária do sistema; (ii) customização de uma função que implementa uma regra de negócio que existe no projeto customizado porém diferente do projeto original; e (iii) customização de um método que existe no projeto original, porém com semântica diferente no projeto customizado.

Exemplo motivador: A Listagem 3.5 ilustra uma utilização da instrução de customização **`replace unit`** no arquivo de customização da *feature* F#3:

```
1 #feature 003-recepcionista-matricula
2 #description Altera a Matricula incluindo
3              recepcionista responsável
4 #dependsOn 002-recepcionista
5 alunos/models.py replace Matricula
```

Listagem 3.5 – Fragmento do arquivo de customização F#3

De forma análoga à instrução **`replace file`**, a nova implementação da classe `Matricula` será recuperada no arquivo local com mesmo nome.

3.5 Conflitos de customização

O objetivo da abordagem **não** é evitar conflitos de *merge* mas reduzi-los, provendo uma forma não *ad hoc* de conduzir projetos constantemente atualizados frente a um projeto original, onde *features* são modularizadas, documentadas e rastreáveis e podem ser reutilizadas. Logo, sua adoção pode gerar alguns tipos de *conflitos de customização* que devem ser considerados quando aplicada.

Nas seções seguintes, são apresentados tipos de *conflitos de customização* agrupados em função do impacto que podem ocasionar na execução da ferramenta de customização. Ou seja, os tipos de conflitos não são agrupados em função do impacto no código resultante, mas pela forma como podem ser identificados ao executar a ferramenta. Para cada um dos tipos de conflito são apresentados:

- cenário: situação em que o conflito pode ocorrer;

- exemplo: um exemplo prático que ilustra a ocorrência do conflito;
- instruções afetadas: a descrição de todas as instruções que estão suscetíveis àquele tipo de conflito.

Ao descrever cada instrução afetada, sugere-se (i) uma ação necessária e (ii) possíveis soluções para o conflito. Todos os conflitos de customização necessitam de análise por parte do desenvolvedor.

3.5.1 Customização em arquivo modificado na origem com impacto na execução da ferramenta

3.5.1.1 Cenário

Um arquivo fonte para o qual existe customização local foi alterado no projeto original e essa alteração resultou em impacto negativo na execução da ferramenta de customização. Em outras palavras, um código fonte incorreto foi obtido ou nem mesmo foi gerado um código resultante em função de um erro de execução da ferramenta.

3.5.1.2 Exemplo

Uma customização que substitui o método **`foo`** do projeto original. Caso esse método seja excluído no projeto original, a abordagem falhará alertando o ocorrido.

3.5.1.3 Instruções afetadas

`add @annotation`: a adição de uma anotação a uma unidade de código do projeto original falhou. *Ação necessária*: A customização deve ser revisada com relação à anotação a ser adicionada, à unidade de código de destino e à anotação de referência de posicionamento, quando informada. *Possíveis soluções*: A customização pode ser:

- descartada: a anotação pode ter sido incluída no projeto original, fazendo com que a customização local seja desnecessária. A unidade de código de destino também pode ter deixado de existir, dispensando assim a adição da anotação;
- modificada: a unidade de código de destino pode ter sido renomeada, fazendo necessária que essa situação seja refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou

com o apoio ferramental, quando disponível. Pode ocorrer ainda mudanças com relação à anotação de referência de localização, quando informada, que pode não mais existir no arquivo do projeto original. Nesse caso, a referência de localização deve ser alterada, ou mesmo dispensada se não for mais necessária.

`add unit`: a adição de uma nova unidade de código ao código fonte do projeto original falhou. *Ação necessária*: A customização deve ser revisada com relação à unidade de código a ser adicionada e à unidade de código utilizada como referência de posicionamento. *Possíveis soluções*: A customização pode ser:

- descartada: pode ter ocorrido uma reformulação mais abrangente do código fonte do módulo ao qual o arquivo fonte pertence e aquela unidade de código a ser adicionada não se faz mais necessária. Tal unidade de código pode ainda ter sido adicionada pelo projeto original, fazendo com que a customização local seja desnecessária;
- modificada: a unidade de código utilizada como referência de posicionamento pode ter sido renomeada e essa situação deve ser refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

`remove @annotation`: a remoção de uma anotação de determinada unidade de código do projeto original falhou. *Ação necessária*: A customização deve ser revisada com relação à anotação que deve ser removida e à unidade de código a qual ela pertence. *Possíveis soluções*: A customização pode ser:

- descartada: a anotação pode ter sido removida no projeto original, dispensando assim a customização local. Pode ainda ter ocorrido uma reformulação mais abrangente do código fonte do módulo ao qual o arquivo fonte pertence e aquela unidade de código a qual a anotação pertence não existe mais;
- modificada: a unidade de código a qual a anotação pertence pode ter sido renomeada e essa situação deve ser refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

`remove string`: a *string* ou cadeia de *strings* a serem removidas não foram mais encontradas no arquivo fonte do projeto original, opcionalmente na classe, função ou método

informado. *Ação necessária*: A customização deve ser revisada com relação à *string* ou cadeia de *strings* e à unidade de código, se essa tiver sido informada. *Possíveis soluções*: A customização pode ser:

- descartada: a *string* ou cadeia de *strings* pode ter sido excluída do projeto original, fazendo com que a customização local seja desnecessária. Pode ainda ter ocorrido a exclusão da unidade de código a qual a *string* ou cadeia de *strings* pertence, quando essa é informada;

- modificada: pequenas alterações na *string* ou cadeia de *strings* podem fazer com que a customização local falhe. Logo, faz-se necessário corrigir a instrução de customização. Pode ser ainda que a unidade de código, quando informada, tenha sido renomeada, fazendo necessário que essa situação seja refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

`remove unit`: a remoção de uma unidade de código do projeto original falhou. *Ação necessária*: A customização deve ser revisada com relação à unidade de código a qual ela pertence. *Possíveis soluções*: A customização pode ser:

- descartada: a unidade de código pode ter sido removida no projeto original, dispensando assim a customização local;

- modificada: a unidade de código pode ter sido renomeada no projeto original, fazendo com que seja necessário renomeá-la também no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

`replace @annotation`: diferentemente de outras instruções do tipo *replace* que possuem implementações específicas, a substituição de uma anotação nada mais é que sua remoção seguida de sua adição. Logo, todas as situações apresentadas para **`remove @annotation`** e para **`add @annotation`** aplicam-se a essa instrução.

`replace string`: a *string* ou cadeia de *strings* a ser substituída não foi mais encontrada no arquivo fonte do projeto original, opcionalmente na classe, função ou método informado. *Ação necessária*: A customização deve ser revisada com relação à *string* ou cadeia de *strings* e à unidade de código, se essa tiver sido informada. *Possíveis soluções*: A customização pode ser:

- descartada: a *string* ou cadeia de *strings* pode ter sido substituída pelo padrão utilizado localmente, ou seja, o projeto original assumiu a codificação local. Logo, a customização local deixa de ser necessária. Pode ainda ter deixado de existir, dispensando sua substituição;

- modificada: pequenas alterações na *string* ou cadeia de *strings* podem fazer com que a customização local falhe. Logo, faz-se necessário corrigir a instrução de customização. Pode ser ainda que a unidade de código, quando informada, tenha sido renomeada, fazendo necessário que essa situação seja refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

`replace unit`: a unidade de código a ser substituída não existe mais no arquivo fonte do projeto original. *Ação necessária*: A customização deve ser revisada com relação à unidade de código em questão. *Possíveis soluções*: A customização pode ser:

- descartada: pode ter ocorrido uma reformulação mais abrangente do código fonte do módulo ao qual o arquivo fonte pertence e aquela unidade de código não se faz mais necessária;

- modificada: a unidade de código pode ter sido renomeada e essa situação deve ser refletida no arquivo de customização. Importante observar que uma unidade renomeada é assim identificada através da inspeção manual do desenvolvedor ou com o apoio ferramental, quando disponível.

3.5.1.4 Instruções não afetadas

Conflitos relacionados a arquivos modificados na origem e com impacto na execução da ferramenta não foram identificados para as seguintes instruções: (i) **`add file`**, (ii) **`remove file`** e (iii) **`replace file`**.

3.5.2 Customização em arquivo modificado na origem sem impacto na execução da ferramenta

3.5.2.1 Cenário

Um arquivo fonte para o qual existe customização local foi alterado no projeto original e essa alteração não impactou na execução da ferramenta de customização. Em outras palavras,

um código fonte aparentemente correto foi obtido. Contudo, em determinadas situações, erros de customização podem ter sido gerados, imperceptíveis em um primeiro momento de análise.

3.5.2.2 Exemplo

Uma customização que remove uma *string* do projeto original. Todas as novas *strings* adicionadas no projeto original serão removidas por tal customização, inclusive aquelas adicionadas ao projeto original após a implementação dessa customização, o que nem sempre pode ser o desejado.

3.5.2.3 Instruções afetadas

`add unit`: a unidade de código a ser incluída pela customização local passou a existir no código fonte do projeto original. *Ação necessária*: A customização deve ser revisada com relação à unidade de código em questão, com ênfase nas diferenças entre a implementação do projeto original e a implementação do projeto customizado. *Possíveis soluções*: A customização pode ser:

- descartada: caso avalie-se que a implementação do projeto original atende as necessidades locais, mesmo existindo diferenças na implementação;
- modificada: se a implementação do projeto original não atender às necessidades locais, a instrução deverá ser substituída por outra que se adeque melhor a essas necessidades, como **`replace unit`** que substituirá a implementação original pela implementação local.

`remove string`: a *string* ou cadeia de *strings* a ser removida passou a ocorrer em novos trechos do arquivo fonte do projeto original, opcionalmente na classe, função ou método informado, quando comparada à uma versão anterior do arquivo. *Ação necessária*: A customização deve ser revisada com relação à *string* ou cadeia de *strings* e à unidade de código, se essa tiver sido informada, com ênfase nas novas ocorrências daquele padrão. *Possíveis soluções*: A customização pode ser:

- descartada: caso a remoção da *string* ou cadeia de *strings* não seja mais possível, porque a sua remoção nos novos trechos representaria erro no código fonte resultante;
- mantida: caso seja possível e logicamente correto remover também as novas ocorrências da *string* ou cadeia de *strings* em questão.

`remove unit`: a unidade de código a ser removida pela customização local passou a ser referenciada (chamada) em novos trechos do projeto original. *Ação necessária*: A customização deve ser revisada com relação às chamadas à unidade de código em questão. *Possíveis soluções*: A ferramenta poderá analisar se existem novas referências (chamadas) à unidade de código a ser removida em novos trechos do arquivo modificado do projeto original. Em caso positivo a instrução poderá ser:

- descartada: caso a remoção da unidade de código não seja mais possível, porque removê-la representaria erro no código fonte resultante;
- mantida: caso seja possível, e logicamente correto, remover também as novas referências (chamadas) à unidade de código em questão, para isso fazendo uso de outras instruções de customização.

`replace @annotation`: a anotação em questão pode ter assumido a mesma definição no arquivo fonte do projeto original. *Ação necessária*: A customização deve ser revisada com relação à anotação a ser alterada. *Possíveis soluções*: A customização pode ser:

- descartada: caso a anotação tenha assumida a mesma definição no código fonte do projeto original, fazendo com que essa customização local seja desnecessária;
- mantida: caso a anotação tenha assumido nova definição, mas diferente daquela necessária ao projeto customizado.

`replace string`: o arquivo fonte onde a *string* ou cadeia de *strings* se encontram foi alterado, opcionalmente na classe, função ou método informado. *Ação necessária*: A customização deve ser revisada com relação à *string* ou cadeia de *strings* e à unidade de código, se essa tiver sido informada. *Possíveis soluções*: A ferramenta de customização poderá analisar se a quantidade de ocorrências daquela *string* ou cadeia de *strings* aumentou no arquivo alterado na origem, opcionalmente dentro de determinada classe, método ou função, com relação ao que existia anteriormente. Caso existam novas ocorrências, a ferramenta deverá sugerir a revisão da customização que poderá ser:

- descartada: o fato da *string* ou cadeia de *strings* terem sido utilizadas em novos trechos de código torna inviável a utilização dessa instrução de customização, forçando seu descarte e opção por outra que se adeque melhor às necessidades de customização local;

- modificada: a inclusão da unidade de código onde a *string* ou cadeia de *strings* deve ser localizada, para situações onde essa não era utilizada, pode contornar o problema do surgimento de novas ocorrências dentro do mesmo arquivo fonte, mas em outras unidades de código;

- mantida: caso a substituição das novas ocorrências da *string* ou cadeia de *strings* também representem a customização local necessária.

`replace unit`: a unidade de código a ser substituída foi alterada no arquivo fonte do projeto original. *Ação necessária*: A customização deve ser revisada com relação à unidade de código em questão, com ênfase nas alterações realizadas pela customização local e os pontos alterados do projeto original. *Possíveis soluções*: A ferramenta de customização poderá analisar se unidades de código customizadas localmente foram alteradas no projeto original. Caso tenham sido, a ferramenta deverá sugerir a revisão da customização que poderá ser:

- descartada: caso as alterações recentes do projeto original reflitam a necessidade local, mesmo que existam pequenas diferenças de implementação;

- mantida: caso as alterações recentes do projeto original continuem a não atender o que se espera do código local customizado.

3.5.2.4 Instruções não afetadas

Conflitos relacionados a arquivos modificados na origem e sem impacto na execução da ferramenta não foram identificados para as seguintes instruções: (i) **`add @annotation`**, (ii) **`add file`**, (iii) **`remove @annotation`**, (iv) **`remove file`** e (v) **`replace file`**.

3.5.3 Customização em arquivo excluído na origem

3.5.3.1 Cenário

Um arquivo fonte para o qual existe customização local foi excluído no projeto original.

3.5.3.2 Exemplo

Uma customização que adiciona um método em uma classe implementada em determinado arquivo, o qual não existe mais no projeto original.

3.5.3.3 Instruções afetadas

Todas as instruções são afetadas. *Ação necessária*: Todas as customizações que envolvem o arquivo fonte em questão devem ser revisadas.

Possíveis soluções: Deve-se buscar pelo arquivo fonte no projeto original para descobrir se ele foi renomeado ou foi movido para outra pasta na estrutura de diretórios daquele projeto. Caso seja possível localizá-lo, deve-se alterar todas as instruções de customização que fazem uso desse arquivo, apontando para o novo arquivo e/ou diretório. Caso não seja possível localizá-lo, deve-se rever a aplicabilidade e necessidade de cada instrução de customização, uma a uma.

3.6 Considerações Finais

Neste capítulo, foi apresentada a abordagem de customização, ponto central desta dissertação de mestrado. Por meio de um exemplo motivador, ilustrou-se sua aplicação e definiu-se o conceito de *feature*: funcionalidade de um sistema de software que resolve algum problema do mundo real. Três *features* foram idealizadas para um pequeno sistema de academia e exploradas ao longo do capítulo para ilustrar as diretrizes e instruções criadas na DSL proposta.

Um potencial *workflow* para aplicação da abordagem em cenário real de desenvolvimento foi apresentado. Os passos necessários para que o desenvolvedor atenda uma solicitação de desenvolvimento seguindo a aplicação da abordagem foram descritos. Tratou-se também da necessidade de versionamento tanto de arquivos de customização quanto de arquivos locais, necessários no processo de customização.

O levantamento empírico das instruções de customização ocorreu por meio de estudo realizado em um sistema real de software. Simulações de conflitos de mesclagem de código foram realizadas para uma data arbitrária desse sistema e os respectivos conflitos foram analisados individualmente de modo a idealizar, generalizar e especificar um conjunto de onze instruções em DSL. A motivação e sintaxe de cada instrução foi apresentada. Exemplos práticos e exemplos motivadores, dentro do contexto das *features* apresentadas para o sistema motivador, foram amplamente explorados para as instruções criadas.

A abordagem de customização não objetiva acabar com os conflitos de mesclagem, mas diminuir sua ocorrência. Os conflitos decorrentes da mesclagem do código fonte do projeto customizado com o código fonte do projeto original deixam de ocorrer uma vez que essa mes-

clagem não é mais necessária. Contudo, outros tipos de conflitos, chamados de conflitos de customização, passam a existir. Esses conflitos foram apresentados em detalhes com a descrição dos cenários em que podem ocorrer, exemplos práticos e instruções afetadas, ações necessárias e possíveis soluções.

Definida a abordagem de customização e suas instruções, faz-se necessário aplicar à cópia do código fonte do projeto original as transformações descritas em DSL, obtendo assim o projeto customizado. O próximo capítulo tratará da ferramenta de customização responsável por essa etapa dentro da abordagem proposta nesta dissertação de mestrado.

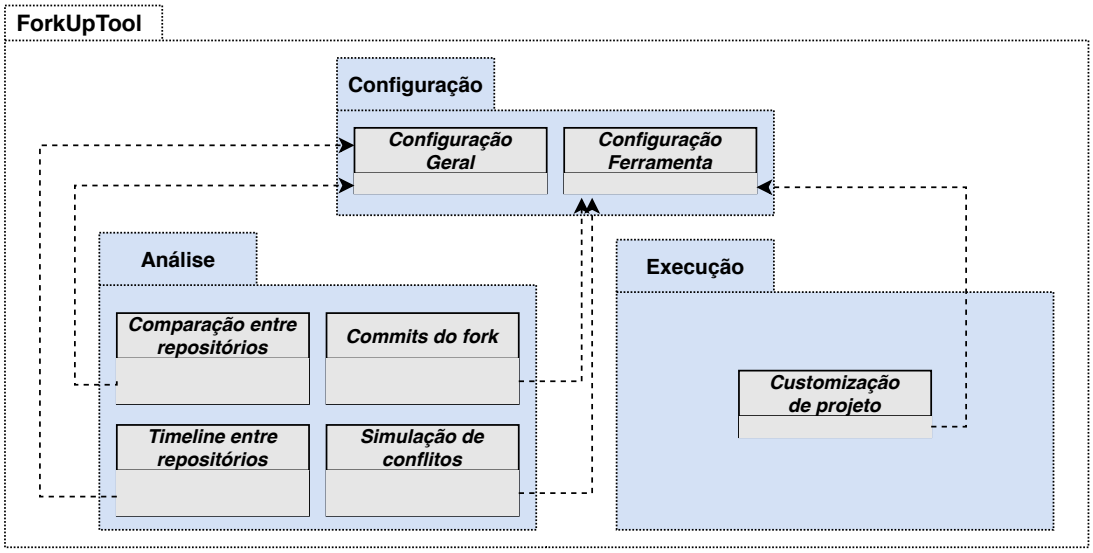
4 FERRAMENTA DE CUSTOMIZAÇÃO

Um protótipo da ferramenta de customização denominado `ForkUpTool` foi desenvolvido para interpretar todas as instruções da linguagem de customização (Tabela 3.1). O protótipo foi desenvolvido em Python com uso do *framework* `Django` e é capaz de trabalhar apenas com sistemas desenvolvidos nessa linguagem, sendo possível ainda pequenas intervenções em arquivos de linguagens de marcação (JavaScript, CSS, HTML, etc.). A ferramenta – além de sua documentação e vídeos ilustrativos – estão publicamente disponíveis em:

<https://github.com/PqES/forkuptool>

A `ForkUpTool` segue uma arquitetura organizada em três módulos, tal como ilustra a Figura 4.1:

Figura 4.1 – Módulos da `ForkUpTool`.



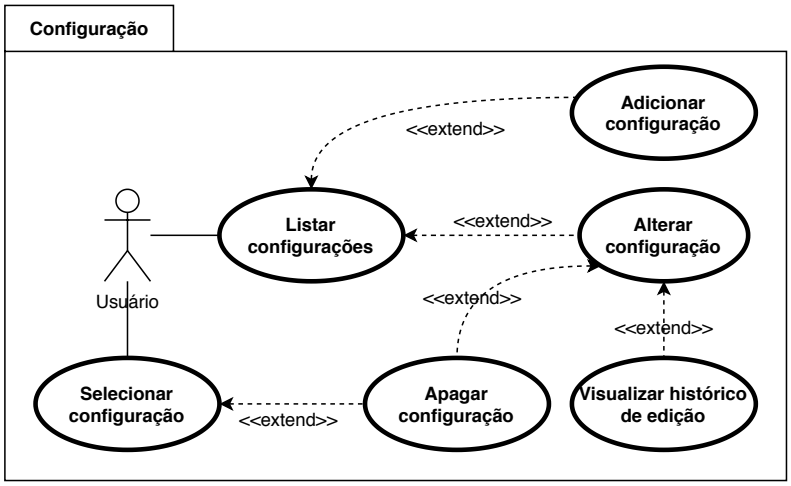
Fonte: Do autor (2021).

O módulo análise e o módulo execução fazem uso das informações persistidas pelo módulo configuração. Detalhes de cada módulo são apresentados nas próximas seções.

4.1 Módulo Configuração

É o módulo responsável pela persistência de entidades de dados que representam as configurações do sistema – **`Configuração Geral`** e **`Configuração Ferramenta`**. Um esquema geral dos casos de uso implementados nesse módulo para cada uma das entidades é apresentado na Figura 4.2.

Figura 4.2 – Casos de uso do módulo Configuração.



Fonte: Do autor (2021).

Os casos de uso da Figura 4.2 representam as operações básicas de um CRUD. É possível selecionar as configurações existentes (Selecionar configuração), persistir novas informações na base de dados da ferramenta (Adicionar configuração), recuperar informações para cada uma das entidades de configuração (Listar configurações e Visualizar histórico de edição), atualizar informações anteriormente persistidas (Alterar configuração) e até mesmo apagar configurações desnecessárias (Apagar configuração).

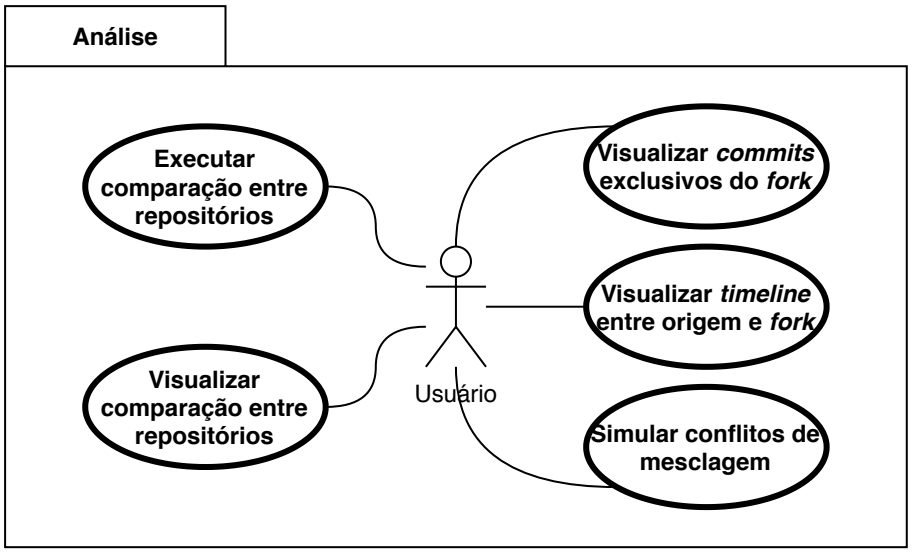
Nesse módulo, são configurados dados sobre os repositórios origem e seu *fork* onde as informações mais importantes são os caminhos físicos de cada repositório. A implementação dos casos de uso se dá com a interface de administração automática do `Django`, que provê todo o código necessário, tanto do *backend* quanto do *frontend*.

Os casos de uso se relacionam seguindo um padrão estipulado pelo `Django`. Apesar do recurso de administração automática permitir certas customizações, como a adição de novos casos de uso ou supressão daqueles pré existentes, adotou-se por conveniência o padrão do *framework* pois são acessórios ao objetivo principal da ferramenta.

4.2 Módulo Análise

É o módulo que permite realizar a comparação automática de dois repositórios de software, sendo o primeiro o projeto original e o segundo um *fork* desse projeto, destacando diferenças de forma visual e permitindo o registro de observações, quando necessário. A Figura 4.3 apresenta os casos de uso implementados nesse módulo.

Figura 4.3 – Casos de uso do módulo Análise.



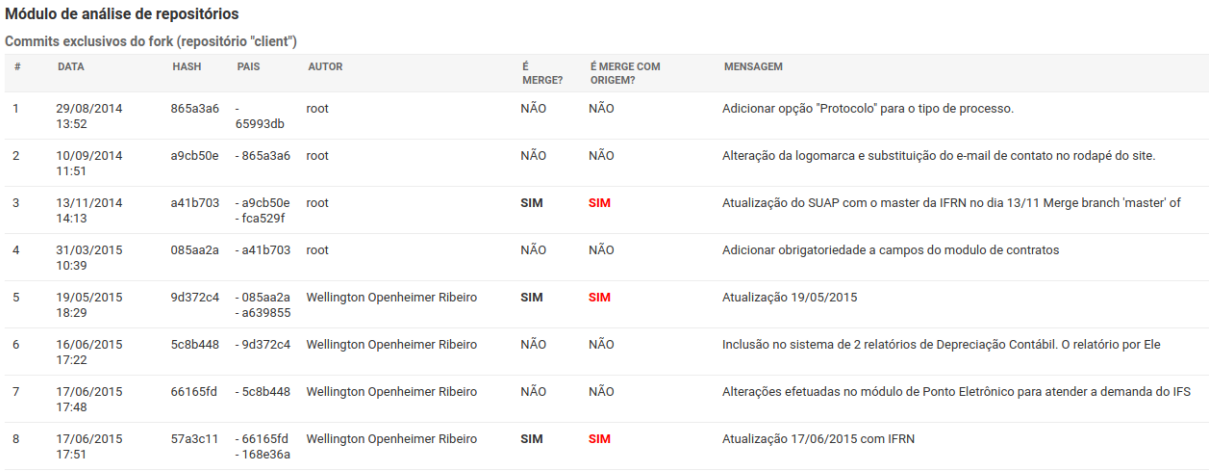
Fonte: Do autor (2021).

A Figura 4.3 apresenta os cinco casos de uso disponíveis no módulo. Dois deles se relacionam: "Executar comparação entre repositórios" e "Visualizar comparação entre repositórios". O primeiro busca diferenças entre repositório origem e repositório customizado. O código fonte do repositório origem é tomado como base e é comparado com o código fonte do repositório customizado, buscando o que há de diferente no segundo repositório, persistindo essas diferenças na base de dados. O segundo caso de uso visualiza os resultados dessa comparação e permite a persistência de observações, eventualmente necessárias no processo de análise.

Os outros três casos de uso do módulo atuam de forma isolada. O primeiro – "Visualizar *commits* exclusivos do *fork*" – varre o histórico de *commits* dos projetos original e customizado, apontando aqueles exclusivos do repositório customizado representando as customizações locais realizadas no projeto customizado ao longo do tempo. O segundo caso de uso – "Visualizar *timeline* entre origem e *fork*" – assemelha-se com o anterior, mas destaca os *commits* iguais e diferentes entre o projeto original e projeto customizado de forma visual, dentro de uma linha do tempo. Por fim, o terceiro caso de uso – "Simular conflitos de mesclagem" – permite a simulação dos conflitos de mesclagem entre repositório origem e seu *fork* quando ocorreram mesclagens de atualização no segundo repositório. Para isso, a ferramenta "navega" no histórico de *commits* do repositório customizado, buscando um *commit* de mesclagem com a origem. Ao encontrá-lo, a ferramenta faz o *checkout* dos dois projetos para os respectivos *commits* que deram origem ao *commit* de mesclagem e mescla os códigos, apurando dados de conflitos.

O objetivo geral desse módulo é avaliar se as transformações realizadas com base nas instruções de customização foram corretamente aplicadas. Além disso, extrai informações entre um *fork* de um projeto e o projeto original, destacando *commits* locais e os *commits* de *merge* (Figura 4.4). Em 19/05/2015, por exemplo, foi realizado um *commit* no projeto customizado do tipo *merge* que se tratou de uma mesclagem de código com o projeto original.

Figura 4.4 – *Commits* exclusivos do *fork*.



Módulo de análise de repositórios

Commits exclusivos do fork (repositório "client")

#	DATA	HASH	PAIS	AUTOR	É MERGE?	É MERGE COM ORIGEM?	MENSAGEM
1	29/08/2014 13:52	865a3a6	- 65993db	root	NÃO	NÃO	Adicionar opção "Protocolo" para o tipo de processo.
2	10/09/2014 11:51	a9cb50e	- 865a3a6	root	NÃO	NÃO	Alteração da logomarca e substituição do e-mail de contato no rodapé do site.
3	13/11/2014 14:13	a41b703	- a9cb50e - fca529f	root	SIM	SIM	Atualização do SUAP com o master da IFRN no dia 13/11 Merge branch 'master' of
4	31/03/2015 10:39	085aa2a	- a41b703	root	NÃO	NÃO	Adicionar obrigatoriedade a campos do modulo de contratos
5	19/05/2015 18:29	9d372c4	- 085aa2a - a639855	Wellington Openheimer Ribeiro	SIM	SIM	Atualização 19/05/2015
6	16/06/2015 17:22	5c8b448	- 9d372c4	Wellington Openheimer Ribeiro	NÃO	NÃO	Inclusão no sistema de 2 relatórios de Depreciação Contábil. O relatório por Ele
7	17/06/2015 17:48	66165fd	- 5c8b448	Wellington Openheimer Ribeiro	NÃO	NÃO	Alterações efetuadas no módulo de Ponto Eletrônico para atender a demanda do IFS
8	17/06/2015 17:51	57a3c11	- 66165fd - 168e36a	Wellington Openheimer Ribeiro	SIM	SIM	Atualização 17/06/2015 com IFRN

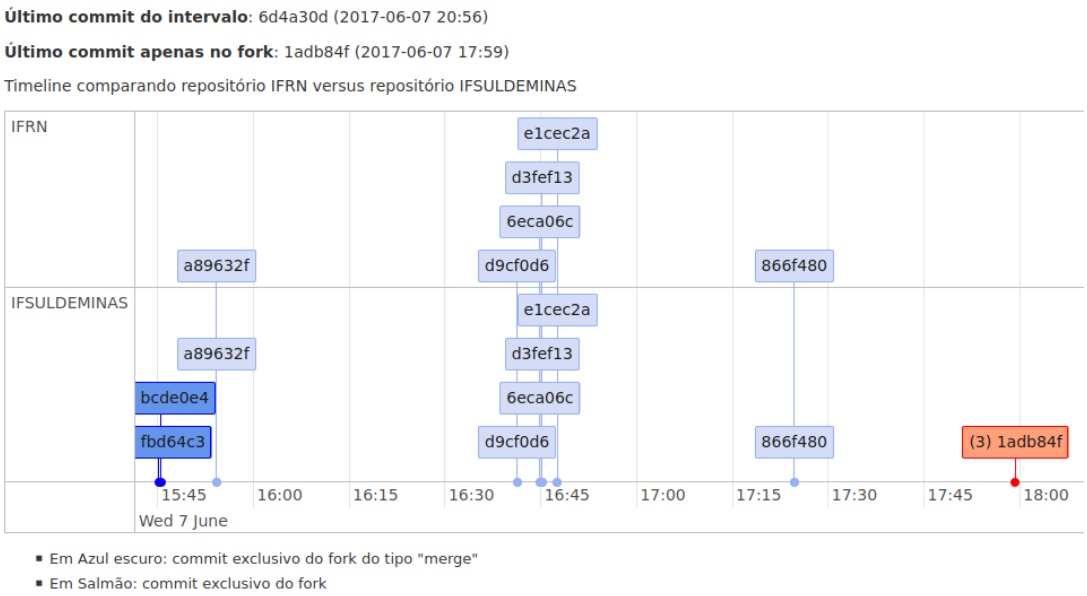
Fonte: Do autor (2021).

É possível ainda realizar uma análise gráfica da diferença entre *commits* utilizando uma *timeline* (Figura 4.5). É possível, por exemplo, observar em 07/06/2017 a existência de dois *commits* no projeto customizado que não existem no projeto original mas não são *commits* de mesclagem (bcde0e4 e fbd64c3). Existe ainda, no mesmo dia, um outro *commit* exclusivo do repositório customizado que é um *commit* de mesclagem com o projeto original (1adb84f).

O módulo Análise permite também a simulação de conflitos de *merge* entre um *fork* e sua origem dentro de um intervalo de datas, levantando de forma automática dados como quantidade de arquivos em conflito, quantidade de trechos e quantidade de linhas conflitantes. Os conflitos são simulados da seguinte maneira:

- toma-se como base a data inicial informada e busca-se no projeto customizado um *commit* do tipo *merge* que possua como “pais” um *commit* exclusivo do projeto customizado (a.k.a., *commit customizado*) e um *commit* oriundo do projeto origem (a.k.a., *commit origem*);
- realiza-se o *checkout* do projeto customizado para o *commit customizado*;
- realiza-se o *checkout* do projeto origem para o *commit origem*;

Figura 4.5 – *Timeline* entre o *fork* e sua origem.



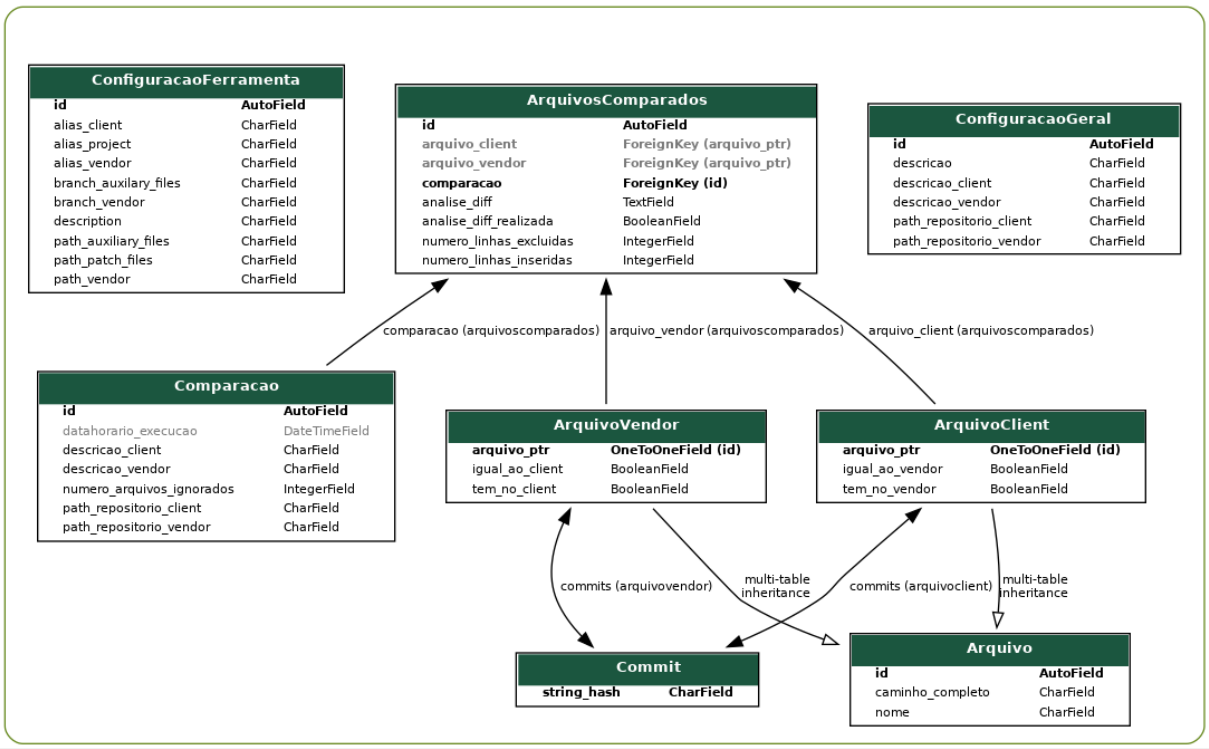
Fonte: Do autor (2021).

- realiza-se o *merge* entre projeto customizado e projeto origem, nessa ordem, obtendo dados simulados com relação a conflitos que ocorreram no passado;
- após apurados os dados de conflitos, a operação de *merge* é anulada, realiza-se o *checkout* dos dois projetos para o estado inicial e busca-se a próxima data onde ocorreu uma atualização do código.

A implementação dessas funções envolve a persistência de diferentes modelos de dados. Diferentemente do que ocorre no módulo Configuração, onde a implementação é automática via recurso de administração automática do `Django`, as entidades de dados são implementadas por meio de classes criadas para esse fim (Figura 4.6).

Na Figura 4.6, são apresentadas oito classes no total, das quais seis são implementadas no módulo Análise. As classes **`ArquivoVendor`** e **`ArquivoClient`** herdam da classe **`Arquivo`**. Um objeto da classe **`Commit`** pode estar relacionado a vários objetos **`ArquivoVendor`** ou **`ArquivoClient`**. A classe **`Comparacao`** representa uma comparação entre repositórios, que envolve vários objetos **`ArquivosComparados`**, que por sua vez se relacionam com um objeto **`ArquivoVendor`** e um objeto **`ArquivoClient`**. As duas classes não citadas – **`ConfiguracaoFerramenta`** e **`ConfiguracaoGeral`** – são aquelas apresentadas na Seção 4.1.

Figura 4.6 – Diagrama de classes da `ForkUpTool`.

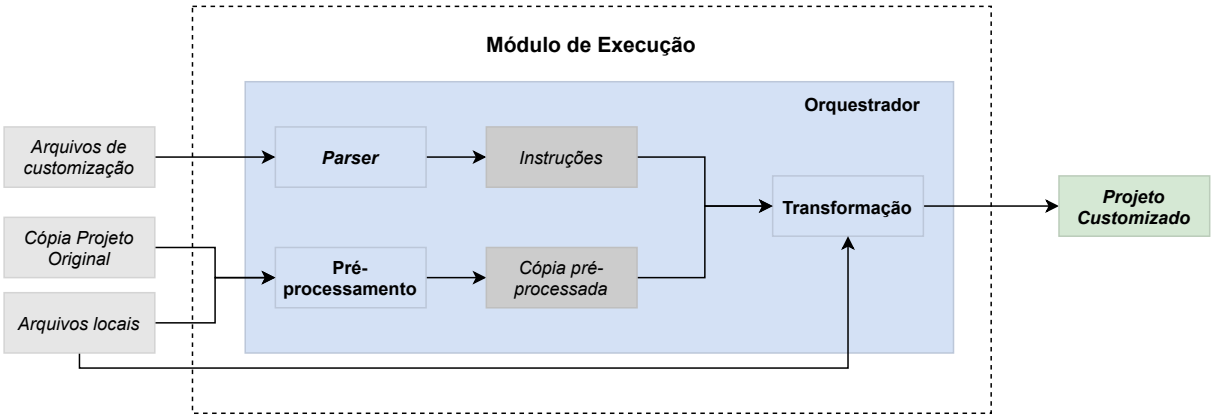


Fonte: Do autor (2021).

4.3 Módulo Execução

É o principal módulo do sistema que efetivamente realiza as customizações sobre a cópia do código fonte do sistema original criando o sistema customizado. Essas customizações são realizadas em quatro etapas que possuem como entrada a cópia do projeto original, os arquivos de customização e os arquivos locais. Uma visão geral das funções realizadas nesse módulo é apresentada na Figura 4.7.

Figura 4.7 – Visão geral do módulo Execução.



Fonte: Do autor (2021).

Parser: realiza o *parser* de instruções de customização. Recebe os arquivos de customização de forma textual. Lê os arquivos de forma sequencial, um a um. Para cada arquivo, realiza a análise linha a linha, interpretando a instrução, validando sua estrutura e gerando alertas de erro caso uma instrução esteja mal formulada;

Pré-processamento: trabalha sobre cada um dos arquivos a serem customizados, limitando-se na versão atual do protótipo a arquivos Python, assim como ocorre com as demais funções do Módulo Execução. Obtém os cabeçalhos de *imports* da cópia do arquivo do projeto original e também do arquivo local, se esse existir. Reordena os *imports*, elimina as duplicidades e inclui novos *imports* eventualmente necessários de acordo com as customizações. É importante observar que o pré-processamento pode mudar em função da linguagem de programação manipulada de tal modo a se adequar às particularidades de cada uma;

Transformação: realiza a transformação do código. Recebe cada instrução gerada pelo *parser* e atua sobre o código fonte de acordo com aquilo que a instrução determina. Abre a cópia do arquivo do projeto original especificada pela instrução, monta uma AST para o código descrito no arquivo, busca na AST o nó correspondente à unidade de código a ser manipulada e aplica as transformações necessárias. Busca também pelo arquivo local, quando necessário, transformando-o em uma AST e extraindo o nó correspondente à unidade de código desejada, para instruções onde essa ação faz-se necessária. Faz uso predominantemente da biblioteca `ast` da linguagem Python. Para instruções baseadas em busca e substituição de padrões (**`replace string`** e **`remove string`**), faz uso de expressões regulares, permitindo não só a maniputação de arquivos em Python, mas também de arquivos JavaScript, CSS, HTML, etc., já que os arquivos são tratados de forma textual. Duas instruções de granularidade alta (**`add file`** e **`replace file`**) fazem uso apenas de funções de manipulação de arquivos definidas em Python e aplicáveis a quaisquer tipos de arquivos, inclusive arquivos binários que ficam fora do escopo de outras instruções;

Orquestrador: é a etapa que orquestra todo o processo de customização. Coordena a execução das etapas realizadas pelo *parser* de instruções de customização, pelo pré-processamento e pela transformação do código. É responsável por acumular resultados detalhados das diferentes

etapas de execução, divididas por arquivo de customização e por instrução de customização, apresentando detalhes ao usuário da ferramenta ao final da execução.

4.4 Considerações Finais

Neste capítulo, foi apresentada a ferramenta que interpreta todas as instruções da linguagem de customização proposta, manipulando o código fonte da cópia do projeto original, de tal modo a torná-lo o sistema customizado. Tal ferramenta foi organizada em três módulos: (i) Configuração, responsável pela persistência de dados de configuração do ambiente e de execução da ferramenta; (ii) Análise, onde estão presentes diversas funções que objetivam faciltar a análise e comparação entre repositório origem e customizado, de modo a verificar se as customizações foram corretamente aplicadas; e (iii) Execução, onde efetivamente é realizada a manipulação do código fonte do sistema original, transformando-o no sistema customizado.

O módulo Configuração persiste em base da dados informações necessárias aos demais módulos, como caminhos físicos dos repositórios origem e customizado. Fez-se uso da interface de administração automática do `Django`, *framework* utilizado no desenvolvimento dessa ferramenta, para prover todas as operações básicas de um CRUD necessárias ao cadastro e manutenção das configurações.

O módulo Análise permite realizar a comparação automática de dois repositórios de software. O objetivo principal é verificar se as transformações foram corretamente aplicadas no projeto customizado. Além disso, é possível visualizar de diferentes formas as diferenças entre um repositório origem e seu *fork*, seja em uma *timeline* ou em uma listagem de *commits*. É possível realizar a simulação de conflitos de mesclagem entre dois repositórios de modo a apurar dados como quantidade de arquivos em conflito, quantidade de trechos e quantidade de linhas conflitantes.

O terceiro e principal módulo do sistema – Execução – é o responsável por aplicar as transformações necessárias na cópia do projeto original. Quatro funções são responsáveis por essa tarefa: (i) o *parser* interpreta as instruções de customização a partir dos arquivos de customização; (ii) o pré-processamento manipula os arquivos customizados, com foco em rotinas de reorganização; (iii) a transformação recebe a instrução do *parser* e cria as árvores de sintaxe abstrata para cada arquivo a ser manipulado, buscando unidades de código e atuando sobre o código conforme a instrução recebida; e (iv) o orquestrador coordena a execução das demais funções e reporta ao usuário da ferramenta os resultados finais de execução.

5 AVALIAÇÃO HISTÓRICA

Este capítulo avalia a abordagem de customização proposta aplicada ao longo de um período histórico de um projeto de software frequentemente atualizado frente ao seu projeto original. Os objetivos da avaliação são:

1 - corroborar a relevância do tema com base em dados históricos de atualização de um sistema real de software, analisando sua frequência de atualização;

2 - mostrar que a abordagem é aplicável a um cenário real;

3 - mostrar que a abordagem evita conflitos de mesclagem de código.

5.1 Questões de pesquisa

A avaliação proposta responde às seguintes questões de pesquisa:

QP #1: Com que frequência o projeto customizado avaliado se atualiza frente ao seu projeto original?

Justificativa: realizar uma análise histórica que evidencie a frequência de atualização de um projeto customizado frente à sua origem.

QP #2: A abordagem é aplicável em um cenário real?

Justificativa: realizar uma análise histórica que evidencie a aplicabilidade da abordagem a um cenário real de software.

QP #3: Existem indícios de que a abordagem evita conflitos?

Justificativa: realizar uma análise histórica que evidencie a ocorrência de conflitos de mesclagem quando a abordagem é empregada.

5.2 Sistema alvo

O sistema real de software escolhido para avaliação foi o SUAP[1], apresentado previamente na Seção 3.4.1. Um período de abrangência de quatro anos foi definido para análise, com início no momento de criação do *fork* do IFSULDEMINAS (agosto/2014) até o *merge* de

[1] Os códigos fontes do sistema SUAP e os arquivos de customização desse sistema não podem ser publicados por questões de confidencialidade.

atualização do mês de agosto/2018. É importante mencionar que esse período é *completamente* diferente (i.e., não inclui) o período utilizado no estudo prévio para levantamento de potenciais instruções (ver Seção 3.4.1).

No período avaliado, ocorreram 42 atualizações frente ao projeto original, ou seja, 42 mesclagens de código entre o *branch master* do sistema customizado com o *branch master* do sistema original (i.e., atualizações de PO foram incorporadas ao PC). Essas atualizações foram identificadas com apoio do *framework* Python `PyDriller`[2], que permitiu buscar as integrações do tipo *merge* no repositório do projeto customizado que envolviam um *commit* do projeto original. Vale ressaltar que a identificação desses "momentos" de atualização podem ser realizadas com o apoio de recursos existentes em diversas linguagens de programação. É o histórico de *commits* dos dois projetos que define as datas de integração. Por fim, ressalta-se que a identificação desses momentos de atualização foi importante para fins de análise, mas não é insumo utilizado no processo de customização proposto pela abordagem.

A rodada de avaliação #1 abrangeu o período entre 29/08/2014 e 13/11/2014, referente à data de criação do *fork* até a primeira atualização. A rodada #2 compreendeu o período entre a primeira (13/11/2014) e a segunda atualizações (19/05/2015). As demais rodadas foram estabelecidas tal como a rodada #2: período entre última atualização e a atualização seguinte (Tabela 5.1).

5.3 Questão *QP #1*

Uma análise com foco na frequência de atualizações de um projeto customizado frente ao projeto original (i.e., momentos em que atualizações de PO foram incorporadas ao PC) foi realizada de forma a assegurar que esta dissertação aborda um problema relevante.

5.3.1 Método

Apura-se a quantidade de dias e a quantidade de *commits* locais entre cada atualização. Mais importante, por meio de simulação de mesclagens de código, apura-se a quantidade de conflitos que ocorreram em cada um dos momentos de atualização.

[2] Disponível em <https://github.com/ishepard/pydriller>.

5.3.2 Resultados e Discussão

A Tabela 5.1 reporta a quantidade de dias decorridos entre as atualizações, i.e., a diferença em dias entre a última atualização (coluna Dt. final) e a atualização anterior (coluna Dt. inicial). Apesar das atualizações do PO serem diárias, a quantidade de dias que o PC levou para incorporar essas atualizações variou consideravelmente – de 0 a 202 dias – evidenciando falta de uniformidade em atualizações do sistema do IFSULDEMINAS com relação ao sistema original. Conjectura-se que isso ocorreu por causa da dificuldade na resolução de conflitos de mesclagem, tal como afirmam Nishimura e Maruyama (NISHIMURA; MARUYAMA, 2016) e McKee et al. (MCKEE et al., 2017).

Após o período mais longo sem atualizações, que ocorreu na rodada #4 (202 dias) e que ocasionou 21 trechos de código com conflitos, conjectura-se que houve preocupação por parte da instituição em realizar atualizações mais constantes, por exemplo, rodadas #5 à #14 ocorreram dentro de um único mês. Na rodada #15, entretanto, as atualizações frequentes deixaram de ser adotadas. Conjectura-se que os desenvolvedores envolvidos no processo de atualização tiveram dificuldades em tratar conflitos e postergaram ao máximo a atualização, que ocorreu após 88 dias, resultando em 26 conflitos. Essa é uma prática comum no desenvolvimento de sistemas, tal como evidenciado por Nelson et al. (NELSON et al., 2019).

Entre as rodadas #16 à #19, ocorreu novamente a preocupação com atualizações constantes, novamente interrompidas potencialmente pela dificuldade de tratar conflitos de *merge*. As rodadas #20 e #21 evidenciam esse fato, com períodos mais longos entre atualizações e maior volume de conflitos. Os períodos que envolvem atualizações mais frequentes com menor volume de conflitos *versus* atualizações mais espaçadas com maior volume de conflitos se repetem ao longo de todas as rodadas.

Um fator que agrava a dificuldade de atualizar o código fonte do projeto customizado frente ao projeto original é o maior volume de customizações próprias que tendem a aumentar a quantidade de conflitos. Tal afirmação tem como base a análise dos períodos compreendidos entre as rodadas #32 à #34 (em destaque na Tabela 5.1). Esse período concentrou o maior volume percentual de *commits* realizados pelo IFSULDEMINAS (39,57% do total) e também o maior volume percentual de conflitos dentro do período analisado (44,15% do total). Foi um período longo que compreendeu 204 dias, equiparando-se àquele descrito na rodada #4.

Das rodadas #35 à #41 ocorreu novamente a preocupação em realizar atualizações constantes. O volume de *commits* próprios (média de 26,6 por rodada) foi proporcionalmente maior

Tabela 5.1 – Períodos considerados em cada rodada de testes.

#	Abrangência				Commits	Conflitos	
	Ano	**Dt. inicial**	**Dt. final**	**Nº dias**	**Total**	**Arq.**	**Trechos**
1	2014	29/08	13/11	76	3	0	0
2		13/11	19/05	187	2	2	2
3	2015	19/05	17/06	29	3	1	1
4		17/06	05/01	202	21	10	21
5		05/01	08/01	3	4	1	1
6		08/01	13/01	5	4	1	1
7		13/01	14/01	1	3	0	0
8		14/01	18/01	4	2	0	0
9		18/01	21/01	3	5	0	0
10		21/01	22/01	1	2	0	0
11		22/01	25/01	3	1	0	0
12		25/01	26/01	1	1	0	0
13		26/01	27/01	1	3	0	0
14	2016	27/01	28/01	1	1	0	0
15		28/01	25/04	88	78	13	26
16		25/04	28/04	3	6	0	0
17		28/04	29/04	1	1	2	2
18		29/04	29/04	0	1	1	1
19		29/04	02/05	3	2	0	0
20		02/05	13/07	72	4	20	43
21		13/07	16/09	65	66	20	40
22		16/09	21/09	5	6	2	8
23		21/09	29/09	8	8	3	3
24		29/09	09/01	102	100	8	12
25		09/01	11/01	2	5	0	0
26		11/01	11/01	0	1	2	2
27		11/01	02/05	111	76	33	53
28	2017	02/05	07/06	36	38	8	10
29		07/06	22/08	76	97	24	40
30		22/08	11/10	50	86	20	30
31		11/10	13/11	33	58	27	49
32		13/11	05/02	84	145	65	218
33		05/02	12/04	66	237	35	64
34		12/04	05/06	54	276	40	50
35		05/06	12/06	7	51	11	13
36		12/06	19/06	7	42	6	8
37	2018	19/06	22/06	3	28	1	1
38		22/06	29/06	7	35	8	16
39		29/06	02/07	3	4	1	6
40		02/07	04/07	2	4	3	4
41		04/07	11/07	7	22	8	11
42		11/07	08/08	28	131	7	16
			Desde o início:	**1.440**	**1.663**	**383**	**752**

Fonte: Do autor (2021).

quando comparado ao período entre as rodadas #5 a #14 do início do projeto (média de 2,6 por rodada) quando também ocorreu essa mesma preocupação. O maior volume de *commits* locais

fez com que o volume de conflitos também se elevasse de apenas dois trechos em conflitos entre as rodadas #5 e #14 para 59 trechos em conflitos entre as rodadas #35 e #41. Isso significa, de forma relativa, que elevou-se a média de 0,2 para 8,42 trechos em conflito por rodada.

Resumo: O projeto customizado avaliado buscou atualizações frente ao seu projeto original em média uma vez a cada um mês e cinco dias dentro do período de 4 anos. Não houve uniformidade entre os períodos de atualização, que oscilaram entre 0 e 202 dias. Observou-se a alternância entre períodos longos e períodos curtos de atualização. Conjectura-se que isso ocorreu por causa das dificuldades encontradas pelos desenvolvedores quanto à resolução dos conflitos de mesclagem. Observou-se também que o maior volume de customizações locais elevou a ocorrência de conflitos.

5.4 Questão *QP #2*

Em uma segunda análise, avaliou-se a expressividade da linguagem de customização proposta. O objetivo é entender como a linguagem se comporta durante a construção gradativa dos arquivos de customização e se a linguagem é capaz de tratar todas as necessidades do cenário real de software avaliado.

5.4.1 Método

Os arquivos de customização são construídos de forma gradativa, rodada a rodada. O objetivo desses arquivos é manipular o código fonte do projeto original **`S`** de modo a transformá-lo no projeto customizado **`S'`**. Essa etapa foi realizada de forma manual com apoio de ferramentas de análise por um desenvolvedor do IFSULDEMINAS que é também o autor desta dissertação.

Como desenvolvimento de software é um processo evolutivo, apura-se, para cada rodada, o número de arquivos criados, o número de arquivos já existentes e alterados e o número de arquivos eventualmente excluídos. Deve-se observar que – nessa etapa da avaliação – os arquivos de customização não foram construídos tal como descrito na Seção 3.2 porque não foi possível obter o histórico de *features* tratadas e implementadas ao longo dos quatro anos de desenvolvimento avaliados. Para viabilizar a análise, os arquivos de customização contemplaram mudanças por módulos do sistema. Logo, foi criado um arquivo de customização para cada módulo.

Por fim, verifica-se se o projeto customizado **`S'`** obtido após a aplicação das transformações descritas pelos arquivos de customização é igual ao projeto customizado resultante da resolução manual de conflitos por parte dos desenvolvedores do IFSULDEMINAS na mesclagem com o *branch master* do projeto original, em cada um dos períodos avaliados. A igualdade entre projetos é avaliada por meio do módulo de análise de repositórios desenvolvido para esse propósito (ver Capítulo 4).

5.4.2 Resultados e Discussão

Construção gradativa. Ao longo de todas as rodadas de atualização a quantidade de arquivos DSL alterados foi baixo, com uma média geral de 5,56% (Tabela 5.2). Em apenas 9 de 42 rodadas, os percentuais relativos ficaram acima de 10% que correspondem justamente aos períodos em que ocorreram os maiores volumes de customizações por parte do IFSULDEMINAS. Isso evidencia que boa parte das instruções de customização criadas em uma etapa anterior não necessitam de alteração pois continuam atendendo ao seu objetivo, mesmo quando o sistema original é atualizado.

Mais importante, para todas as rodadas de avaliação, o projeto customizado obtido a partir das instruções de customização foi igual ao projeto customizado resultante da resolução manual de conflitos por parte dos desenvolvedores do IFSULDEMINAS na mesclagem com o *branch master* do projeto original. Ressalta-se que, em apenas quatro dos 195 arquivos customizados, a ferramenta proposta falhou ao tratar uma instrução particular do *framework* `Django` relacionada ao registro de *models* no ambiente de administração automática desse *framework*. A falha gerou erro de execução da ferramenta e a correção para esse problema será contemplada em uma versão futura do protótipo.

Expressividade da linguagem. A Tabela 5.3 descreve o uso de nove das onze instruções de customização. A décima instrução – **`add file`** – não foi considerada na análise porque foi criada posteriormente para formalizar a necessidade de adição de novos arquivos ao projeto customizado. As nove instruções de customização propostas e consideradas na análise mostraram-se eficazes em realizar as customizações de um projeto durante quatro anos.

Embora especificadas e analisadas nove instruções de customização, notou-se o uso expressivo de três instruções em particular. A primeira delas – **`replace unit`** – foi utilizada 245 vezes ao longo de 58 arquivos de customização distintos (47,57% do total). Apesar do

Tabela 5.2 – Escrita gradual de arquivos de customização

#	Novos	Existentes				
		Total	Alterados	%	Excluídos	%
1	3	0	0	-	0	-
2	3	3	0	0,00%	0	0,00%
3	6	6	0	0,00%	0	0,00%
4	19	12	0	0,00%	0	0,00%
5	2	31	2	6,45%	0	0,00%
6	0	33	2	6,06%	0	0,00%
7	0	33	1	3,03%	0	0,00%
8	0	33	1	3,03%	0	0,00%
9	0	33	1	3,03%	0	0,00%
10	0	33	1	3,03%	0	0,00%
11	0	33	1	3,03%	0	0,00%
12	0	33	1	3,03%	0	0,00%
13	0	33	1	3,03%	0	0,00%
14	0	33	1	3,03%	0	0,00%
15	18	33	7	21,21%	0	0,00%
16	1	51	1	1,96%	0	0,00%
17	0	52	1	1,92%	0	0,00%
18	0	52	1	1,92%	0	0,00%
19	1	52	0	0,00%	0	0,00%
20	3	54	9	16,67%	0	0,00%
21	7	55	8	14,55%	0	0,00%
22	0	63	1	1,59%	0	0,00%
23	0	63	0	0,00%	0	0,00%
24	6	63	10	15,87%	0	0,00%
25	0	69	1	1,45%	0	0,00%
26	0	69	1	1,45%	0	0,00%
27	21	69	11	15,94%	2	2,90%
28	7	88	11	12,50%	1	1,14%
29	6	94	8	8,51%	0	0,00%
30	30	100	16	16,00%	0	0,00%
31	9	130	15	11,54%	0	0,00%
32	10	139	18	12,95%	3	2,16%
33	21	147	10	6,80%	0	0,00%
34	18	167	12	7,19%	1	0,60%
35	1	184	6	3,26%	3	1,63%
36	8	182	10	5,49%	0	0,00%
37	3	190	2	1,05%	0	0,00%
38	1	193	4	2,07%	0	0,00%
39	1	194	3	1,55%	0	0,00%
40	0	195	1	0,51%	0	0,00%
41	0	198	6	3,03%	0	0,00%
42	4	195	8	4,10%	1	0,51%

amplo uso, percebe-se pela análise da Tabela 5.3 que houve uma distribuição relativamente uniforme dessa utilização com relação ao tipo de unidade de código tratada: funções (37,96% do total), métodos (31,84%) e classes (30,20%). Essa distribuição segue, portanto, uma linha onde

Tabela 5.3 – Uso de instruções de customização.

Instrução		**Nº utiliz.**		**% utiliz.**		**Nº dsls**
add @annotation		6		1,17%		2
	classe		*28*		*30,77%*	
add unit	*função*	91	*53*	17,67%	*58,24%*	36
	método		*10*		*10,99%*	
remove @annotation		2		0,39%		2
remove string		0		0,00%		0
remove unit		0		0,00%		0
replace @annotation		5		0,97%		2
replace file		122		23,69%		122
replace string		44		8,54%		27
	classe		*74*		*30,20%*	
replace unit	*função*	245	*93*	47,57%	*37,96%*	58
	método		*78*		*31,84%*	

Fonte: Do autor (2021).

customizações mais específicas (funções e métodos) foram tratadas antes de customizações mais amplas (classes).

Acredita-se que a ampla utilização de **`replace unit`** se deu pelo fato de a instrução representar uma customização que se adequa à maioria das necessidades de customização, possibilitando a tratativa de situações em diferentes níveis de granularidade.

A segunda instrução mais utilizada – **`replace file`** – teve seu uso expressivo justificado pela forma como a abordagem de customização foi idealizada, aliada às características do projeto customizado. Como se trata de um projeto web, o uso de arquivos HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e JavaScript é significativo. No entanto, as instruções de customização focam na linguagem principal do sistema e os arquivos supracitados são, em geral, customizados "por completo" substituindo-se o arquivo fonte **`file`** pelo arquivo fonte **`file'`** correspondente.

A terceira instrução mais utilizada – **`add unit`** – teve seu uso justificado por sua semântica: criação de novas classes, funções e métodos. Isso permitiu principalmente customizações que agregaram novas *features* ao sistema, parte significativa do processo como um todo.

Em uma última análise, observou-se a ausência de utilização das instruções **`remove unit`** e **`remove string`**. Embora idealizadas e implementadas durante a prova de conceito descrita na Seção 3.4.1, não foi observada a necessidade de sua utilização ao longo das 42 rodadas de avaliação.

Resumo: A abordagem proposta mostrou-se aplicável a um cenário real de software. Foram avaliadas a construção dos arquivos de customização, o código resultante após execução da ferramenta e a expressividade das instruções criadas. Os arquivos de customização foram construídos de forma gradativa e incremental, com baixos percentuais de alteração (média de 5.56%). A DSL proposta mostrou expressividade suficiente para tratar todas as necessidades de customização identificadas ao longo do período avaliado (quatro anos), gerando o código fonte equivalente ao resultante da resolução manual de conflitos. Três de nove instruções avaliadas (**`replace unit`**, **`replace file`** e **`add unit`**) concentram o maior volume de utilizações, ao passo que duas instruções (**`remove unit`** e **`remove string`**) não foram utilizadas durante a avaliação.

5.5 Questão *QP #3*

Em uma terceira análise, investigou-se a hipótese de que a abordagem proposta contribui na diminuição dos conflitos de mesclagem.

5.5.1 Método

Analisa-se a quantidade de conflitos que ocorreram ao longo das 42 atualizações de código e a quantidade de arquivos de customização criados e mantidos ao longo desse mesmo período.

5.5.2 Resultados e Discussão

Durante as atualizações frente ao projeto original usando a abordagem proposta, não houve ocorrência de conflitos de mesclagem, pois a abordagem sempre busca o código fonte do projeto original e aplica sobre ele as transformações necessárias; não há mesclagem de código e, sendo assim, a ausência de conflitos se justifica. É importante observar, todavia, que a adoção da abordagem torna possível a ocorrência de outros tipos de conflitos, previamente detalhados na Seção 3.5.

Conforme apresentado na Tabela 5.1, conflitos em 752 trechos de código deixaram de ocorrer. Com a abordagem proposta, evitou-se uma média de $\sim$18 trechos em conflito por atualização.

Por fim, observou-se que, quanto mais customizado é o sistema, mais conflitos de mesclagem ocorrem, independentemente de quão frequentes ou não são realizadas as atualizações. Em uma análise Pearson da correlação pareada entre a quantidade de *commits* locais e a quantidade de conflitos observou-se alta correlação dos valores apresentados ($rho = 0.7786$). Para testar a hipótese formal dessa correlação utiliza-se como critério de decisão rejeitar a hipótese nula $r = 0$ quando o valor absoluto exceder os valores críticos. Para um nível de significância $\alpha = 0.050$ e tamanho da amostra $n = 42$ temos valor crítico $rc \approx 0.312$[3]. Como $rho > rc$ rejeita-se a hipótese nula, o que significa que há evidência suficiente para apoiar a afirmativa de correção linear entre as duas variáveis.

Resumo: A abordagem evita os conflitos de mesclagem decorrentes da fusão do código fonte do projeto customizado com o código fonte do projeto original já que essa fusão não é mais necessária. Em contrapartida, outros tipos de conflitos passam a ser possíveis e devem ser considerados (ver Seção 3.5) .

5.6 Discussão

Através da análise dos resultados obtidos para a *QP #1* foi possível observar que a frequência de atualização do projeto customizado frente ao projeto original foi relativamente alta. Ao longo dos 48 meses analisados, ocorreram 42 atualizações, o que implica aproximadamente em uma atualização a cada um mês e cinco dias. Foi possível observar ainda que uma frequência curta ou longa de atualizações não soluciona a ocorrência de conflitos de *merge*, i.e., os conflitos persistiram mesmo com atualizações constantes de código.

A análise dos resultados obtidos para a *QP #2* mostrou que, ao longo de um período de quatro anos, arquivos de customização se mostraram suficientes para obter o projeto customizado **`S'`** a partir do código fonte do projeto original. Em resumo, **`S'`** foi sempre equivalente ao sistema customizado pelos próprios desenvolvedores do IFSULDEMINAS. Ademais, a quantidade de arquivos de customização cresceu de forma gradativa e com baixos percentuais de alteração (Tabela 5.2).

Por fim, a análise dos resultados obtidos para a *QP #3* mostrou que a abordagem possui potencial para evitar conflitos de mesclagem em cenários reais. Com a abordagem proposta,

[3] Conforme tabela disponível em <http://www.sestatnet.ufsc.br/data/downloads/10/Tabela_coeficientes_de_correlacao.pdf>.

conflitos em 752 trechos de código deixaram de ocorrer. Isso significa que menos tempo e trabalho poderiam ter sido empregados para manter o sistema atualizado, ajudando a equipe de desenvolvimento a cumprir prazos e atingir metas.

5.7 Ameaças à validade

É possível identificar duas ameaças à validade para este estudo.

Validade interna: A criação dos arquivos de customização foi realizada pelo autor desta dissertação e, embora não houve revisão por pares, o autor desta dissertação é também desenvolvedor do IFSULDEMINAS e apenas utilizou as instruções propostas para a customização de trechos de código do projeto original como havia sido feito após resolução de conflitos de *merge*.

Validade externa: Embora a expressividade da DSL proposta tenha demonstrado eficaz para tratar todas as necessidades de customização avaliadas, não se pode afirmar que a abordagem proposta fornecerá resultados equivalentes em outros sistemas, o que é usual em estudos de caso em engenharia de software. No entanto, foi analisado um período longo de quatro anos em um projeto real de software.

5.8 Considerações Finais

Neste capítulo, foi apresentada a avaliação histórica realizada sobre um projeto real frequentemente atualizado frente ao seu projeto original. Por meio de três questões de pesquisa, avaliou-se com que frequência o projeto customizado se atualiza, se a abordagem proposta é aplicável em um cenário real e se existem indícios de que ela evita conflitos.

Definiu-se um período de quatro anos para avaliação desse sistema, com início no momento de sua criação (agosto/2014) até a última mesclagem de atualização de código realizada no mês de agosto/2018. Dentro desse período ocorreram 42 atualizações frente ao projeto original, o que implica em quase uma atualização por mês, em média.

Constatou-se falta de uniformidade entre períodos de atualização, com alternância entre períodos curtos e períodos longos. Conjectura-se que isso tenha ocorrido por causa das dificuldades encontradas pelos desenvolvedores responsáveis pelo processo de atualização. A literatura da área indica que há tendência natural dos desenvolvedores em postergar a mescla-

gem do código quando muitos conflitos ocorrem, ou quando esses são de difícil resolução. O fato agrava-se ao longo do tempo com aumento do volume de customizações locais.

A abordagem mostrou-se aplicável a um cenário do mundo real. Os arquivos de customização para o sistema avaliado foram construídos de forma gradativa e incremental ao longo das 42 rodadas de atualização analisadas, com baixos percentuais de alteração. As instruções da DSL proposta mostram-se suficientes para tratar todas as necessidades de customização identificadas ao longo do período de tempo avaliado gerando, ao final de cada período, um código fonte equivalente àquele obtido na ocasião da resolução manual de conflitos por parte dos desenvolvedores do sistema real. Novas avaliações que contemplem novos sistemas e novas linguagens de programação são necessárias para uma avaliação mais geral da aplicabilidade da abordagem.

Três das nove instruções avaliadas foram as mais utilizadas: (i) **`replace unit`** (47,57% do total), onde acredita-se que seja aquela que mais se adéqua às diferentes necessidades de customização; (ii) **`replace file`** (23,69% do total), cuja utilização está ligada à forma como a abordagem foi idealizada aliada às características do projeto analisado; e (iii) **`add unit`** (17,67% do total), que tem seu uso justificado pelo fato de permitir a criação de novas classes, métodos e funções, algo essencial para criação de novas *features* em um projeto customizado.

Por fim, foi possível constatar que a abordagem tem potencial para evitar conflitos de mesclagem pois atua diretamente sobre a cópia do código fonte atualizado do sistema original, transformando-o no projeto customizado. Não há, portanto, mesclagem de código entre os códigos fonte dos dois projetos, o que justifica a ausência de conflitos dessa natureza. Deve-se considerar que a adoção da abordagem torna possível a ocorrência de outros tipos de conflitos, tratados neste trabalho como conflitos de customização (ver Seção 3.5).

6 AVALIAÇÃO FRENTE AOS DESENVOLVEDORES

Este capítulo avalia a perspectiva dos desenvolvedores de um projeto de software frequentemente atualizado frente ao seu projeto original com relação à abordagem proposta. Os objetivos da avaliação são:

1 - identificar se a abordagem seria aceita pelos desenvolvedores;

2 - se os desenvolvedores vêem vantagem ao utilizá-la;

3 - confirmar se os desenvolvedores acreditam que é possível implementar *features* reais usando a abordagem;

4 - confirmar se os desenvolvedores concordam com sua aplicabilidade em cenários concretos.

6.1 Participantes

A avaliação envolve desenvolvedores do projeto SUAP dentro do IFSULDEMINAS. No momento em que este estudo foi conduzido, quatro desenvolvedores trabalhavam ativamente no projeto e puderam ser envolvidos. O perfil desses participantes é apresentado na Tabela 6.1.

Tabela 6.1 – Perfil dos participantes.

Des.	Formação	Idade	Exp. I[1]	Exp. II[2]	Exp. III[3]
D1	Mestrado - área de Computação ou afins	42	8 a 10 anos	1 a 2 anos	1 a 2 anos
D2	Mestrado - área de Computação ou afins	34	10 a 15 anos	4 a 6 anos	4 a 6 anos
D3	Mestrado - outras áreas	31	6 a 8 anos	4 a 6 anos	4 a 6 anos
D4	Esp. - área de Computação ou afins	28	2 a 4 anos	1 a 2 anos	1 a 2 anos

Fonte: Do autor (2021).

Todos possuem graduação na área de Computação (não expresso na tabela) e a maioria com formação complementar também nessa área (75% dos participantes), sendo que desses 66% Mestrado. A experiência em desenvolvimento de software é variada, mas a grande maioria possui mais de 6 anos de experiência (75% dos participantes), com média geral de 10 anos. Tanto a experiência em desenvolvimento com as tecnologias `Python` e `Django` quanto com o sistema SUAP é equilibrada. Metade dos participantes possui de 4 a 6 anos de experiência tanto

[1] Tempo de experiência em desenvolvimento de software.

[2] Tempo de experiência em desenvolvimento com `Python/Django`.

[3] Tempo de experiência em desenvolvimento no SUAP.

em `Python` e `Django` quanto com o SUAP. Os demais participantes possuem de 1 a 2 anos de experiência.

6.2 Método

Selecionam-se quatro *features* reais do *backlog* de desenvolvimento do SUAP, uma para cada desenvolvedor. Cada desenvolvedor implementa sua *feature* em um *branch* distinto criado a partir do *branch master*. Esses *branches* não podem ser atualizados com outros *branches* durante o desenvolvimento do estudo, sob qualquer circunstância, de tal modo que representem fielmente a implementação de cada *feature*.

Ao término do desenvolvimento de cada *feature* F_i, o pesquisador recebe acesso de leitura ao código fonte. Com base nesse código, o pesquisador implementa cada uma das *features* F_1 à F_4 utilizando a abordagem proposta no estudo.

Uma reunião de trinta minutos é agendada com cada desenvolvedor para que o pesquisador:

- apresente uma visão geral da abordagem proposta no estudo, com foco em sua essência (5 min.);
- explique ao desenvolvedor como sua *feature* F_i ficaria implementada dentro da abordagem proposta (15 min.);
- solicite ao desenvolvedor que responda às questões apresentadas em formato de uma entrevista orientada (10 min.).

6.3 Entrevista orientada

As questões para a entrevista orientada foram elaboradas de tal modo a alcançar os objetivos descritos estabelecidos para a avaliação. O formulário completo com as questões propostas e opções de respostas em escala *Likert* constam no Apêndice A. A correspondência entre cada questão elaborada e os objetivos da avaliação é apresentada na Tabela 6.2.

Para alcançar o objetivo #1, todas as respostas das questões elaboradas são consideradas. Para o objetivo #2, consideram-se apenas as respostas da questão QP #2. Para os objetivos #3 e #4, consideram-se as respostas das questões QP #1, QP #3 e QP #4.

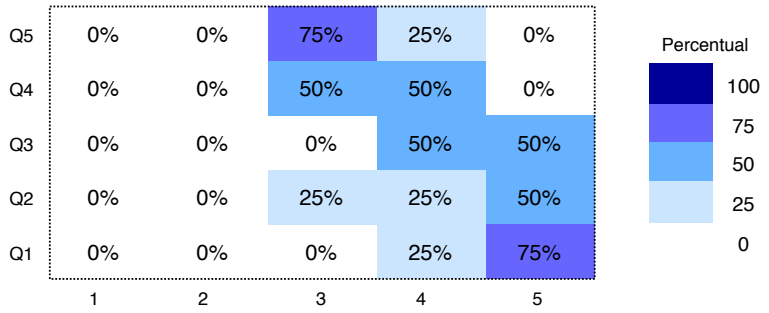
A Figura 6.1 apresenta os percentuais obtidos nas respostas a cada uma das questões elaboradas.

Tabela 6.2 – Questões de pesquisa *versus* objetivos.

Questões	Objetivo #1	Objetivo # 2	Objetivo # 3	Objetivo # 4
QP #1	x		x	x
QP #2	x	x		
QP #3	x		x	x
QP #4	x		x	x
QP #5	x			

Fonte: Do autor (2021).

Figura 6.1 – Percentuais por questão.



Fonte: Do autor (2021).

Conforme observa-se na Figura 6.1, os percentuais indicam uma aceitação geral da abordagem de customização proposta nesta dissertação. Uma discussão detalhada para os resultados obtidos em cada questão é realizada nas próximas seções.

6.3.1 *QP #1* - Você entendeu como implementamos a *feature* utilizando a abordagem proposta?

Todos entenderam como as *features* desenvolvidas por eles foram implementadas utilizando a abordagem proposta, sendo que três dos quatro participantes entenderam perfeitamente. Não foram expostas dúvidas ou comentários adicionais.

Resumo: Todos os participantes entenderam como as *features* foram implementadas utilizando a abordagem proposta.

6.3.2 *QP #2* - Você achou a abordagem interessante?

As respostas foram variadas, com tendência positiva. Dois dos quatro participantes acharam a abordagem muito interessante (desenvolvedores D1 e D3), enquanto um achou bem interessante (desenvolvedor D4) e o outro interessante (desenvolvedor D3). Nas justificativas houve consenso com relação à relevância do tema. O impacto negativo do cenário atual de

atualização entre o projeto customizado e o projeto original foi citado por todos, com ênfase no tempo que se perde em resolver os conflitos de mesclagem.

Os participantes opinaram ainda sobre a adoção da abordagem com relação ao cenário atual, com diferentes opiniões. O desenvolvedor D2 acredita que um experimento faz-se necessário para comparar o custo de trabalhar com a abordagem proposta frente ao custo de resolução de conflitos no cenário atual. Tal necessidade é discutida com detalhes na Seção 8.3. O desenvolvedor D3 sinalizou que, de imediato, preocupou-se com tempo necessário para criar os arquivos de customização, o que poderia ser uma limitação. Entretanto, ponderou que essa limitação, quando comparada à dificuldade do processo atual de resolução de conflitos, torna-se pequena. Os demais deram ênfase na dificuldade do processo atual e não traçaram comparativos entre uma ou outra forma de conduzir a atualização frente ao projeto original.

Resumo: Todos acharam a abordagem, no mínimo, interessante. Destacaram a relevância do tema e o tempo perdido na resolução manual de conflitos de mesclagem. Novos experimentos foram apontados como necessários para comparar o custo de adoção da abordagem frente ao custo de resolução de conflitos no cenário atual.

6.3.3 *QP #3* - Você acredita que poderia funcionar dentro do cenário recomendado pela abordagem?

Existem grandes chances da abordagem proposta funcionar quando aplicada desde o início em um projeto novo, segundo afirmaram os participantes. Metade dos participantes acredita que existem grandes chances para isso ocorrer (desenvolvedores D1 e D2), enquanto a outra metade afirma ser muito provável funcionar (desenvolvedores D3 e D4). O desenvolvedor D1 lembrou que muito ainda deve ser estudado com relação a aplicação da abordagem, mas que funcionará. O desenvolvedor D3 compreendeu que, ao adotar a abordagem proposta desde o início de um novo projeto, as customizações vão sendo criadas de forma incremental, o que facilita sua adoção.

Resumo: Todos acreditam que existem grandes chances da abordagem funcionar quando aplicada a projetos novos, desde seu início. A análise com relação a projetos já em andamento é tratada na próxima questão.

6.3.4 *QP #4* - Você acredita que poderia funcionar dentro do cenário de um projeto antigo / em andamento (tal como o SUAP no IFSULDEMINAS), com certas adaptações?

Os participantes foram cautelosos ao afirmar se a abordagem poderia ou não funcionar no cenário de projetos antigos, em andamento. Apesar disso, há uma tendência positiva nesse sentido.

Afirmam que adaptações seriam, certamente, necessárias. O desenvolvedor D3 aponta que, de uma forma geral, deveria haver uma adaptação do fluxo do processo de desenvolvimento atual para adoção da abordagem. Afirma também que a criação dos arquivos de customização poderia ser feita pelo desenvolvedor ou poderia haver uma pessoa específica que ficasse responsável por essa atividade, alguém que possuísse uma visão mais ampla de tudo o que vem sendo desenvolvido pela equipe.

O desenvolvedor D2 sentiu-se relutante ao dar sua resposta. Ele afirma que a abordagem poderia funcionar, com adaptações, mas questiona: "a que custo?". Quanto às adaptações, afirma que descrevê-las de imediato seria algo muito difícil de fazer pois ainda são abstratas. Mesmo assim, apresentou sugestões de adaptações interessantes que poderiam ampliar a atuação da ferramenta de customização e, consequentemente, a aceitação da abordagem, como recomendações de criação automatizada de arquivos de customização, análise de dependências de código etc.

O desenvolvedor D1 focou no volume de implementações locais realizadas e o trabalho a ser dispensado para "alcançar" o código atual do sistema em produção. É um ponto relevante que deve ser considerado. Quanto às adaptações, acredita que envolveriam questões relacionadas ao uso da abordagem e ao conhecimento necessário para sua adoção. Seriam situações semelhantes àquelas que a equipe de desenvolvimento do IFSULDEMINAS enfrentou no passado quando da adoção do `Git` e mais tarde do `Docker`[4], bem como de outras práticas de *DevOps*[5]. Todos tiveram que conhecer as novas tecnologias e fazer as adaptações necessárias e hoje percebem os facilitadores introduzidos ao adotá-las.

[4] Disponível em <https://www.docker.com/>.

[5] Contração de *development* e *operations*, é um termo criado para descrever um conjunto de práticas para integração entre as equipes de desenvolvimento de softwares e de operações (infraestrutura) e a adoção de processos automatizados para produção rápida e segura de aplicações e serviços (4LINUX, 2020).

Resumo: Os participantes acreditam que a abordagem pode funcionar quanto aplicada a cenários de sistemas já em andamento, desde que ocorram adaptações que facilitem sua adoção. Ressalta-se, contudo, que metade dos participantes possuem preocupações quanto ao custo-benefício para adoção da abordagem decorrente do esforço necessário para adotá-la.

6.3.5 *QP #5* - Como você vê o fato de desenvolver de forma desacoplada (desenvolver um código customizado + arquivos de customização separados do código principal do sistema)?

Três dos quatro participantes acreditam que desenvolver de forma desacoplada pode ser desconfortável no início, mas, com o tempo, acostuma-se. Dois deles (desenvolvedores D1 e D2) realizaram analogias diferentes, mas equivalentes, com relação à adoção de novas tecnologias: desenvolvimento focado em testes de software e a adoção de um novo *framework*. Destacam que, no início, há desconforto e resistência de adaptação. Contudo, com o tempo, acabam percebendo as vantagens na adoção de uma nova tecnologia, com a minimização de problemas e ganho de tempo. Em suma, acostuma-se tendo consciência dos benefícios que são obtidos. O desenvolvedor D3 destacou que o "acostuma-se" por certas vezes não é muito saudável, principalmente em situações onde uma nova tecnologia é empregada sem discussão entre as partes envolvidas, fazendo com que o trabalho se torne oneroso e pouco prazeroso.

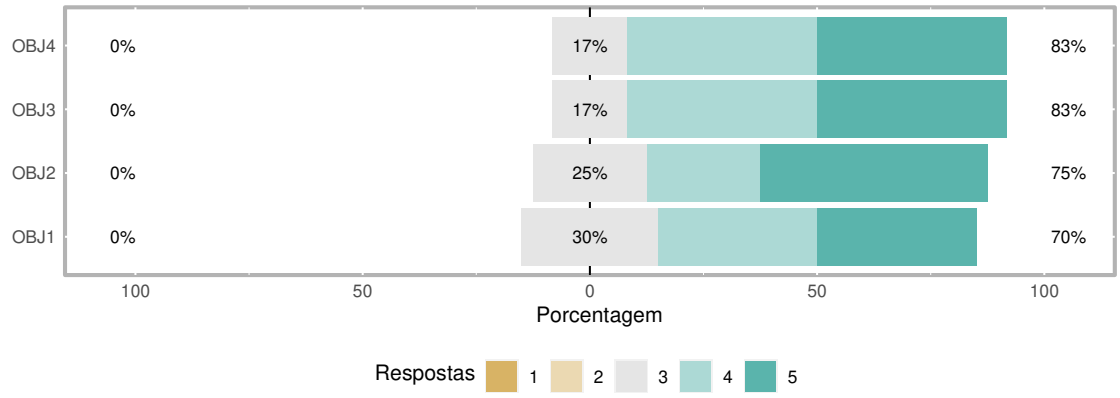
Resumo: A maioria dos participantes (75%) acredita que desenvolver de forma desacoplada pode ser desconfortável no início, mas acostuma-se. Dois realizaram analogias com a adoção de novas tecnologias. Um terceiro participante destacou que o ato de "acostumar" pode não ser saudável quando realizada a contragosto das partes envolvidas.

6.4 Discussão

A análise da Figura 6.1 monstra tendência de aceitação da abordagem proposta por parte dos desenvolvedores. Essa tendência acentua-se nas respostas para as questões QP#2 e QP#3, embora os desenvolvedores tenham demonstrado preocupações quanto ao esforço necessário para sua adoção. As questões QP#4 e QP#5 sugerem um resultado mais equilibrado, mas ainda com tendência em favor da abordagem.

A análise isolada das respostas em função dos objetivos propostos também corrobora com aceitação geral da abordagem, tal como ilustrado na Figura 6.2. As respostas foram agrupadas em função dos objetivos, seguindo a organização apresentada na Tabela 6.2 .

Figura 6.2 – Respostas dos participantes por objetivo.



Fonte: Do autor (2021).

A aceitação da abordagem, identificada como primeiro objetivo, é forte ou muito forte entre 70% das respostas. Esse percentual eleva-se quanto ao segundo objetivo se os desenvolvedores vêem vantagem ao utilizá-la (75%) e chega em seu ápice (83%) na avaliação para os objetivos três e quatro que tratam da aplicabilidade da abordagem em cenários do mundo real.

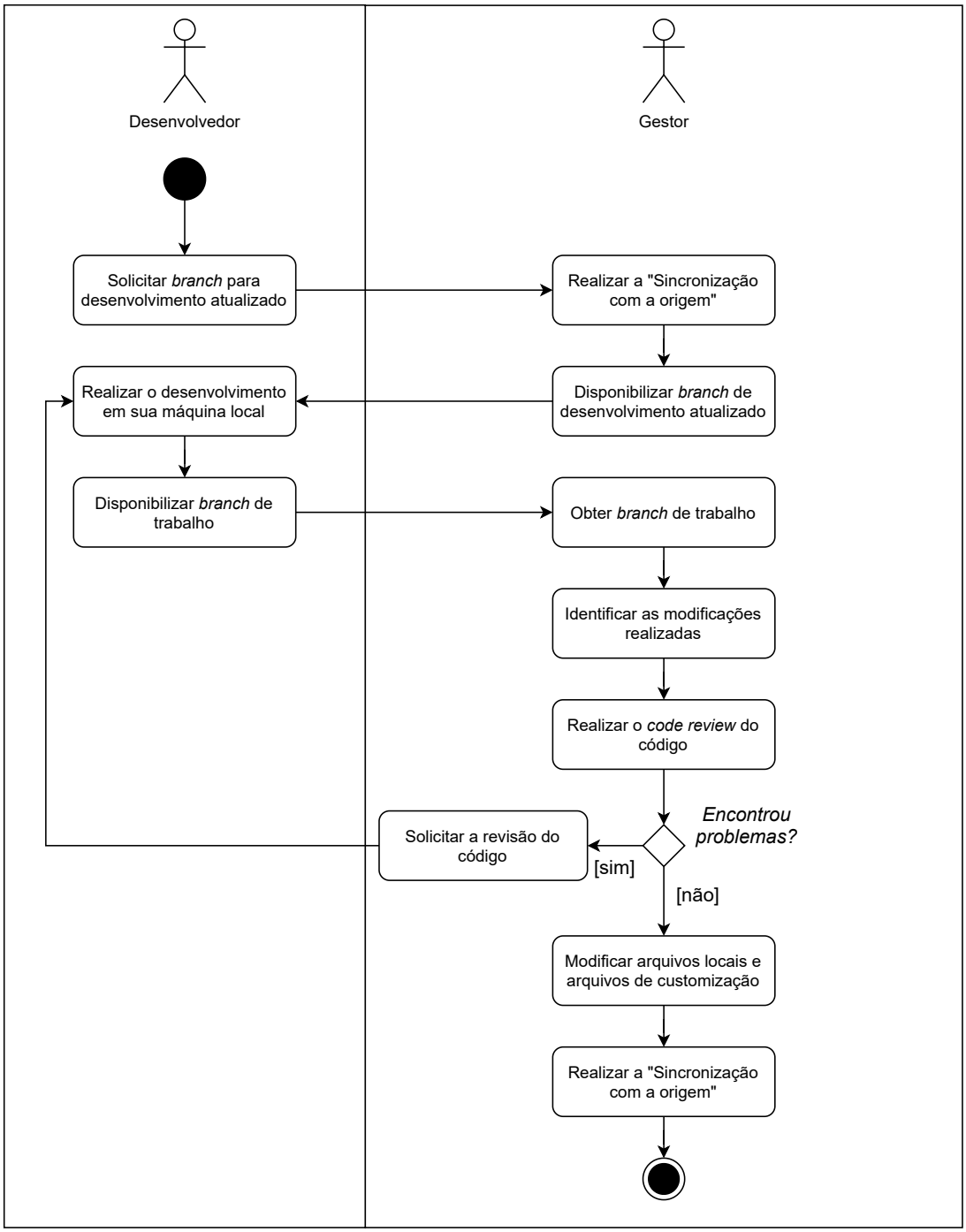
Por fim, embora tenha se observado uma tendência positiva em favor da abordagem, um ponto importante foi destacado pelos participantes. Trata-se da necessidade de comparação entre o custo de adoção da nova abordagem frente ao custo do cenário atual de resolução de conflitos. Essa necessidade é discutida em detalhes na Seção 8.3.

6.5 Lições aprendidas

A realização das entrevistas orientadas junto aos desenvolvedores de um projeto real levantou ideias para proposta de um novo *workflow* de desenvolvimento, diferente daquele descrito na Seção 3.3. Nesse *workflow*, conjectura-se que o trabalho dos desenvolvedores é pouco afetado com relação ao emprego da abordagem de customização porque a responsabilidade de adaptação do código fonte produzido por eles dentro da estrutura criada pela abordagem é de um "gestor de desenvolvimento". Para pequenas equipes acredita-se que o papel de "gestor" pode ser compartilhado entre desenvolvedores, seja simultaneamente, como em um desenvolvimento em pares, ou seja em forma de revezamento de papéis.

Esse *workflow* baseia-se no método empregado na realização do estudo (Seção 6.2) adaptado ao cenário real de desenvolvimento dentro do IFSULDEMINAS e nas sugestões realizadas pelos participantes. A descrição do *workflow* inicia-se com a análise da Figura 6.3.

Figura 6.3 – Realizar desenvolvimento local.



Fonte: Do autor (2021).

Na Figura 6.3, visualiza-se a existência de dois papéis no processo de desenvolvimento: o desenvolvedor e o gestor de desenvolvimento. O primeiro realiza seu trabalho de forma

muita próxima ao processo corriqueiro atual. O segundo é a figura responsável por atividades diretamente relacionadas à abordagem proposta nesta dissertação de mestrado.

Ao iniciar uma nova atividade de desenvolvimento, o desenvolvedor deve solicitar ao gestor um *branch* de desenvolvimento atualizado frente ao projeto original. Para isso, o gestor executa a atividade "Sincronização com a origem", descrita adiante, obtendo ao seu final o *branch* de desenvolvimento devidamente atualizado. Esse *branch*, inclusive, é candidato apto para ser colocado em produção, observadas as demais etapas de testes e validações.

De posse do *branch* atualizado, o desenvolvedor inicia o atendimento de uma nova demanda de desenvolvimento tal como o faz hoje, trabalhando sobre o código em sua máquina local. Após o término das etapas necessárias ao desenvolvimento, incluindo testes locais e validações junto a usuários, o desenvolvedor disponibiliza ao gestor o *branch* de trabalho que representa o sistema customizado que roda em produção somado às modificações por ele realizadas, necessárias ao atendimento da demanda específica. Nesse ponto, inicia-se o trabalho do gestor, similar ao realizado no estudo descrito no Capítulo 6.

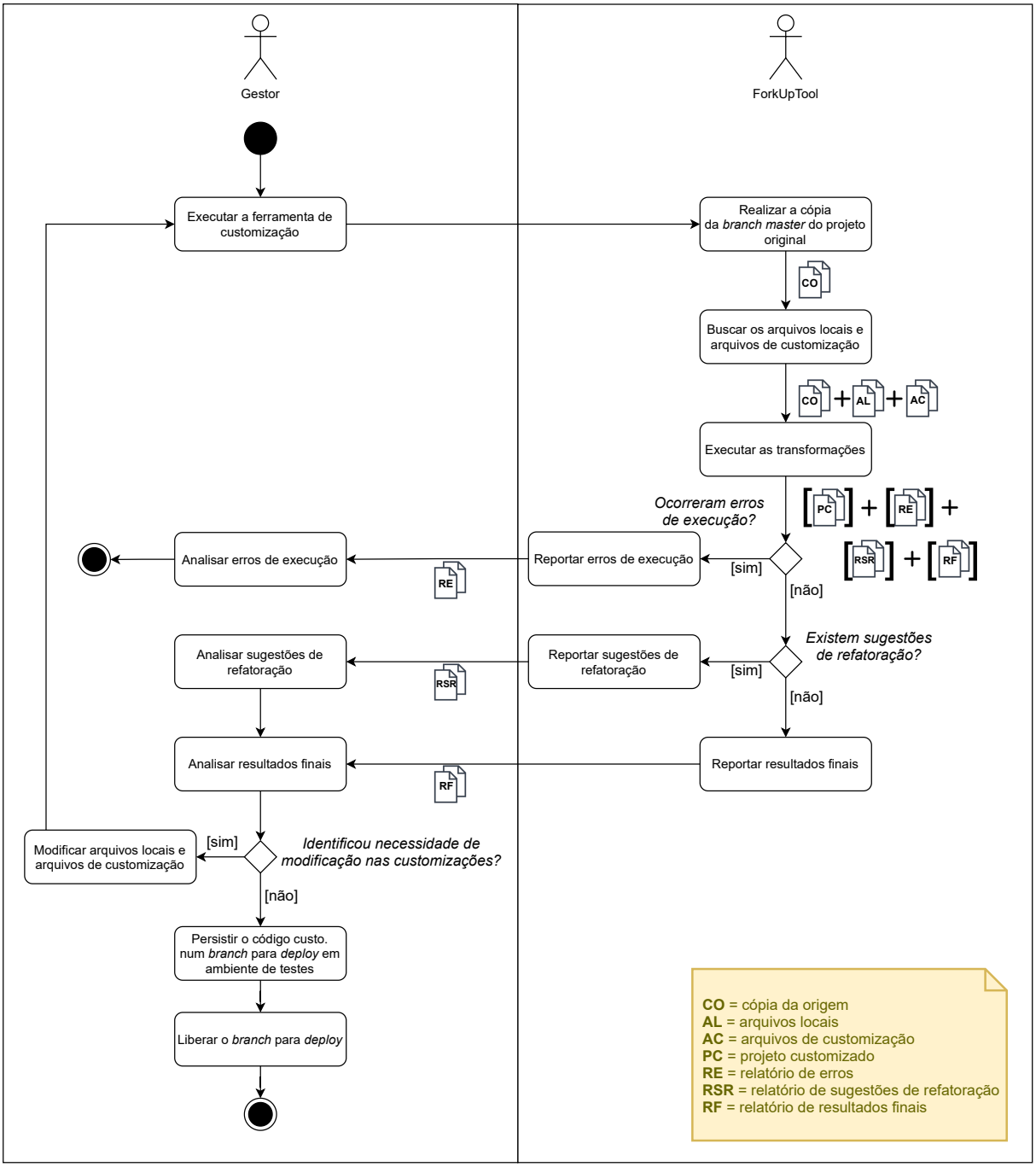
O gestor identifica as modificações realizadas pelo desenvolvedor no código fonte do projeto e faz um *code review* que ajuda a promover a qualidade. Nesse momento problemas podem ser identificados e encaminhados ao desenvolvedor para correção. Em seguida, o gestor modifica os arquivos de customização e também os arquivos locais, se necessário. Após isso, o gestor executa novamente a atividade "Sincronização com a origem" representada na Figura 6.4.

A primeira atividade realizada pelo gestor, tal como ilustra a Figura 6.4, é executar a ferramenta de customização. A ferramenta realiza o seu papel, tal como descrito no Capítulo 4: realiza uma cópia da origem (**CO**), busca por arquivos de customização (**AC**) e arquivos locais (**AL**) e realiza as transformações necessárias. Ao final de sua execução pode produzir como resultado:

- o código fonte do projeto customizado (**PC**), se não ocorrerem erros críticos de execução;
- um relatório de erros gerais (**RE**), se identificados;
- um relatório de sugestões de refatoração (**RSR**), se identificadas;
- um relatório de resultados finais (**RF**), se não interrompido por erros críticos.

De posse dessas informações, cabe ao gestor de desenvolvimento avaliar se existem modificações necessárias a serem realizadas nos arquivos de customização e nos arquivos locais.

Figura 6.4 – Sincronização com a origem.



Fonte: Do autor (2021).

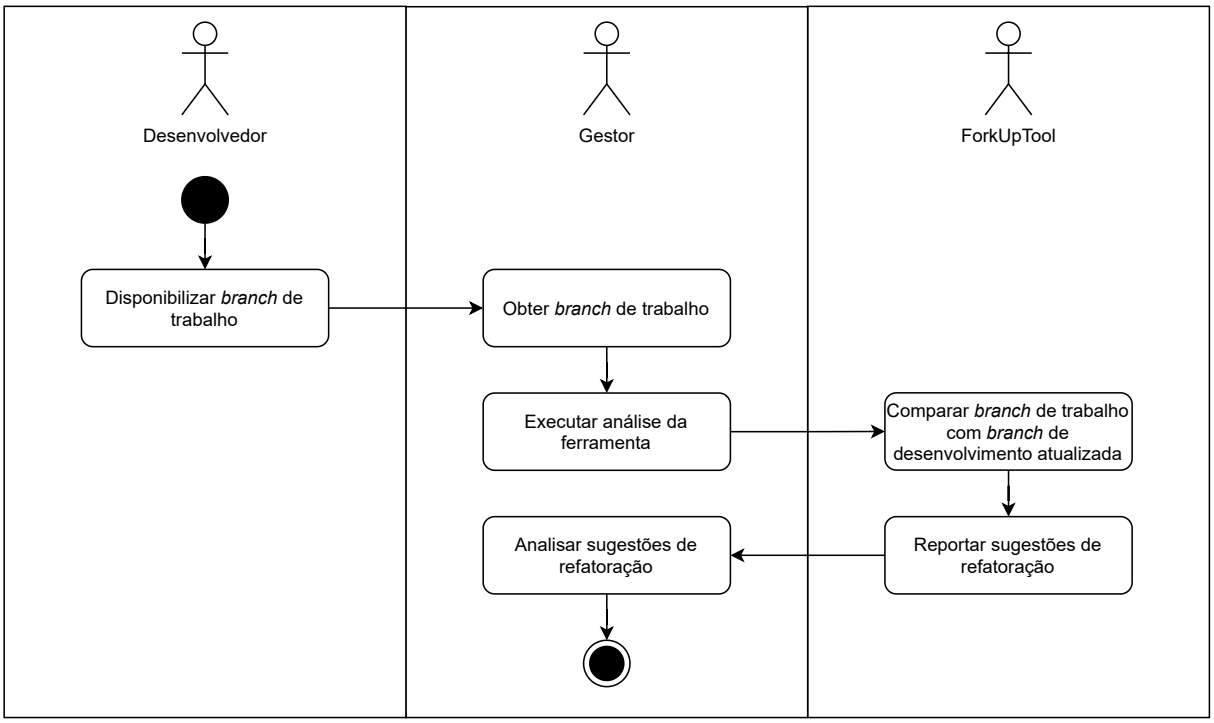
Havendo necessidade, o gestor deve realizá-las manualmente e retomar o processo, desde seu início (atividade "executar ferramenta de customização"). Não havendo necessidades de modificação, o gestor persiste o **PC** em um *branch* para *deploy* e libera esse *branch* para continuidade de outras atividades.

Deve-se notar que nessa proposta de *workflow* de desenvolvimento há pouco impacto no trabalho atual dos desenvolvedores. Eles necessitam estar cientes da abordagem de custo-

mização e de seu funcionamento, mas cabe ao gestor as atividades diretamente relacionados a essa abordagem. Com relação às atividades descritas e funcionalidades da ferramenta, apenas o "relatório de sugestões de refatoração" é inexistente, mas será pontuado como trabalho futuro.

Trazendo para essa proposta de *workflow* de desenvolvimento algumas sugestões propostas pelos participantes do estudo, em particular do desenvolvedor D2, dois facilitadores foram imaginados para ampliar a atuação da ferramenta e, consequentemente, contribuir para a evolução da abordagem como um todo. São facilitadores hoje inexistentes no protótipo da ferramenta, mas que podem ser tomados como necessidade de trabalho futuro. Esses facilitadores estão diretamente relacionados, mas um não depende do outro. O primeiro deles – "Identificar modificações realizadas" – é visualizado na Figura 6.5.

Figura 6.5 – Identificar modificações realizadas.

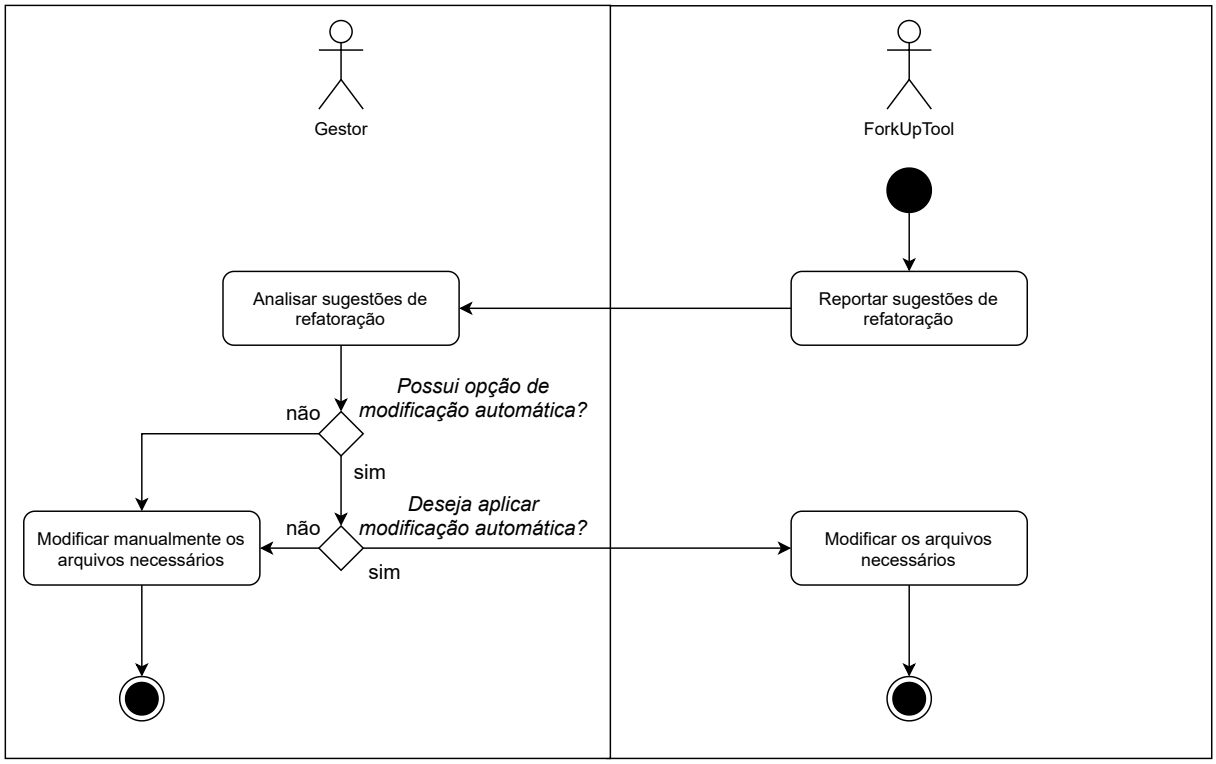


Fonte: Do autor (2021).

Esse facilitador procura automatizar parte do trabalho do gestor de desenvolvimento com relação à atividade "Identificar as modificações realizadas" (Figura 6.3). Após obter o *branch* de trabalho, o gestor, ao invés de identificar tudo o que foi modificado pelo desenvolvedor de forma manual, executa a opção de análise da ForkUpTool. É a ferramenta que identifica as mudanças e reporta as sugestões de refatoração, que então serão analisadas pelo gestor.

O segundo facilitador – “Analisar sugestões de refatoração” – é apresentado na Figura 6.6. Ele prevê a criação de sugestões de refatoração “automáticas” para a ferramenta. Isso vai diretamente ao encontro do que o desenvolvedor D2 sugeriu em resposta à questão de pesquisa QP#4 (Seção 6.3.4).

Figura 6.6 – Analisar sugestões de refatoração.



Fonte: Do autor (2021).

Nesse facilitador, a ferramenta identifica algumas situações onde pode atuar de forma automática, apresentando essa opção ao gestor de desenvolvimento. Exemplo: o desenvolvedor, ao atender determinada demanda de desenvolvimento, realizou uma série de modificações em determinado arquivo de renderização do tipo HTML. A ferramenta pode sugerir ao gestor a aplicação de uma refatoração de customização automática, que seria a criação de uma instrução **`replace file`** para o arquivo em questão. As situações passíveis de proposta de resolução automatizada necessitam ser mapeadas, mas representariam potencial melhoria para a ferramenta.

O *workflow* em questão será melhor avaliado no futuro e será proposto como parte das etapas necessárias para adoção da abordagem em projetos customizados em andamento.

6.6 Ameaças à validade

É possível identificar três ameaças à validade para este estudo.

Validade de conclusão: A quantidade reduzida de participantes pode afetar os resultados observados. No entanto, buscou-se avaliar não apenas as respostas fechadas em escala *Likert*, mas principalmente as respostas abertas, das quais foram retiradas contribuições para este trabalho.

Validade de construção: Sendo o pesquisador um interessado direto nos resultados da pesquisa, pode ter ocorrido um viés na análise dos resultados em decorrência de suas expectativas. Para amenizar essa ameaça, um segundo pesquisador avaliou os resultados e ponderou sobre limitações da abordagem, descritas ao longo do texto e particularmente na Seção 8.2.

Validade externa: A avaliação tratou apenas de um sistema de software, escrito em uma linguagem específica, o que prejudica a generalização dos resultados. Todavia, tomou-se como base um sistema do mundo real e a avaliação foi realizada o mais próximo possível da realidade de trabalho dos profissionais envolvidos.

6.7 Considerações Finais

Neste capítulo, foi avaliada a perspectiva dos desenvolvedores de um projeto de software quanto à abordagem proposta. Quatro objetivos foram definidos para a avaliação: (i) identificar se a abordagem seria aceita pelos desenvolvedores; (ii) se os desenvolvedores veem vantagem ao utilizá-la; (iii) explicar aos desenvolvedores que é possível implementar *features* reais com a abordagem; e (iv) demostrar aos desenvolvedores sua aplicabilidade em cenários concretos.

Os participantes envolvidos trabalhavam ativamente no projeto no momento em que a avaliação foi conduzida. Todos são graduados na área de Computação e metade tem mestrado também nessa área. A maioria (75%) possui mais de 6 anos de experiência em desenvolvimento de software e dois dos quatro participantes possuem de 4 a 6 anos de experiência em desenvolvimento com `Python/Django` e no sistema real avaliado.

A avaliação envolveu a programação de quatro *features* reais do *backlog* de desenvolvimento do sistema. Cada desenvolvedor desenvolveu sua *feature*, que depois foi implementada pelo pesquisador utilizando a abordagem proposta no estudo. Uma entrevista orientada foi con-

duzida junto aos participantes que responderam a cinco questões formuladas com o objetivo de alcançar os quatro objetivos estipulados para a avaliação.

Na QP#1, pôde-se constatar que todos os participantes entenderam como suas *features* foram implementadas utilizando a abordagem. As respostas da QP#2 mostraram que todos acharam a abordagem, no mínimo, interessante, destacando a relevância do tema e a necessidade de novos experimentos. Na QP#3, todos concordaram que a abordagem possui grandes chances de funcionar quando aplicada desde o início de um novo projeto. Na QP#4, não houve segurança dos participantes em realizar tal afirmação pois, no cenário de projetos em andamento, existem diversas questões a serem consideradas, tal como adaptações ao uso da abordagem, mais conhecimento da abordagem, necessidade de tratativa do volume de modificações realizadas no projeto customizado, melhorias na ferramenta e necessidade de realização de novos estudos. Por fim, na QP#5, os participantes afirmaram que desenvolver de forma desacoplada pode ser desconfortável no início, mas acostuma-se. Citaram que a adoção da abordagem de customização assemelha-se à adoção de outras tecnologias, como Git e Docker, pois no início todos sentem-se desconfortáveis, mas acostumam-se com o tempo. Destacaram também que a adoção de novas tecnologias deve ser acordada entre todos, ou seja, o "acostuma-se" deve ser consensual e não impositivo.

Por fim, um *workflow* de desenvolvimento foi proposto com base nos passos realizados no estudo, adaptados ao cenário real de desenvolvimento da instituição avaliada, agregando as sugestões dos participantes. Nesse *workflow*, há pouca ou nenhuma mudança no fluxo do processo de desenvolvimento com relação ao desenvolvedor. Todas as ações necessárias para adaptação à abordagem de customização são realizadas por um "gestor de desenvolvimento". Foram previstos ainda melhorias no fluxo com base em sugestões dos participantes, que podem melhorar a aceitação da abordagem e promover sua adoção.

7 TRABALHOS RELACIONADOS

Os trabalhos relacionados foram divididos em cinco grupos: (i) recursos do `Git`, que procuram auxiliar no processo de mesclagem e resolução de conflitos e podem ser confundidos com a abordagem proposta nesta dissertação; (ii) previsão ou antecipação de conflitos, que tentam antecipar ações para evitar a ocorrência de conflitos ou evitar que se tornem mais difíceis de resolver; (iii) apoio na resolução de conflitos, que encaram os conflitos de mesclagem como inevitáveis e buscam apoiar a sua resolução utilizando diferentes abordagens; (iv) busca de conhecimento, que procuram maior conhecimento empírico sobre a ocorrência de conflitos e a tratativa dada pelos desenvolvedores; e (v) estudos de caso de mesclagem entre projeto original e projeto customizado, que buscam entender e descrever a natureza dos conflitos de mesclagem que ocorrem nesse cenário, investigando ainda a possibilidade de suporte automatizado para correção desses conflitos.

Recursos do Git: A opção *ours* disponível no *git merge* (CHACON; STRAUB, 2014) – que sinaliza ao `Git` que, na ocorrência de conflitos, deve-se optar pelo lado *ours* do par de arquivos conflitantes – assemelha-se à abordagem proposta nesta dissertação. No entanto, cabe ressaltar algumas vantagens da abordagem proposta: (i) não se aplica exclusivamente a repositórios `Git`; (ii) "*ours*" força a escolha de um "lado" no momento do *merge* (*ours* ou *theirs*) para **todo** o *merge* realizado, i.e., todos os arquivos que eventualmente resultem em conflito de mesclagem, e nos pares de arquivos em conflito em **todos** os trechos conflitantes, enquanto que a abordagem proposta permite especificar pela implementação local (*ours*) em diferentes níveis de granularidade, e.g., para um método ou um trecho específico; e (iii) a abordagem proposta não propõe evitar apenas a ocorrência de conflitos, mas promover modularidade, documentação, rastreabilidade e reúso.

O *git rerere* (CHACON; STRAUB, 2014), acrônimo de *reuse recorded resolution* ("reutilizar resolução gravada"), permite ao `Git` buscar no histórico de mesclagens bem sucedidas soluções de conflitos para situações semelhantes, evitando solicitar nova resolução manual para um mesmo cenário. Apesar de benéfico, é um recurso com escopo limitado e sua semântica é completamente diferente da semântica da abordagem de customização proposta nesta dissertação.

O *git patch* (CHACON; STRAUB, 2014) é um recurso que permite a geração de "arquivos de *patch*" que nada mais são que representações das alterações realizadas por um

commit específico. Portanto, sua semântica difere da semântica dos arquivos de customização desta dissertação. Além disso, a aplicação de um arquivo de *patch* do `Git` pode resultar nos mesmos conflitos de uma mesclagem entre *branches* realizada com `git merge`.

Previsão ou antecipação de conflitos. Guimarães e Silva (GUIMARÃES; SILVA, 2012) afirmam que a detecção antecipada de conflitos permite aos desenvolvedores de software atuar de forma pró-ativa sobre o código fonte, fazendo com que sua resolução ocorra de forma mais fácil quando comparada à resolução de um conflito depois do *merge*.

Os autores citam que as soluções propostas na literatura focam em abordagens para conscientizar desenvolvedores sobre as atividades paralelas realizadas por colegas de trabalho sobre o mesmo código fonte. O objetivo seria permiti-los agir de forma antecipada para evitar os conflitos. Contudo, essa forma de conscientização acaba exigindo que os desenvolvedores trabalhem sozinhos na tarefa de resolução de problemas. Destacam ainda que esse artifício pode sobrecarregá-los com notificações indesejadas, atrapalhando mais ainda o fluxo de desenvolvimento.

Em seu artigo, os autores propõem uma nova solução que mescla continuamente os trabalhos ainda não confirmados em um sistema em segundo plano, de tal forma a detectar os conflitos de forma antecipada. Essa mesclagem ocorre de forma automática enquanto os desenvolvedores trabalham e esses podem resolver eventuais conflitos ainda enquanto programam. Com o estudo empírico, afirmam que a solução proposta evita sobrecarregá-los e melhora a detecção antecipada frente a outras abordagens existentes. Introduzem o conceito de "mesclagem contínua" implementando dentro da IDE Eclipse um *plug-in* com esse propósito.

Apesar das vantagens levantadas pelos autores do estudo, sua aplicabilidade fica restrita a cenários onde existe uma equipe única de desenvolvimento ou equipes diferentes sob mesma coordenação, trabalhando em um mesmo projeto de software. Pressupõe-se que o trabalho de desenvolvimento deve ocorrer de forma coordenada, pois a proposta é apresentada, construída e avaliada sobre uma linguagem de programação específica (Java) fazendo uso de uma IDE específica (Eclipse).

Ideia semelhante com relação à minimização de conflitos de mesclagens envolve o trabalho de Kasi e Sarma (2013). Segundo os autores, conflitos de software são regulares e os estudos atuais que promovem o reconhecimento antecipado permitem tratar conflitos o quanto antes, enquanto ainda são pequenos e mais fáceis de resolver. Mas, na prática, os conflitos continuam ocorrendo e demandam tempo de desenvolvimento para resolução.

Kasi e Sarma (2013) apresentam uma nova técnica de minimização de conflitos que procura identificá-los de forma pró ativa, criando restrições e recomendações de diferentes caminhos de desenvolvimento para uma equipe de trabalho de tal forma a evitar conflitos. A ideia é agrupar as mudanças previstas para um projeto em tarefas, avaliar restrições de tarefas e recomendar uma ordem de realização de tarefas ótimas para cada desenvolvedor, ou seja, identificar tarefas conflitantes, que irão conflitar quando executadas em paralelo, e agendá-las adequadamente recomendando "caminhos livres de conflitos". Um sistema é desenvolvido para esse fim – o Cassandra – que permite o agendamento das tarefas tratando conflitos em nível de tarefa, não de código.

Observa-se nesse trabalho uma "troca" na fonte de problemas. Conflitos de mesclagem são evitados ao criar "caminhos livres de conflito" para os desenvolvedores, mas os conflitos agora podem ocorrer no nível de tarefas. Nem sempre é possível garantir que esses conflitos são mais fáceis de resolver ou que tomam menos tempo para resolução que os primeiros. Além disso, parte-se do pressuposto que conflitos de *merge* ocorreriam em determinado momento, mas não se pode afirmar isso com absoluta certeza em todas as situações, uma vez que a ocorrência de conflitos de *merge* se dá em função da combinação de diversos fatores (trabalho paralelo no mesmo código fonte, ordem de execução e cronologia de comandos *pull* e *push*).

Os estudos deste grupo (GUIMARÃES; SILVA, 2012; KASI; SARMA, 2013) não se aplicam ao cenário de atualização de *forks* apresentado nesta dissertação. A mesclagem de código, nesse cenário, envolve trabalhos persistidos em *branches* de produção pertencentes a projetos distintos, ou seja, códigos finalizados. Os estudos apresentados abordam estratégias para códigos em desenvolvimento, pertencentes a um mesmo projeto.

Apoio na resolução de conflitos. O foco dado ao problema de conflitos de mesclagem apresentado por Nishimura e Maruyama (2016) difere dos anteriores. Segundo os autores, boa parte dos estudos na área preocupam-se em propor ferramentas para incrementar os sistemas de controle de versão, responsáveis por orquestrar o desenvolvimento de software paralelo e concorrente, dando-lhes capacidade de detecção e consciência antecipada de conflitos. Apesar de parecer razoável, acreditam que não apenas isso é importante para os desenvolvedores, mas é importante entender como os fragmentos de código conflitantes foram alterados no passado.

A ideia central é as informações sobre as alterações de código granuladas "por detrás" de conflitos de mesclagem serem úteis para reconciliá-las. Propõem então uma nova ferramenta – o MergeHelper – que explora o histórico de operações de edição do código-fonte para apoiar

a resolução de conflitos de mesclagem. Edições gravadas pelo ChangeTracker (módulo do MergeHelper) são persistidas no repositório da aplicação pelo EGit (*plug-in git* do Eclipse). A ferramenta provê funcionalidades dentro da IDE de desenvolvimento (nesse caso, o Eclipse) de forma a facilitar a tarefa de resolução de conflitos de mesclagem, explorando o histórico de edições que ela gera e guarda ao longo do tempo.

Um estudo empírico é conduzido pelos autores e, ao final, conclui-se que os resultados são satisfatórios ao empregar o MergeHelper para solução de conflitos de código. Contudo, destacam-se pontos de ameaça à validade, como (i) limitação de análise de diferentes ramos de trabalho: o sistema MergeHelper só consegue trabalhar com duas *branches* ao mesmo tempo; (ii) limitação quanto à linguagem de programação utilizada (nesse caso, Java) e a IDE de desenvolvimento utilizada (nesse caso, o Eclipse); e (iii) não observância de conflitos semânticos, que podem causar comportamento indesejado ou a falha de teste do código resultante da mesclagem.

Similarmente, Costa et al. (COSTA et al., 2016) propõem o TIPMerge, cujo propósito é identificar e recomendar os desenvolvedores mais aptos a atuar na resolução de conflitos de mesclagem, com base em sua experiência passada dentro do projeto, nas alterações por eles realizadas e nas dependências com relação aos arquivos envolvidos.

O trabalho de Costa et al. (2016) tem um objetivo muito próximo ao apresentado por Nishimura e Maruyama (2016): minimizar o impacto negativo dos conflitos de mesclagem, depois que ocorreram. Percebe-se nesse ponto uma diferença quanto aos dois trabalhos citados no grupo anterior, que tinham como principal foco tentar antecipar ou evitar a ocorrência de conflitos.

A identificação dos desenvolvedores mais aptos para resolução de conflitos de software pode ser tão difícil quanto à resolução dos próprios conflitos (COSTA et al., 2016). Logo, o trabalho desenvolvido no estudo justifica-se como contribuição importante na área, pois a identificação desses profissionais, levando-se em consideração as características do conflito em particular, como artefatos envolvidos e o histórico de *branches* e contribuições, não seria viável se realizada manualmente.

Ambos os artigos supracitados (NISHIMURA; MARUYAMA, 2016; COSTA et al., 2016) não se aplicam ao cenário proposto nesta dissertação. O primeiro baseia-se em um histórico de operações de edição construído ao longo do processo de desenvolvimento, que não existe no cenário de mesclagem de *forks*. O segundo necessita de um histórico de

desenvolvimento para indicar o desenvolvedor mais apto para resolução de um conflito. Mesmo que esse histórico pudesse ser reconstituído no cenário em questão, a estratégia poderia falhar ao indicar um desenvolvedor da equipe do projeto de origem para resolução de um conflito, e não do projeto customizado.

Busca de conhecimento. McKee et al. (2017) alegam que há pouco conhecimento empírico sobre como os profissionais realmente abordam e executam a resolução de conflitos de mesclagem. Sem esse conhecimento, ferramentas podem estar se baseando em suposições erradas ao implementar soluções e os pesquisadores podem perder oportunidades de melhorar o estado da arte.

Com o objetivo de buscar esse conhecimento, um estudo junto aos profissionais da área foi realizado. Entrevistas semi-estruturadas foram conduzidas junto a 10 profissionais de 7 diferentes empresas, em projetos *open-source* e de mercado. Conceitos e percepções chave foram identificados e depois validados por meio de pesquisas com outros 162 profissionais.

Os autores puderam identificar quais os profissionais são impactados diretamente pela percepção própria/pessoal de quão complexo é o código conflitante. Essa percepção pode alterar o tempo para resolução dos conflitos, pois muitos profissionais optam por adiá-la, e ainda pode alterar o método empregado para resolução.

Para McKee et al. (2017), as percepções dos praticantes alteram o impacto de ferramentas e processos projetados para resolver de forma preventiva e eficiente os conflitos de mesclagem. Entender se os profissionais reagiram de acordo com os casos de uso estabelecidos é importante ao criar ferramentas destinadas a esse fim.

Accioly, Borba e Cavalcanti (2018) também priorizam a busca de conhecimento sobre os conflitos de mesclagem em seu trabalho. Segundo os autores, o entendimento da estrutura de mudanças que levam aos conflitos de mesclagem pode esclarecer como evitá-los.

Os autores catalogaram nove padrões de conflitos em termos de mudanças de estrutura de código que os ocasionaram. O estudo empírico por eles realizado abrangeu mais de 70.000 fusões de código em 123 projetos Java de código aberto, disponíveis no GitHub.

Dentre suas contribuições, os autores constataram: (i) que a probabilidade de criar um conflito é a mesma ao editar um método, campos de uma classe, etc.; (ii) que os desenvolvedores editam de forma independente as mesmas linhas ou linhas consecutivas do mesmo método, de forma concomitante; (iii) que conflitos em geral envolvem normalmente mais de

dois desenvolvedores; e (iv) que o hábito de copiar e colar trechos de código em diferentes repositórios também causam conflitos de mesclagem.

Estudos de caso de mesclagem entre projeto original e projeto customizado. Mahmoudi e Nadi (2018) conduzem um estudo empírico em que analisam como fabricantes de *smartphones* utilizam o Android como sistema operacional subjacente e o estendem para adicionar novas funcionalidades e torná-lo compatível com equipamentos de *hardware* específicos. Em especial, buscam entender a natureza das mudanças que ocorrem nos projetos customizados e os desafios de mesclagem de código ou replicação de personalizações quando uma nova versão do Android é lançada.

Os autores analisaram oito versões distintas de uma variante de código aberto do Android chamada LineageOS. Com base na semântica das mudanças, categorizam se as mudanças sobrepostas têm o potencial de serem ou não integradas automaticamente. O objetivo é entender a natureza das mudanças e investigar a possibilidade de ter suporte automatizado para a mesclagem do código fonte do projeto customizado com o código atualizado do projeto original. Como resultado mais relevante, identificam que 56% das alterações no LineageOS têm potencial para serem automatizadas com segurança. Contudo, não implementam uma ferramenta ou especificam qualquer algoritmo que realize tal automatização.

O trabalho de Sung et al. (2020) apresenta avanços com relação ao anterior. Os autores descrevem a natureza dos conflitos de mesclagem que surgem por causa das mesclagens do Microsoft Edge com sua origem, o Chromium. Mas, além disso, investigam a viabilidade de corrigir automaticamente determinada classe de conflitos relacionados às quebras de compilação, implementando uma ferramenta com esse propósito.

Na primeira parte do estudo de caso industrial, Sung et al. (2020) descrevem a natureza dos conflitos de mesclagem que surgem quando o projeto do Microsoft Edge é mesclado com sua origem. Classificam os conflitos em três grandes grupos: (i) conflitos textuais, (ii) quebras de compilação e (iii) falhas de testes. Sub categorias dentro de cada classe são especificadas e analisadas.

Na segunda parte do estudo, Sung et al. (2020) tomam para análise uma sub classe de conflitos de quebra de compilação – falhas estruturais em arquivos C++ – e investigam a viabilidade de corrigi-las automaticamente. Implementam uma ferramenta que realiza a diferenciação de AST dos dois projetos envolvidos para identificar mudanças na origem e criar *patches* que podem ser aplicados no projeto customizado. Concluem, após a análise de três meses de dados

reais de desenvolvimento, que cerca de 40% das quebras de compilação podem ser reparadas automaticamente.

Os trabalhos apresentados nesse grupo são os que mais se aproximam do propósito da abordagem de customização descrita nesta dissertação. Contudo, focam na análise e resolução de conflitos de mesclagem ao passo que esta abordagem procura evitá-los.

A análise dos artigos citados evidencia a preocupação presente na literatura em estudar e propor meios para lidar com os problemas que envolvem o merge de códigos e a resolução de conflitos. Alguns estudos trabalham sobre a detecção antecipada (GUIMARÃES; SILVA, 2012) (KASI; SARMA, 2013), outros abordam meios para facilitar a resolução de conflitos, depois que ocorreram (COSTA et al., 2016) (NISHIMURA; MARUYAMA, 2016) e existem ainda trabalhos que se preocupam em analisar como os profissionais lidam com problemas de resolução de conflitos no dia a dia para, a partir desse conhecimento, direcionar novos estudos (MCKEE et al., 2017), ou ainda que mudanças estruturais levam ao surgimento desse conflitos (ACCIOLY; BORBA; CAVALCANTI, 2018). Poucos, contudo, tratam do cenário específico tratado por esta dissertação (SUNG et al., 2020), (MAHMOUDI; NADI, 2018), onde um sistema customizado, criado a partir do *forking* de um projeto original, e que possui customizações e implementações próprias, necessita ser atualizado constantemente com o código fonte da origem, ocasionando inúmeros problemas de conflito de *merge*. Esses trabalhos, todavia, não apresentam soluções de forma abrangente, genérica, tal como esta abordagem se propõe. Nesta dissertação, é proposta uma solução que promove documentação, rastreabilidade e reúso, além de evitar tais conflitos, baseada em uma nova abordagem para customização de funcionalidades *forks* de sistemas em constante evolução. Trata-se, portanto, de um enfoque novo e diferente dos trabalhos citados e outros estudos encontrados na literatura.

8 CONCLUSÃO

O *problema* abordado nesta dissertação de mestrado é manter sistemas customizados atualizados frente aos projetos originais.

Apesar de serem independentes e possuírem evolução própria, é interessante os projetos customizados estarem atualizados frente aos respectivos projetos originais. Obter a partir do código fonte atualizado do projeto original novas *features*, correções de *bugs* e melhorias de desempenho implementadas e testadas pode significar economia de tempo e de recursos ao projeto customizado (STĂNCIULESCU; SCHULZE; WĄSOWSKI, 2015). Essa ação, contudo, implica em mesclagem de código e, consequentemente, na ocorrência de conflitos de mesclagem.

A mesclagem do código do projeto original no projeto customizado pode gerar inúmeros conflitos por se tratar de dois projetos independentes com diferentes equipes trabalhando de forma paralela e, mais relevante, a equipe do projeto original sequer tem conhecimento do que vem sendo customizado em seus *forks*, da mesma forma que as mudanças na origem ocorrem sem conhecimento dos projetos customizados (SUNG et al., 2020).

Embora seja algo rotineiro no desenvolvimento de software, a resolução de conflitos de mesclagem pode ser considerada tarefa árdua para desenvolvedores (NISHIMURA; MARUYAMA, 2016) pois normalmente exigem solução dispendiosa (GUIMARÃES; SILVA, 2012) e, quando mal executada, leva a erros de integração ou interrupção de fluxo de trabalho comprometendo a eficiência de projetos (MCKEE et al., 2017).

Diante disso, esta dissertação propôs, implementou e avaliou uma abordagem para atualização de *forks* frente ao projeto original. A ideia central da abordagem proposta é o projeto customizado ser desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original". Desta forma, é possível obter a qualquer tempo uma cópia atualizada do projeto original e aplicar nessa cópia um conjunto de alterações, descritas por arquivos de customização, que possibilitam obter, por meio da execução da ferramenta criada para esse fim, o sistema customizado e atualizado frente ao projeto original.

A abordagem mostrou-se eficaz ao evitar conflitos de mesclagem no escopo do estudo realizado e apresentou, de forma geral, boa aceitação por parte de usuários desenvolvedores. Em uma avaliação histórica, foi possível constatar que a abordagem, aplicada ao cenário de atualização de um sistema de software, evitaria 752 conflitos que ocorreram durante quatro anos de atualização desse projeto. Em uma avaliação junto aos usuários, observou-se boa aceitação

da abordagem proposta, com análise controlada de cenários reais de desenvolvimento. Foi possível explicar aos desenvolvedores como implementar *features* reais com a abordagem e mostrar sua aplicabilidade em cenários concretos. Todos os participantes acharam a abordagem interessante e acreditam ser possível aplicá-la no mundo real, principalmente quando adotada desde o início de um novo projeto, desde que observadas suas limitações e necessidade de adaptações.

O restante deste capítulo está organizado da seguinte forma. Na Seção 8.1, são enumeradas as contribuições desta dissertação de mestrado. Em seguida, na Seção 8.2, são pontuadas as limitações deste estudo. A Seção 8.3 trata das dificuldades e desafios encontrados durante o estudo. Por fim, na Seção 8.4, são apresentados os trabalhos futuros.

8.1 Contribuições

Este projeto de pesquisa realizou as seguintes contribuições:

- A proposta de uma nova abordagem para atualização de sistemas criados a partir do *forking* de outros sistemas que permite mantê-los atualizados frente aos respectivos projetos originais;
- A especificação de uma DSL com três diretivas e onze instruções de customização que se mostraram eficazes em realizar as customizações de um projeto durante um período de quatro anos;
- Identificação e discussão de *conflitos de customização* que devem ser considerados e tratados ao empregar a abordagem proposta;
- Uma ferramenta de customização que aplica as transformações descritas nos arquivos de customização sobre a cópia do código fonte do projeto original, de tal forma a torná-lo o projeto customizado, além da implementação de outros recursos na ferramenta que permitem a análise dos repositórios;
- Uma avaliação histórica do emprego da abordagem sobre um sistema de software, para um período de quatro anos, onde 752 trechos de código em conflito seriam evitados;
- Uma avaliação junto a desenvolvedores, onde foi possível constatar uma boa aceitação para a abordagem proposta;

- Redução de conflitos, pois o código do projeto original não é mais alterado diretamente;
- Modularidade, uma vez que customizações de uma *feature* encontram-se especificadas em um único arquivo;
- Documentação, pois informações relevantes de cada *feature* estão nos arquivos de customização, assim como as dependências;
- Rastreabilidade, pois é possível rastrear a *feature* a qual um trecho de código pertence;
- Reúso, pois basta fornecer o arquivo de customização a um outro projeto customizado.

8.2 Limitações

Este projeto de pesquisa tem as seguintes limitações:

- Não se aplica a cenários diferentes daquele a que a abordagem se propõe, ou seja, atualização de sistemas criados a partir do *forking* frente aos projetos que lhes deram origem; e
- Tanto a ferramenta de customização criada quanto as avaliações realizadas contemplaram apenas sistemas escritos em Python com uso do *framework* Django.

8.3 Dificuldades e desafios

No desenvolvimento deste projeto de pesquisa algumas dificuldades e desafios foram enfrentados e implicaram na adoção de estratégias de trabalho nem sempre ideais, mas realistas frente às limitações existentes.

Mensuração de esforço: a necessidade de comparar o esforço para adoção da abordagem frente ao esforço para resolução de manual de conflitos foi destacada pelos participantes da avaliação do Capítulo 6 como importante para aceitação da mesma. É uma necessidade que vem à tona desde o primeiro contato com a abordagem. Embora seja um ponto relevante, trata-se de objeto de trabalho futuro, pois envolve maior tempo de pesquisa, maior quantidade de pesquisadores no projeto e maior tempo de colaboração e dedicação de participantes de experimentos. No cenário desta pesquisa, apenas um pesquisador esteve envolvido e a quantidade de participantes externos foi limitado às possibilidades de participação da instituição colaboradora (IFSULDEMINAS).

Avaliação junto aos desenvolvedores: Cenários alternativos para condução do experimento descrito no Capítulo 6 foram considerados, como entregar uma mesma *feature* a dois participantes diferentes, sendo que um conduz o desenvolvimento de forma tradicional e o outro conduz com aplicação da abordagem. Contudo, tal linha de trabalho implica em mais tempo de envolvimento dos participantes, em especial para treinamento prévio com relação à aplicação da abordagem. Para uma equipe de desenvolvimento reduzida, um maior tempo de envolvimento torna-se prejudicial, somado ao fato de que dois desenvolvedores estarem trabalhando sobre uma mesma *feature* implica em parar outras demandas de desenvolvimento do *backlog* do IFSULDEMINAS. Aliado às dificuldades expostas, deve-se considerar ainda as limitações impostas pelo cenário mundial da pandemia de Covid19 que limita o tempo de trabalho e a realização de atividades presenciais.

Conflitos de customização: embora uma taxa da ocorrência de conflitos de customização não tenha sido levantada, a expectativa é que a resolução desses conflitos seja menos custosa do que a resolução de conflitos de mesclagem, uma vez que o desenvolvedor consegue rastrear os trechos de códigos envolvidos e a qual *feature* eles pertencem. Isso promove maior manutenibilidade a médio e longo prazo. Conjectura-se que o esforço dispensado no início de adoção da abordagem com a escrita de arquivos de customização e arquivos locais é compensado ao longo do tempo com melhorias na manutenção do código. Além disso, a abordagem é uma solução mais elegante, porque não é invasiva e não necessita de qualquer intervenção ou ação junto ao projeto original, mantendo a independência do projeto customizado.

8.4 Trabalhos futuros

Trabalhos futuros incluem:

- Realizar a avaliação da abordagem em um projeto customizado desde o início do desenvolvimento;
- Criar instruções de granularidade fina para arquivos JavaScript, CSS, HTML, etc.;
- Definir um conjunto de indicadores que auxiliem na recomendação de refatorações em arquivos de customização existentes, por exemplo, existe um **`replace class`** embora apenas um método é de fato alterado;

- Realizar avaliações em sistemas escritos em outras linguagens de programação, de modo a verificar se a abordagem funciona em contextos mais abrangentes, i.e., avaliar a extrapolação do resultado de uma pesquisa do tipo *design research*;
- Implementar versões da ferramenta que manipulem código em diferentes linguagens de programação, avaliando o uso do ANTLR4;
- Avaliar, propor melhorias e realizar adaptações no *workflow* de desenvolvimento apresentado na Seção 6.5;
- Estender as funcionalidades da ferramenta de customização de tal modo a adaptá-la às melhorias elencadas para o *workflow* da Seção 6.5;
- Avaliar a viabilidade de inserção da abordagem em um processo de desenvolvimento, considerando o custo-benefício.

APÊNDICE A – Questionário da entrevista orientada.

1. Você entendeu como implementamos a *feature* utilizando a abordagem proposta?
   Respostas possíveis:

   1 - não entendi nada

   2 - entendi muito pouco

   3 - entendi o básico

   4 - entendi bem

   5 - entendi perfeitamente

   1.a. Tem alguma dúvida?

2. Você achou a abordagem interessante?
   Respostas possíveis:

   1 - não achei nada interessante

   2 - achei pouco interessante

   3 - achei interessante

   4 - achei bem interessante

   5 - achei muito interessante

   2.a. Quer justificar o por quê?

3. Você acredita que poderia funcionar dentro do cenário recomendado pela abordagem? (recapitulando: projetos novos criados a partir do *forking* de um projeto original, e que necessitam atualizar constantemente com tal projeto original).
   Respostas possíveis:

   1 - não poderia funcionar de forma alguma

   2 - pouco provável que funcione

   3 - poderia funcionar

4 - grandes chances de funcionar

5 - muito provável que funcione

3.a. Quer justificar o por quê?

4. Você acredita que poderia funcionar dentro do cenário de um projeto antigo / em andamento (tal como o SUAP no IFSULDEMINAS), com certas adaptações?
   Respostas possíveis:

   1 - não funcionará, mesmo com adaptações

   2 - pouco provável que funcione, mesmo com adaptações

   3 - poderia funcionar, com adaptações

   4 - grandes chances de funcionar, com adaptações

   5 - muito provável que funcione, com adaptações

   4.a. - [Se sua resposta foi 3 à 5] Quais seriam as adaptações que acredita serem necessárias?
   4.b. - [Se sua resposta foi 1 ou 2] O que acredita que seja mais impeditivo?

5. Como você vê o fato de desenvolver de forma desacoplada (desenvolver um código customizado + arquivos de customização separados do código principal do sistema)?
   Respostas possíveis:

   1 - totalmente problemática

   2 - não vejo problemas, mas acho que existiria soluções melhores

   3 - pode ser desconfortável no início, mas acostuma-se

   4 - consigo ver vantagens no desenvolvimento desacoplado

   5 - vejo como a melhor solução

   5.a. - [resposta entre 3-5] Quer justificar o por quê?
   5.b. - [resposta entre 1-2] Teria uma outra ideia de como manter as atualizações do *branch* principal?

REFERÊNCIAS

4LINUX. O que é DevOps. 2020. Disponível em: <https://4linux.com.br/o-que-e-devops/>. Acesso em: 14 dec. 2020.

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source Java projects. **Empirical Software Engineering**, v. 23, n. 4, p. 2051–2085, 2018.

ACCIOLY, P. et al. Analyzing conflict predictors in open-source Java projects. In: **15th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2018. p. 576–586.

AYEWAH, N. et al. Using static analysis to find bugs. **IEEE Software**, v. 25, n. 5, p. 22–29, 2008.

BLAW, F. F. The use of git as version control in the south african software engineering classroom. In: **13th IST-Africa Week Conference (IST-Africa)**. [S.l.: s.n.], 2018. p. 1–8.

BORGES, H.; HORA, A.; VALENTE, M. T. Predicting the popularity of GitHub repositories. In: **12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)**. [S.l.: s.n.], 2016. p. 9.

BORGES, H.; HORA, A.; VALENTE, M. T. Understanding the factors that impact the popularity of GitHub repositories. In: **32rd IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2016. p. 334–344.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: **9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2015. p. 1–10.

CHACON, S.; STRAUB, B. **Pro git**. [S.l.]: Apress, 2014.

COSENTINO, V.; IZQUIERDO, J. L.; CABOT, J. A systematic mapping study of software development with GitHub. **IEEE Access**, v. 5, n. 5, p. 7173–7192, 2017.

COSTA, C. et al. Tipmerge: recommending experts for integrating changes across branches. In: **24th International Symposium on Foundations of Software Engineering (ESE/FSE)**. [S.l.: s.n.], 2016. p. 523–534.

EMANUELSSON, P.; NILSSON, U. A comparative study of industrial static analysis tools. **Electronic notes in theoretical computer science**, v. 217, n. 217, p. 5–21, 2008.

FOWLER, M. **Domain-specific languages**. [S.l.]: Pearson Education, 2010.

FOWLER, M. **Domain-Specific Language Guide**. 2019. Disponível em: <https://martinfowler.com/dsl.html>. Acesso em: 12 dec. 2020.

GUIMARÃES, M. L.; SILVA, A. R. Improving early detection of software merge conflicts. In: **34th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2012. p. 342–352.

HATTORI, L.; LANZA, M. Syde: A tool for collaborative software development. In: **32nd ACM/IEEE International Conference on Software Engineering-Volume (ICSE)**. [S.l.: s.n.], 2010. p. 235–238.

JIANG, J. et al. Why and how developers fork what from whom in GitHub. **Empirical Software Engineering**, v. 22, n. 1, p. 547–578, 2017.

JUNIOR, L. F. D. **O que é melhor: Feature Teams ou Component Teams?** 2018. Disponível em: <https://imasters.com.br/devsecops/o-que-e-melhor-feature-teams-ou-component-teams>. Acesso em: 15 dec. 2020.

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: **35th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2013. p. 732–741.

LEßENICH, O. et al. Indicators for merge conflicts in the wild: survey and empirical study. **Automated Software Engineering**, v. 25, n. 2, p. 279–313, 2018.

LI, Y. et al. Fhistorian: Locating features in version histories. In: **21st International Systems and Software Product Line Conference (SPLC)**. [S.l.: s.n.], 2017. p. 49–58.

MAHMOOD, W. et al. Causes of merge conflicts: a case study of elasticsearch. In: **14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)**. [S.l.: s.n.], 2020. p. 1–9.

MAHMOUDI, M.; NADI, S. The Android update problem: An empirical study. In: **15th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2018. p. 220–230.

MARTINS, L. A. **Análise da modularidade de características aspectuais de tecnologias para implementar linhas de produtos de software**. Dissertação (Mestrado) — Universidade Federal de Lavras, Programa de Pós-graduação em Ciência da Computação, Lavras / MG, 2019.

MCKEE, S. et al. Software practitioner perspectives on merge conflicts and resolutions. In: **33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2017. p. 467–478.

MENS, T. A state-of-the-art survey on software merging. **IEEE Transactions on Software Engineering**, v. 28, n. 5, p. 449–462, 2002.

NELSON, N. et al. The life-cycle of merge conflicts: processes, barriers, and strategies. **Empirical Software Engineering**, v. 24, n. 1, p. 1–44, 2019.

NGUYEN, H. L.; IGNAT, C.-L. An analysis of merge conflicts and resolutions in Git-based open source projects. **Computer Supported Cooperative Work**, v. 27, n. 3, p. 741–765, 2018.

NISHIMURA, Y.; MARUYAMA, K. Supporting merge conflict resolution by using fine-grained code change history. In: **23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. p. 661–664.

PORFIRIO, A.; PEREIRA, R.; MASCHIO, E. Atualização do modelo do aprendiz de programação de computadores com o uso de parser ast. In: **6th Congresso Brasileiro de Informática na Educação (CBIE)**. [S.l.: s.n.], 2017. p. 1121.

RUBIN, J. et al. Managing forked product variants. In: **16th International Software Product Line Conference (SPLC)**. [S.l.: s.n.], 2012. p. 156–160.

SANTOS, L. **Node.js Por Baixo Dos Panos 4 - Vamos Falar do V8**. 2019. Disponível em: <https://dev.to/khaosdoctor/node-js-por-baixo-dos-panos-4-vamos-falar-do-v8-4pai>. Acesso em: 12 dec. 2020.

SPINELLIS, D. Git. **IEEE Software**, v. 29, n. 3, p. 100–101, 2012.

STĂNCIULESCU, Ş.; SCHULZE, S.; WĄSOWSKI, A. Forked and integrated variants in an open-source firmware project. In: **31st IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2015. p. 151–160.

SUNG, C. et al. Towards understanding and fixing upstream merge induced conflicts in divergent forks: an industrial case study. In: **42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE)**. [S.l.: s.n.], 2020. p. 172–181.

TOMASSETTI, F. **The complete guide to (external) Domain Specific Languages**. 2017. Disponível em: <https://tomassetti.me/domain-specific-languages/>. Acesso em: 12 dec. 2020.

XU, X. et al. Enforcing access control in distributed version control systems. In: **20th IEEE International Conference on Multimedia and Expo (ICME)**. [S.l.: s.n.], 2019. p. 772–777.

ZHOU, S. et al. Identifying features in forks. In: **40th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2018. p. 105–116.