
Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Ciência da Computação

# Análise comparativa sobre estratégias de integração de renomeação ou deleção de métodos na ferramenta S3M

João Victor de Sá Ferraz Coutinho

Trabalho de Graduação

Recife
16 de maio de 2022

Universidade Federal de Pernambuco
Centro de Informática

João Victor de Sá Ferraz Coutinho

# Análise comparativa sobre estratégias de integração de renomeação ou deleção de métodos na ferramenta S3M

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

Orientador: *Paulo Henrique Monteiro Borba*
Co-orientador: *Guilherme José de Carvalho Cavalcanti*

Recife
16 de maio de 2022

# Agradecimentos

Inicialmente, gostaria de agradecer aos meus pais e ao restante de minha família, por estarem sempre comigo e me apoiarem em todos os momentos. Inclusive, o incentivo por ingressar na área de Computação veio por grande parte deles, e sou grato por isso.

Claro, agradeço também aos amigos que formei durante minha passagem pelo curso. Alguns se tornaram queridos demais e terão meu carinho e parceria por muito tempo.

Sou intensamente grato aos professores do Centro de Informática, que embora sofram com o caos que atinge a classe no país, fazem um trabalho excepcional de educar e serem guias aos ingênuos estudantes que passam pela universidade. Tendo aspirações educacionais eu mesmo, busco inspiração neles. Em especial, gostaria de enaltecer os professores Anjolina Grisi, Ruy Guerra e Paulo Borba, por todo o apoio em minha jornada na universidade e por serem mestres na condução de suas próprias áreas de atuação, as quais me encantaram grandiosamente.

Finalmente, num gesto questionável de potencial narcisismo, gostaria de agradecer a mim mesmo, por permitir amadurecer bastante em todos esses anos e suportar a caminhada – nada fácil, mas maravilhosa – pela estrada da vida.

*A felicidade é o fim que a natureza humana visa. E, a felicidade é uma atividade, pois não está acessível àqueles que passam sua vida adormecidos. Ela não é uma disposição. À felicidade nada falta, ela é completamente autossuficiente. É uma atividade que não visa nada a não ser a si mesma. O homem feliz, basta a si mesmo.*

—ARISTÓTELES

# Resumo

No âmbito de desenvolvimento de software, é comum que cada desenvolvedor altere independentemente uma parte do código e posteriormente integre suas mudanças utilizando uma estratégia de integração. A mais conhecida é a não-estruturada, que faz uma análise puramente textual nas alterações e é a mesma implementada pelo Git. Uma alternativa é a integração semiestruturada, que faz uma análise entre árvores sintáticas parciais dos códigos originais e é implementada pela ferramenta S3M para projetos Java. Um dos desafios da integração semiestruturada é a renomeação e deleção de métodos, visto que seus nós correspondentes existem na árvore do arquivo base mas deixam de existir nas árvores dos arquivos modificados e é trabalho da ferramenta encontrar tal correspondência utilizando heurísticas. Assim, o objetivo deste trabalho é analisar comparativamente diferentes estratégias (implementadas por componentes ou "handlers"do S3M) para lidar com renomeações ou deleções de métodos ou construtores no contexto da ferramenta. É possível notar que os *handlers* que implementamos e avaliamos neste trabalho não diferem tanto e que o *handler Keep Both Methods* ganha em falsos positivos – a ferramenta indica que houve conflito quando de fato não houve –, mas perde em negativos, ao mesmo passo em que os outros *handlers* perdem em falsos positivos, mas ganham em falsos negativos.

**Palavras-chave:** Integração de Código, Integração Semiestruturada, S3M, Desenvolvimento Colaborativo, Engenharia de Software

# Abstract

In the context of software development, it's usual that each developer independently change a piece of code and merge it later using a merge strategy. The most known strategy is the non-structured merge, which does a purely textual analysis of the changes and it's the same implemented by Git by default. An alternative is the semistructured merge, which does an analysis of the partial syntax trees of the original codes and it's implemented for Java projects by the S3M tool. A challenge of semistructured merge is the renaming and deletion of methods, because the corresponding nodes exist in the base file's tree but do not in the modified files' ones, and it's responsibility of the tool to find such correspondence using heuristics. As such, the present work aims to compare different strategies (implemented by S3M's components or "handlers") for merging renamed or deleted methods or constructors in the context of S3M. We observed that the handlers we implement and evaluate in this work don't differ much and that the *Keep Both Methods* handler performs better on false positives (the tool reports a conflict but there's no actual conflict) but does worse on negatives, as well as the others perform better on false positives but do worse on false negatives.

**Keywords:** Code Merge, Semistructured Merge, S3M, Collaborative Development, Software Engineering

# Sumário

# Lista de Figuras

# Lista de Tabelas

CAPÍTULO 1

# Introdução

É comum no ambiente de desenvolvimento colaborativo de *software* que desenvolvedores alterem independentemente regiões do código e posteriormente integrem suas mudanças utilizando uma estratégia de integração. Possivelmente, duas ou mais alterações são conflitantes, no sentido de que, se integradas juntas, podem impactar a corretude do sistema. Por exemplo, isso acontece quando dois desenvolvedores alteram a mesma linha de código. A detecção, entendimento e correção desses tais chamados conflitos de integração podem introduzir não apenas um longo atraso e uma consequente diminuição da produtividade do time, como também, se resolvidos de forma errada, uma perda da qualidade do projeto [7, 8].

Para combater tal problema, diversas estratégias de integração foram propostas. A mais adotada é a integração textual, que faz análises de similaridade de texto entre as alterações [12] e é a que o Git (ferramenta de controle de versão), por exemplo, implementa. Por se importar apenas com o texto dos códigos, essa estratégia é altamente rápida e simples, mas deve pagar o custo de reportar mais falsos positivos – conflitos que não deveriam ter ocorrido, mas a ferramenta os reportou – do que uma abordagem que tenha alguma consideração pela estrutura do código [5, 9]. Por exemplo, se dois desenvolvedores adicionam métodos diferentes em uma mesma região de código de uma classe, a integração textual reportará um conflito. Contudo, a situação poderia ser trivialmente resolvida mantendo os dois métodos em regiões distintas da classe, tornando desnecessária a intervenção manual pelos desenvolvedores. Esse exemplo será retomado no Capítulo 2.

A abordagem semiestruturada, por outro lado, processa árvores sintáticas parciais para cada artefato envolvido na integração e integra os nós com o mesmo identificador (assinatura de métodos ou nome de atributos, por exemplo) usando a abordagem textual para integrar os corpos das declarações. Tal estratégia diminui os falsos positivos do algoritmo textual, mas eleva os falsos negativos [10] – conflitos que deveriam ter ocorrido, mas a ferramenta deixou de reportá-los – devido a seus próprios problemas. Por exemplo, se dois desenvolvedores alteram um mesmo bloco estático, a ferramenta manterá ambos os blocos variantes, já que não possuem identificador para uní-los, e possivelmente impactará a corretude do sistema. Esse exemplo será também retomado no Capítulo 2.

A ferramenta S3M, que implementa o algoritmo de integração semiestruturada para códigos escritos em Java, trata esses casos por meio de componentes denominados *handlers*. Assim, este trabalho objetiva implementar quatro *handlers* diferentes para o caso de renomeação ou deleção de métodos ou construtores, implementar um método automático de avaliação para eles e então avaliá-los no quesito de quantas vezes eles diferem e de como eles afetam a acurácia da ferramenta, medida em termos de falsos positivos e falsos negativos reportados.

## 1.1 Estrutura do Trabalho

Este trabalho está segmentado em 6 capítulos, onde

- no Capítulo 2 está contida a fundamentação teórica deste trabalho, abordando detalhes sobre a integração textual e a semiestruturada e as especificações dos quatro *handlers* alvos do estudo;
- no Capítulo 3 está contida a metodologia utilizada para a análise comparativa dos *handlers*;
- no Capítulo 4 estão contidos os resultados obtidos com base na análise;
- no Capítulo 5 estão contidas menções a trabalhos relacionados;
- e no Capítulo 6 estão contidas a conclusão com uma análise final e perspectivas de trabalhos futuros.

CAPÍTULO 2

# Fundamentação Teórica

Este capítulo tem como objetivo apresentar conceitos importantes para a construção desta pesquisa e entendimento dos resultados obtidos. Assim, abordará conceitos iniciais de integração textual e semiestruturada e dos quatro *handlers* alvos do estudo.

## 2.1 Three-Way Merge

Tipicamente, uma integração (ou *merge* ou *three-way merge*) consiste de três artefatos: um base e duas contribuições a este. Esses artefatos podem assumir diversas representações, como linhas em arquivos ou nós em uma árvore (incluindo até mesmo arquivos em um diretório). Eles são comparados e integrados pelas estratégias, que definem o que foi modificado da base e, dentre as modificações, quais são conflitantes e que requerem intervenção manual pelos usuários. Para fins de clareza, daqui em diante, às contribuições serão dadas os nomes *LEFT* e *RIGHT*.

No cenário trivial em que apenas um desenvolvedor fez alterações – ou seja, apenas uma contribuição difere da base –, o resultado da integração é exatamente a contribuição variante. Tal integração é chamada de *fast-forward merge* [13] e está exemplificada na Figura 2.1.


**Figura 2.1** Funcionamento do *Fast-Forward Merge* em linhas de um arquivo, onde apenas *RIGHT* difere da base.

Em um cenário não-trivial – ou seja, onde ambas as contribuições diferem da base –, o resultado da integração tipicamente depende do local das alterações. Por exemplo, alterações paralelas em arquivos diferentes ou em regiões diferentes de um mesmo arquivo tipicamente podem coexistir e são integradas diretamente. Em contrapartida, alterações em uma mesma região de um arquivo são problemáticas pois elas podem não coexistir na região da mesma forma em que foram feitas e devem ser repensadas. A Figura 2.2 ilustra tal cenário.

Cada estratégia de integração define uma maneira própria de lidar com tais situações não-triviais. Como foco do estudo, serão abordadas as integrações não-estruturada e semiestruturada.


**Figura 2.2** Problema típico de uma integração não-trivial envolvendo linhas de um arquivo: no *merge*, deveria estar B1 (alteração de *LEFT*) ou B2 (alteração de *RIGHT*)?

## 2.2 Integração Não-estruturada

Também chamada de "integração textual", é utilizada pelo Git, ferramenta de controle de versão mais popular atualmente [2], tem bases no algoritmo de Myers [14] e é implementada formalmente pela ferramenta *diff3* [12]. Essa estratégia analisa e integra blocos de texto e, de modo geral, reporta conflitos quando os desenvolvedores fazem alterações diferentes na mesma linha ou em linhas consecutivas de uma mesma região de código. A Figura 2.3 ilustra o resultado de aplicar esse algoritmo no cenário da Figura 2.2.

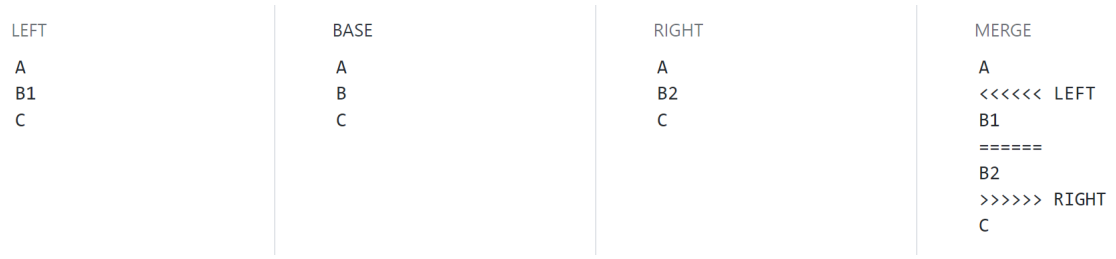


**Figura 2.3** A integração textual marca as alterações simultâneas como conflitantes e requer intervenção manual dos desenvolvedores para resolver o conflito.

Por lidar apenas com análises textuais, essa estratégia obtém os benefícios de ser performática e agnóstica a linguagens de programação. Contudo, tal simplicidade envolve um débito em precisão, visto que a ferramenta tende a reportar muitos falsos positivos [5, 9], os quais por sua vez deterioram a produtividade do time de desenvolvimento, que deve resolvê-los manualmente. Um cenário típico (envolvendo classes Java) desse defeito pode ser visto na Figura 2.4.

## 2.3 Integração Semiestruturada

Dado o problema com falsos positivos da integração textual, houve motivação para criar um algoritmo de *merge* que buscasse consideração pela estrutura do arquivo sendo integrado [4]. Tal estratégia foi chamada de integração estruturada, e envolve gerar árvores sintáticas dos artefatos envolvidos e então integrá-las. Apesar de reduzir falsos positivos, esse algoritmo requer um custo muito alto de performance e desenvolvimento [4], já que uma implementação do al-

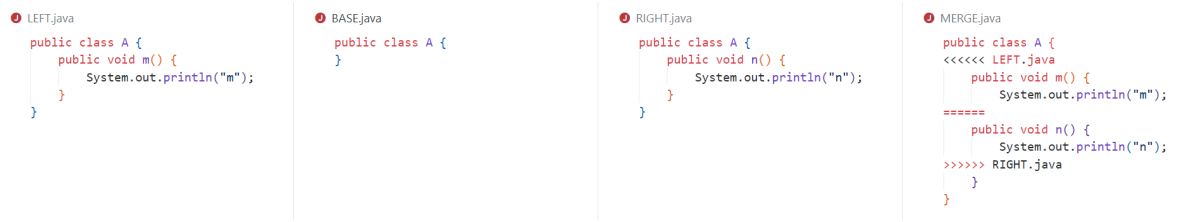


**Figura 2.4** Cenário onde tanto *LEFT* quanto *RIGHT* adicionam um método novo na classe A. Apesar de que semanticamente as alterações não são conflitantes (já que os métodos podem coexistir na mesma classe), a integração textual reporta um conflito visto que elas foram feitas nas mesmas linhas.

goritmo só serveria potencialmente para uma versão específica de uma linguagem específica. Além disso, a integração estruturada tende a reportar muitos falsos negativos em relação a integração textual [11], tendo maiores chances de comprometer a corretude do projeto e causar danos em longo prazo.

Dessa forma, sendo concebida como um intermediário entre a integração textual e a estruturada, a integração semiestruturada visa criar árvores sintáticas parciais dos códigos envolvidos na integração, encontrar correspondências nas três árvores pelos identificadores de seus nós (assinatura de métodos ou nome de atributos, por exemplo) e integrá-los. A Figura 2.5 ilustra o mesmo cenário da Figura 2.4, porém diferindo pelo resultado, que foi determinado pela abordagem semiestruturada.

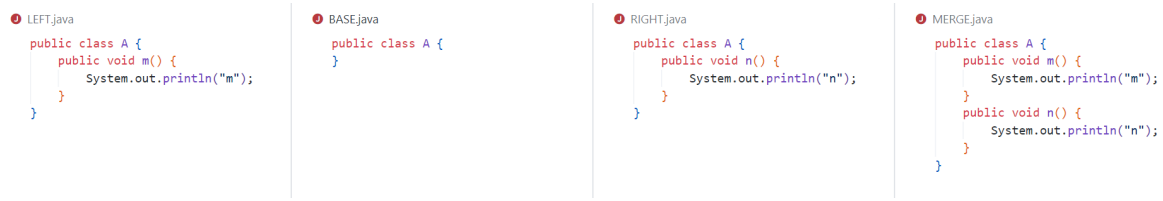


**Figura 2.5** Diferentemente da integração textual, a semiestruturada mantém ambos os métodos adicionados sem conflito.

Ainda, na integração semiestruturada, o corpo das declarações internas de uma classe (métodos, atributos etc) são armazenados como texto nos nós correspondentes. Quando os nós são integrados, o *merge* textual é chamado em seus corpos. A Figura 2.6 exemplifica esse cenário.

Finalmente, apesar de que a integração semiestruturada efetivamente possui uma ocorrência menor de falsos positivos quando comparada com a integração textual, ela eleva a ocorrência de falsos negativos e introduz outros falsos positivos (que tipicamente são mais difíceis de resolver) [10]. Um exemplo de um falso negativo extra da integração semiestruturada está descrito na Figura 2.7, também citado no Capítulo 1.

Para então combater esse problema, Cavalcanti et al. [10] sugerem a criação de *handlers*, que são componentes baseados em heurísticas que incrementam a execução da integração semiestruturada, possivelmente desfazendo conflitos criados por esta ou criando outros.

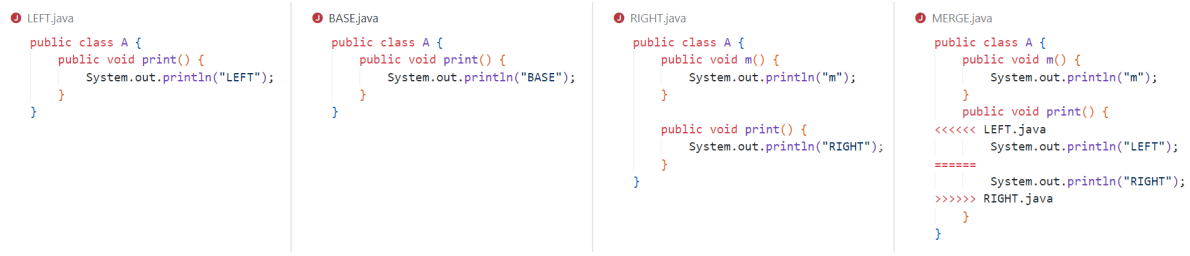


**Figura 2.6** Na integração semiestruturada, o método *print()* está presente nas três declarações e seu corpo é integrado utilizando a abordagem textual, que reporta conflito já que houve alterações na mesma linha. Alternativamente, caso uma contribuição tivesse deletado o método, assumiria-se que o corpo do nó faltante seria a *string* vazia.


**Figura 2.7** Cenário onde *LEFT* e *RIGHT* alteraram simultaneamente o mesmo bloco estático. Uma vez que a declaração não tem identificadores claros, a integração mantém ambos. Porém, a ausência de conflitos faz com que a integração altere silenciosamente a corretude do programa original. Vale notar que a integração textual reportará um conflito, já que as alterações ocorreram na mesma linha.

Esse trabalho possui um foco na atuação de *handlers* que lidam com renomeação ou deleção de métodos ou construtores. A seguir, será mostrado a motivação que Cavalcanti et al. [10] encontram para a criação desse tipo de *handler* e a implementação de quatro *handlers* desse tipo que esse trabalho tem como objetivo.

## 2.4 *Handlers* de renomeação e deleção de métodos ou construtores

Na Figura 2.8, está ilustrado um cenário de conflito de renomeação. Nesse cenário, integrar ambas as contribuições – ou seja, o novo corpo com a nova assinatura – seria válido porque as mudanças não afetam as expectativas dos desenvolvedores: o método teria o comportamento esperado por um deles e seria chamado como esperado pelo outro. Inclusive, é exatamente assim que a integração textual se comportaria nesse cenário, visto que as mudanças não ocorrem em linhas consecutivas e, portanto, esse conflito é um falso positivo extra da integração semiestruturada. Em contrapartida, ainda no exemplo anterior, suponha que *LEFT* houvesse também adicionado uma nova chamada a *div(int, int)*. A consequência disso é que o conflito se tornaria um verdadeiro positivo, uma vez que silenciosamente integrar as contribuições acarretaria em uma chamada adicional a um método que já não mais existe.

Assim, para lidar com cenários de integração envolvendo renomeações ou deleções de mé-

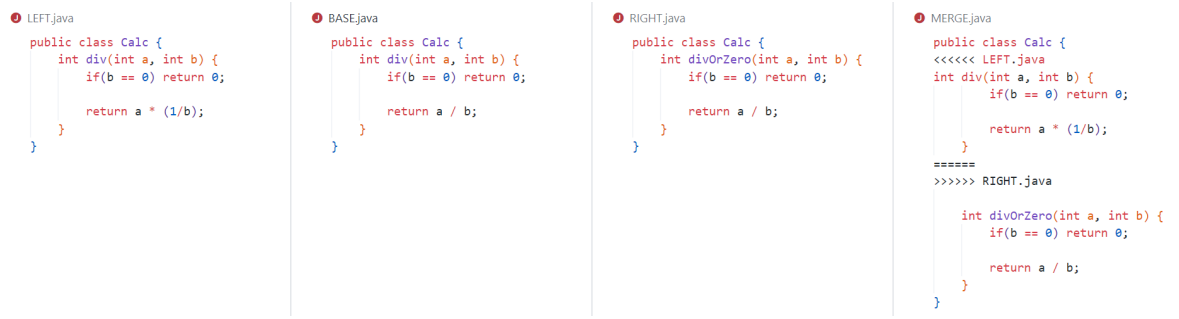


**Figura 2.8** Cenário onde *LEFT* altera o corpo do método *div(int, int)* e *RIGHT* renomeia o mesmo método para *divOrZero(int, int)*. Uma vez que não há uma correspondência de *div(int, int)* em *RIGHT*, a integração semiestruturada assume que *RIGHT* deletou o método. Assim, seu corpo é uma *string* vazia para os propósitos da integração textual e consequentemente tem um conflito com o corpo modificado de *div(int, int)* em *LEFT*.

todos ou construtores, esse trabalho propõe a implementação e análise comparativa de quatro *handlers*, definidos abaixo. Notório salientar, de antemão, que dois dos quatro *handlers* citados estendem a lógica de correspondência feita pela integração semiestruturada. Ou seja, definem mais possibilidades – além de somente "mesmo identificador" – para considerar dois nós em árvores diferentes como correspondentes. Isso é intuitivo pelo fato de que, após uma renomeação ou deleção, os nós não mais terão o mesmo identificador. As regras de correspondência foram pensadas da seguinte forma:

Dado um par de nós $(n_1, n_2)$, onde $n_1$ e $n_2$ representam métodos ou construtores, $n_1$ e $n_2$ são correspondentes se qualquer uma das condições a seguir for verdadeira:

- $n_1$ e $n_2$ possuem a mesma assinatura;
- $n_1$ e $n_2$ possuem o mesmo corpo;
- $n_1$ e $n_2$ possuem corpos 70% similares (distância de Levenshtein [6]) e tem ou o mesmo nome ou requerem os mesmos parâmetros;
- o corpo de $n_1$ está contido em $n_2$ ou o corpo de $n_2$ está contido em $n_1$.

No cenário da figura 2.8, por exemplo, o método *div(int, int)* na base corresponderia ao método *div(int, int)* em *LEFT*, uma vez que os métodos têm a mesma assinatura (1ª condição), e corresponderia ao método *divOrZero(int, int)* em *RIGHT*, uma vez que os métodos possuem o mesmo corpo (2ª condição). Se um nó não encontrou um par que atendeu a alguma das condições acima, ele é considerado deletado para os propósitos do *handler*.

### 2.4.1 Merge Methods

Se a integração semiestruturada utilizasse as heurísticas de correspondência como definidas acima, então ela poderia estender a integração textual consequente nos corpos das declarações

para métodos que não precisam necessariamente ter a mesma assinatura. Assim, essa estratégia foi externalizada neste *handler*, sendo efetivamente, portanto, uma "extensão natural" da integração semiestruturada. Sua atuação está ilustrada na Figura 2.9.

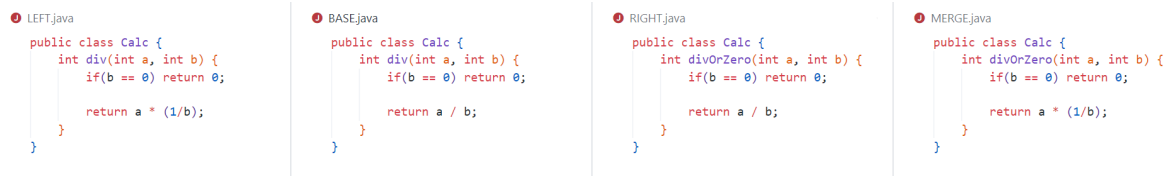


**Figura 2.9** Atuação do *Merge Methods*. O *handler* identifica que *div(int, int)* na base corresponde a *div(int, int)* em *LEFT* por terem a mesma assinatura e a *divOrZero(int, int)* em *RIGHT* por terem o mesmo corpo. Então, integra os três métodos utilizando a integração textual nos corpos das declarações, o que causa nenhum conflito.

### 2.4.2 Check References and Merge Methods

No exemplo anterior, se *LEFT* houvesse adicionado uma referência nova a *div(int, int)*, significaria que a ausência de conflitos seria um falso negativo, como já discutido anteriormente, e ilustrado na Figura 2.10. Como o *Merge Methods* não lida com isso, ele é suscetível a esse problema. Assim, um novo *handler* foi proposto que fizesse a checagem de novas chamadas para o método que não foi renomeado e reportasse conflito se essa situação se confirmasse. Entretanto, na implementação, essa filosofia não persistiu. Ao invés disso, o *handler* reporta conflitos devido a novas chamadas somente quando os desenvolvedores alteram o método para a mesma assinatura e para corpos diferentes. Essa situação está ilustrada na Figura 2.11. Em quaisquer outros casos, o *Check References and Merge Methods* realiza a integração textual nos corpos das declarações, assim como o *Merge Methods*.


**Figura 2.10** Falso negativo do *Merge Methods*. Como *div(int, int)* não existe mais, a integração sem conflitos introduziu um erro de compilação no código. Assim, um conflito deveria ocorrer envolvendo os métodos *div(int, int)* e *divOrZero(int, int)*.

Vale notar que esse *handler* age como um intermediário entre a integração semiestruturada e a estruturada: apesar de executar dentro do contexto da primeira, utiliza da granularidade de estrutura proposta pela segunda. Tal estratégia compactua com a tese, levantada por Cavalcanti et al. [11], de que a acurácia máxima de integração deve ser obtida por um intermediário entre as duas abordagens de integração.

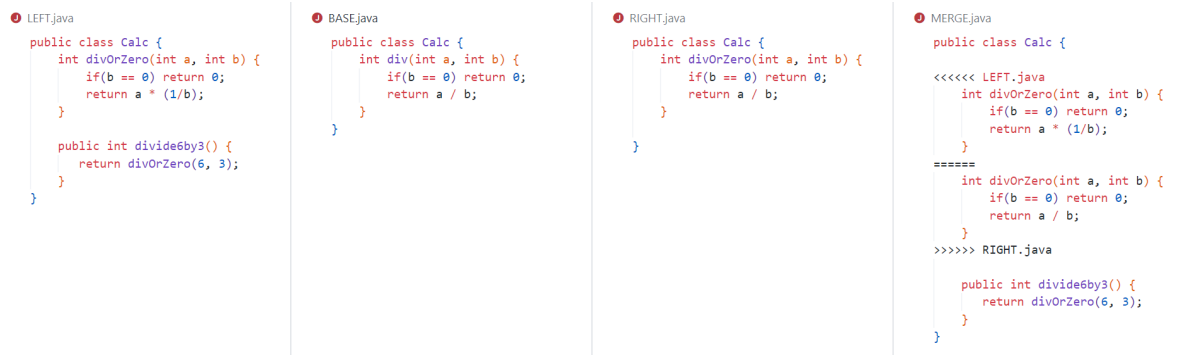


**Figura 2.11** Atuação do *Check References and Merge Methods*. O *handler* detecta que o método *div(int, int)* foi renomeado para a mesma assinatura pelos dois desenvolvedores e, uma vez que os corpos dos métodos são textualmente diferentes e há uma nova chamada a *divOrZero(int, int)*, ele reporta um conflito.

### 2.4.3 Keep Both Methods

No exemplo da Figura 2.8, observamos um falso positivo de renomeação típico da integração semiestruturada, uma vez que as mudanças dos desenvolvedores podem coexistir. Cenários onde um desenvolvedor deleta ou renomeia um método e o outro altera seu corpo sempre acarretam em conflito pela integração semiestruturada. Assim, o *Keep Both Methods* tem como objetivo desfazer tais conflitos, mantendo, no resultado do *merge*, o método alterado – no caso de uma deleção – ou ambos os métodos – no caso de uma renomeação, tendo ciência de que essa estratégia não é conservadora para o contexto de falsos negativos. Essa estratégia está ilustrada na Figura 2.12.

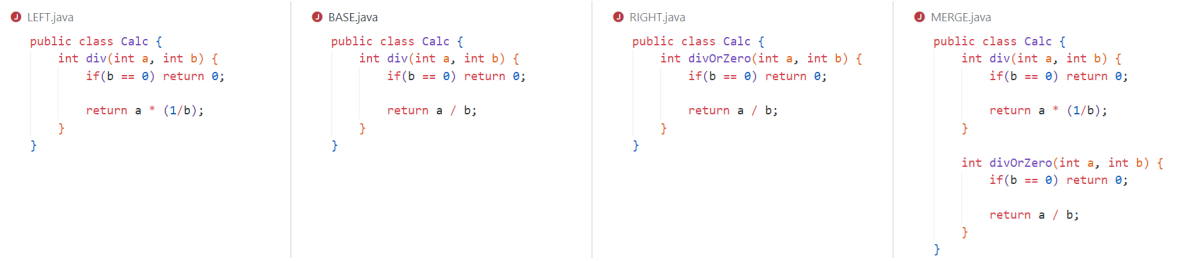


**Figura 2.12** Mesmo cenário da Figura 2.8, porém sob atuação do *Keep Both Methods*, que mantém ambos os métodos.

Vale notar, porém, que a atuação do *handler* em cenários onde ambos os desenvolvedores renomeiam o método para a mesma assinatura pode acarretar em um falso negativo óbvio, já que dois métodos com a mesma assinatura em uma classe é uma construção inválida para a linguagem Java. Assim, nesses casos, o *handler* se omite e não desfaz nada feito pela integração semiestruturada. Esta, por sua vez, faz a integração textual no corpo dos métodos assumindo que o corpo do método da base é a *string* vazia, e consequentemente reporta conflito se qualquer

linha no corpo dos métodos alterados for distinta nas duas contribuições. Essa situação está ilustrada na Figura 2.13.

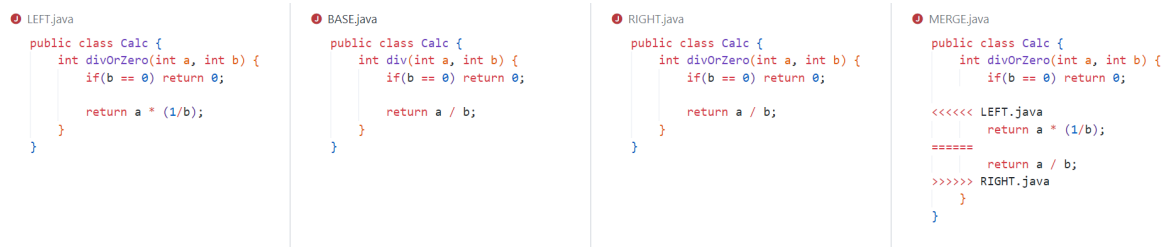


**Figura 2.13** Cenário onde *LEFT* e *RIGHT* renomeiam o método *div(int, int)* para o mesmo nome *divOrZero*. Como as assinaturas são as mesmas, a integração semiestruturada (e efetivamente o *Keep Both Methods*) faz a integração entre eles assumindo que o corpo do nó faltante da base é a *string* vazia, reportando um conflito.

### 2.4.4 Check Textual and Keep Both Methods

Ainda no cenário da Figura 2.8, vimos um caso de falso positivo extra da integração semiestruturada. Porém, se a integração textual também houvesse reportado um conflito envolvendo o método *div(int, int)*, o falso positivo deixaria de ser extra para ser comum às duas estratégias. Essa análise corrobora a ideia, também citada por Cavalcanti et al. [10], de que a integração semiestruturada, sempre que possível, não deveria ser pior que a integração textual.

Assim, o *Check Textual and Keep Both Methods* visa nivelar o *merge* semiestruturado e reportar conflito de renomeação apenas na situação em que a integração textual também reportou um conflito envolvendo a assinatura do método original. Ou seja, se não for o caso, o *handler* desfaz o conflito que a integração semiestruturada teria feito nesse cenário, agindo de forma similar ao *Keep Both Methods*. Dessa forma, o *handler* aposta que, se o conflito for na verdade um verdadeiro positivo, o consequente falso negativo seria comum às duas integrações. A atuação do *handler* pode ser observada na Figura 2.14.

Em qualquer outro cenário, o *handler* se comporta exatamente como o *Keep Both Methods*.

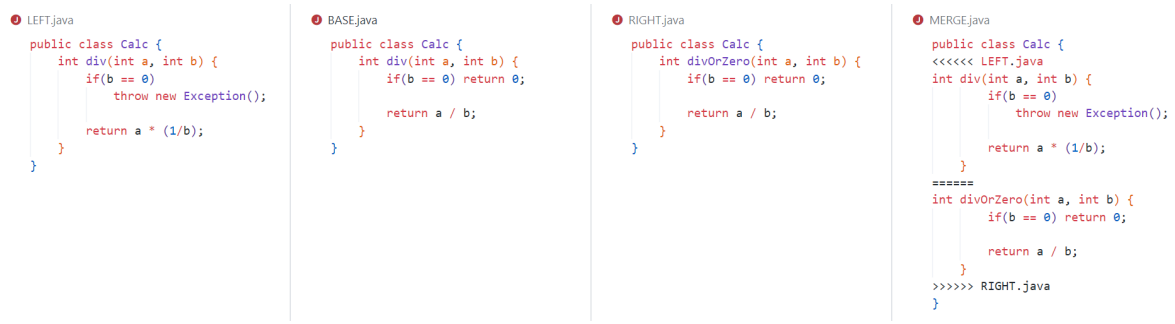


**Figura 2.14** Atuação do *Check Textual and Keep Both Methods*. Nesse cenário, *RIGHT* renomeou o método *div(int, int)* para *divOrZero(int, int)* e *LEFT* alterou o corpo de *div(int, int)* imediatamente após a assinatura deste. Como houve alteração em linhas consecutivas, a integração textual reportaria conflito. Assim, o *handler* mantém o conflito, mas adapta-o para englobar os dois métodos.

CAPÍTULO 3

# Metodologia

Este capítulo tem como objetivo apresentar as perguntas de pesquisa que este trabalho visa responder, bem como a metodologia usada para tal fim.

Inicialmente, queremos saber se há muita diferença entre utilizar um ou outro *handler* para podermos avaliar o impacto de sua atuação na integração semiestruturada. Por exemplo, se, em um dado cenário, todos os *handlers* deram o mesmo resultado, então escolher entre um deles para aquele cenário não fez diferença. Ainda, se isso é tendência, significa que a atuação deles é a mesma em maioria dos cenários, dando evidência que pode ser prudente simplificá-los.

Além disso, nos cenários em que eles diferem – ou seja, quando escolher um deles é relevante –, queremos avaliar quais deles contribuem mais positivamente para a acurácia da ferramenta. Tal avaliação tem como objetivo coletar alguma evidência de que um *handler* (ou sua filosofia) possa ser mais capacitado do que outro para resolver o problema da integração envolvendo renomeação ou deleção de métodos ou construtores.

Dessa forma, podemos condensar essas considerações iniciais em duas perguntas de pesquisa:

**Q1:** Os *handlers* diferem em quantos % dos cenários de integração?

**Q2:** Quando diferem, quais *handlers* se comportam melhor no contexto de falsos positivos e falsos negativos?

Assim, este trabalho propõe testá-los isoladamente em diversos cenários de integração – *commits* de *merge* – e comparar seus resultados.

Os cenários foram extraídos de um grupo de 6 repositórios de código aberto com maior número de estrelas do *GitHub* – o nome deles e seus *links* estão disponíveis no Apêndice A. Para extraí-los, estendemos e instanciamos a ferramenta *Mining Framework*, que pode ser instanciada para realizar diversos tipos de estudo similares. Em especial, essa ferramenta permite a injeção de componentes que fazem filtragem e coleta de dados de cenários de *merge* em projetos Git. A próxima seção detalha sobre como os cenários foram filtrados, enquanto que a seguinte detalha como foram usados para avaliar os *handlers*.

## 3.1 Filtragem dos cenários de integração

Para obter maior controle sobre a execução dos *handlers*, os cenários foram filtrados por duas condições:

1. não devem ser *fast-forward*, pois a integração é resolvida trivialmente (sem atuação dos *handlers*);

2. e devem conter no mínimo uma situação envolvendo renomeação ou deleção de métodos ou construtores.

Para validar a segunda condição, utilizamos a ferramenta *JavaParser* para fazer o parseamento dos arquivos envolvidos no cenário de integração. Se a base possuir um nó representando um método ou um construtor que não está presente em alguma das contribuições, então podemos assumir com segurança de que o método foi ou renomeado ou deletado por algum desenvolvedor.

## 3.2 Avaliação dos *handlers*

A ferramenta S3M permite execução isolada de cada *handler* de renomeação ou deleção de métodos ou construtores via argumentos de linha de comando. Assim, para cada cenário coletado e devidamente filtrado, a ferramenta S3M foi executada quatro vezes: uma para cada *handler* habilitado por vez.

Para responder à primeira pergunta de pesquisa, o resultado de cada *handler* foi comparado textualmente entre si. Assim, os *handlers* diferem em um determinado cenário se o resultado de pelo menos um deles difere textualmente do de um outro. Em alguns casos, os *handlers Keep Both Methods* e *Check Textual and Keep Both Methods* não respeitaram a identação original do arquivo ao serem executados. Assim, para evitar imprecisões devido a isso, a comparação foi feita desconsiderando espaços em branco.

Para responder à segunda, os resultados foram analisados em cada cenário seguindo os critérios a seguir:

- Se a atuação de algum *handler* no cenário gerou um resultado sem conflitos e igual ao do *commit* de *merge* do projeto, então o cenário é contabilizado como contendo um falso positivo para os *handlers* que geraram um resultado com conflitos. A comparação com o *commit* de *merge* é feita pensando que ele é, a princípio, o que o desenvolvedor quer que seja o resultado da integração.

- Se a atuação de algum *handler* no cenário gerou um resultado com conflitos, então o cenário é contabilizado como contendo um falso negativo para os *handlers* que geraram um resultado sem conflitos e que introduzira um erro de compilação no projeto. Isso é possível saber substituindo o resultado do *handler* no *commit* de *merge* do projeto e avaliando a compilação do mesmo.

CAPÍTULO 4

# Resultados

Neste capítulo serão apresentados os resultados encontrados com as análises comparativas feitas baseadas no processo de coleta e execução apresentado no capítulo anterior. Este capítulo está dividido em duas seções, uma para cada pergunta de pesquisa.

## 4.1 Diferença textual entre os *handlers*

Respondendo à primeira pergunta de pesquisa – *Os handlers diferem em quantos % dos cenários de integração?* –, temos um resultado bem direto: de 678 cenários elegíveis de integração e extraídos de seis repositórios, os *handlers* alvo diferiram textualmente em seus resultados em apenas 31 cenários. Isso equivale a uma porcentagem de aproximadamente 4,7%. A distribuição dos cenários por repositório pode ser vista na Tabela 4.1.

Uma possível hipótese para tal resultado baixo é que, em situações onde um desenvolvedor renomeou ou deletou um método, o outro fez alterações desconexas com essa ou em outras regiões do código. Analisando apenas o método, é possível fazer a metáfora de que essa situação é essencialmente um *fast-forward* para a integração do método e, assim, todo *handler*, assim como a integração semiestruturada pura, gerará o mesmo resultado.

| Projeto | Cenários coletados e filtrados | Cenários com diferença textual entre *handlers* |
|---|---|---|
| Dubbo | 143 (21,0%) | 19 (2,8%) |
| ElasticSearch | 128 (18,8%) | 10 (1,5%) |
| JsonIterator | 29 (4,2%) | 0 (0%) |
| Kubernetes | 29 (4,2%) | 0 (0%) |
| PubNub | 96 (14,1%) | 2 (0,3%) |
| TheAlgorithms | 253 (37,3%) | 0 (0%) |

**Tabela 4.1** Distribuição de cenários para cada repositório

## 4.2 Diferença de erros entre os *handlers*

Já para a segunda pergunta de pesquisa – *Quando diferem, quais handlers se comportam melhor no contexto de falsos positivos e falsos negativos?* –, como definidas no capítulo anterior, análises foram feitas para cada um dos cenários em que os *handlers* diferiram, buscando possíveis falsos positivos ou negativos nos cenários. Os resultados podem ser vistos na Tabela 4.2.

| | Cenários contendo FPs | Cenários contendo FNs |
|---|---|---|
| Merge Methods | 18 (58,0%) | 0 (0%) |
| Check References and Merge Methods | 18 (58,0%) | 0 (0%) |
| Keep Both Methods | 0 (0%) | 7 (22,5%) |
| Check Textual and Keep Both Methods | 14 (45,1%) | 0 (0%) |

**Tabela 4.2** Número de erros para cada *handler* avaliado

Como esperado, *handlers* que possuem uma política de evitar conflitos, como o *Keep Both Methods*, arriscam reportar menos falsos positivos ao mesmo passo em que arriscam reportar mais falsos negativos. Em contrapartida, *handlers* que possuem regras para definir conflitos, como o *Check References and Merge Methods*, tendem a reportar mais falsos positivos, enquanto reduzem os falsos negativos.

Um dado adicional é que, dos 7 cenários contendo falsos negativos reportados pelo *Keep Both Methods*, 3 envolveram atributos ou métodos faltantes – um método que referenciava um atributo ou um método deletado por outro desenvolvedor permaneceu após a integração –, o que pode indicar que a estratégia desse *handler* deve possuir um ponto fraco nas referências e levanta a hipótese de viabilidade de um *handler* que faça essas checagens.

Ainda, outro ponto a observar é que os *handlers* que trabalham integrando diretamente os métodos (*Merge Methods* e *Check References and Merge Methods*) não são melhores que a estratégia de avaliar a integração textual previamente (*Check Textual and Keep Both Methods)*, nem em falsos positivos, nem em falsos negativos. Esta, inclusive, possui uma leve vantagem em falsos positivos, com 4 cenários de diferença entre as estratégias.

Finalmente, é notável que as regras adicionais de checagem de referências do *Check References and Merge Methods* não fazem diferença na precisão do mesmo, quando comparado com o *Merge Methods*. É possível supor, porém, que se o *handler* tivesse sido implementado de acordo com sua filosofia original, esse resultado mudasse. Isso parte da hipótese de que a atuação principal da implementação atual – requer renomeação dupla para a mesma assinatura, alteração de corpo e chamadas novas – é rara demais.

Portanto, é possível concluir que o *handler Check Textual and Keep Both Methods* parece ser o mais adequado em situações onde falsos negativos são indesejáveis. Nas situações onde falsos negativos forem um problema menor, então o *handler Keep Both Methods* se torna o mais adequado.

## 4.3 Ameaças a validade

Mesmo com a metodologia definida, é necessário levantar possíveis ameaças à validade dos resultados encontrados neste trabalho. Uma possível ameaça, por exemplo, é a quantidade relativamente pequena de cenários coletados. Note, ainda, que não incluímos, na filtragem de cenários, uma condição adicional de checar se ambos os desenvolvedores alteraram um mesmo arquivo. Sem essa validação, nas ocasiões onde isso não ocorre, é esperado que os *handlers* não dêem resultados diferentes. Afinal, a renomeação ou deleção obrigatória pela 2ª condição pode ter ocorrido por apenas um desenvolvedor, e o outro pode não ter modificado seu corpo.

Outra plausível ameaça é a possível falta de robustez nos critérios de definição de falsos negativos, uma vez que não capturamos falsos negativos que podem acarretar em conflitos de *teste* – apenas a nível de compilação.

CAPÍTULO 5

# Trabalhos Relacionados

Inicialmente, Apel et al. [5] propuseram a criação de uma ferramenta de integração semiestruturada, apelidada de *FSTMerge*, afim de combater os problemas vistos ocasionados pela integração textual. Sete anos depois, Cavalcanti et al. [10] publicam um trabalho que avalia as diferenças de precisão e performance das duas integrações, além de propor melhorias nessas duas frentes com o uso de *handlers* ou módulos.

Em 2021, Clementino et al. [3] propuseram o estudo de uma ferramenta de merge não-estruturado que simulasse a integração estruturada, utilizando não somente linhas na busca por mudanças, como também separadores sintáticos específicos da linguagem. Apesar de não ter obtido resultados muito expressivos, ainda sim notou evidência de redução de falsos positivos, o que abre espaço para novos estudos.

Ainda, em um estudo similar a este, Oliveira et al. [1] propuseram a implementação e avaliação de *handlers* para o caso de blocos estáticos. Como vimos no Capítulo 2, esse é um problema claro da integração semiestruturada. Em seu estudo, Oliveira et al. mostram que transicionar de fazer correspondência entre blocos apenas com regra de nível de similaridade textual não era robusto o suficiente, e a melhoria proposta de criar um novo *handler* que checasse também o nível de inserção em um bloco efetivamente permitiu acréscimo da precisão da ferramenta.

CAPÍTULO 6

# Conclusão

Nesse trabalho, visamos implementar quatro componentes (chamados *handlers*) para lidar com os casos de renomeação e deleção de métodos ou atributos da ferramenta *S3M*, que implementa a integração semiestruturada para projetos Java. Além disso, foram avaliados utilizando de uma instanciação da ferramenta *Mining Framework*, também implementada para esse estudo, sob os critérios de quantas vezes eles diferem e de como eles afetam a acurácia da ferramenta, medida em termos de falsos positivos e falsos negativos reportados.

Com os resultados obtidos, é possível perceber que, como esperado, os *handlers* diferem pouco. Uma hipótese provável é que maioria dos cenários não envolve atuação completa dos mesmos: apenas um desenvolvedor altera um método.

Além disso, podemos concluir que *handlers* que possuem políticas de remover conflitos (como o *Keep Both Methods*), por tenderem a reportar menos conflitos, também reportam menos falsos positivos. Porém, por esse mesmo motivo, reportam mais falsos negativos. Além disso, os *handlers Check Textual and Keep Both Methods*, *Merge Methods* e *Check References and Merge Methods* não diferem tanto em seus erros, mas com uma leve vantagem em falsos positivos para o primeiro, devido a sua política conservadora de evitá-los.

Finalmente, é possível notar que *handlers* com políticas de integrar os métodos, como *Merge Methods* e *Check References and Merge Methods*, reportam o maior número de falsos positivos, visto que devem lidar com a suscetibilidade de conflitos da integração textual. Checar referências parece não fazer diferença na precisão do último *handler*, mas dada a hipótese que sua atuação atual é mais limitada.

## 6.1 Trabalhos Futuros

Como trabalhos futuros, uma mesma análise pode ser feita com: (1) os mesmos *handlers*, mas refinando os critérios de falsos negativos; (2) outros *handlers*, obedecendo outras estratégias; (3) refinamento da implementação do *handler Check References and Merge Methods*, como mencionado no Capítulo 2, para compactuar melhor com sua filosofia inicial; (4) uma maior amostra ou melhor definição de filtro de cenários, a fim de capturar com mais precisão a execução dos *handlers*.

Além disso, um futuro trabalho envolve fazer uma análise que avalia a contribuição das heurísticas adicionais que os *handlers* utilizam, para medir seu impacto em detecção de renomeações ou deleções de métodos ou construtores.

APÊNDICE A

# Repositórios utilizados e *dataset*

Aqui temos a lista de repositórios cujos *commits* de integração foram utilizados para a análise.

**Dubbo** https://github.com/apache/dubbo

**Elasticsearch** https://github.com/elastic/elasticsearch

**Kubernetes** https://github.com/kubernetes-client/java

**The Algorithms** https://github.com/TheAlgorithms/Java

**Json Iterator** https://github.com/json-iterator/java

**PubNub** https://github.com/pubnub/java

O dataset com todos os *commits* coletados e filtrados pode ser encontrado neste *link*: shorturl.at/tDRY3

APÊNDICE B

# Instanciação da ferramenta *Mining Framework*

*Link* que redireciona para a ferramenta *Mining Framework*: https://github.com/spgroup/miningframework

A ferramenta foi instanciada com a classe *services.dataCollectors.S3MMergesCollector*, que implementa a interface *DataCollector*, para coleta de dados dos cenários; e com a classe *services.commitFilters.S3MCommitFilter*, que implementa a interface *CommitFilter*, para filtro de cenários.

APÊNDICE C

# *Links* para algoritmos formais e implementação dos *handlers*

*Link* para uma formalização proposta dos algoritmos da integração semiestruturada e *handlers* de renomeação ou deleção de métodos ou atributos, também proposto por este trabalho:
https://github.com/guilhermejccavalcanti/jFSTMerge/blob/master/documentation/Algorithms/Algorithms.pdf

Suas implementações podem ser encontradas nestes *links*:

**Merge Methods** shorturl.at/vyCTY

**Check References and Merge Methods** shorturl.at/deFZ9

**Keep Both Methods** shorturl.at/dnzMX

**Check Textual and Keep Both Methods** shorturl.at/dmtR8

# Referências Bibliográficas


[1] Aperfeiçoando técnicas de merge entre blocos de inicialização estáticos para uma ferramenta de merge semiestruturado. https://www.cin.ufpe.br/~tg/2019-1/TG_EC/TG_azbo.pdf. Accessado em 01/05/2022.

[2] Comparação de popularidade dos sistemas de controle de versão. https://www.openhub.net/repositories/compare. Accessado em 09/05/2022.

[3] Merge textual baseado em separadores de elementos sintáticos de linguagens de programação. https://www.cin.ufpe.br/~tg/2020-1/TG_CC/tg_joc.pdf. Accessado em 01/05/2022.

[4] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 120–129, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, and William Cook. Semistructured merge in revision control systems. 01 2010.

[6] Bonnie Berger, Michael S. Waterman, and Yun William Yu. Levenshtein distance, sequence comparison and biological database search. *IEEE Transactions on Information Theory*, 67(6):3287–3294, 2021.

[7] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 168–178, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. Assessing semistructured merge in version control systems: A replicated experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2015.

[10] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[11] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. The impact of structure on software merging: Semistructured versus structured merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1002–1013, 2019.

[12] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07, page 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] José William Menezes, Bruno Trindade, João Felipe Pimentel, Tayane Moura, Alexandre Plastino, Leonardo Murta, and Catarina Costa. What causes merge conflicts? In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 203–212, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Eugene W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 2005.