UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Presidente Prudente

Leandro Ungari Cayres

Aprendizado de Padrões Sintáticos utilizando Histórico de Versão para Sugestão Automatizada de Modificações de Código

Presidente Prudente
Janeiro – 2021

Leandro Ungari Cayres

Aprendizado de Padrões Sintáticos utilizando Histórico de Versão para Sugestão Automatizada de Modificações de Código

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação - Área de Concentração Sistemas de Informação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: CAPES

Orientador: Prof. Dr. Rogério Eduardo Garcia

Presidente Prudente
Janeiro – 2021

C385a

Cayres, Leandro Ungari

Aprendizado de Padrões Sintáticos utilizando Histórico de Versão para Sugestão Automatizada de Modificações de Código / Leandro Ungari Cayres. -- Presidente Prudente, 2021

100 f. : il., tabs., fotos

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências e Tecnologia, Presidente Prudente

Orientador: Rogério Eduardo Garcia

1. Ciência da Computação. 2. Engenharia de Software. 3. Software Manutenção. 4. Aprendizado de Modificações de Código-Fonte. 5. Manutenabilidade de Software. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências e Tecnologia, Presidente Prudente. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de São José do Rio Preto

IBILCE

ATESTADO DE APROVAÇÃO - DEFESA

Atestamos que **LEANDRO UNGARI CAYRES**, RA nº: CCO180271, RG nº 50.600.108-8, expedido pela SSP/SP, defendeu, no dia 26/01/2021, a dissertação intitulada ***Aprendizado de Padrões Sintáticos utilizando Histórico de Versão para Sugestão Automatizada de Modificações de Código***, junto ao Programa de Pós Graduação em Ciência da Computação, Curso de Mestrado Acadêmico, tendo sido 'APROVADO'.

Atestamos ainda que a obtenção do título dependerá de homologação pelo Órgão Colegiado competente.

São José do Rio Preto, 26 de janeiro de 2021

Silvia Emiko Kazama
Supervisor Técnico de Seção
Seção Técnica de Pós-Graduação

Leandro Ungari Cayres

Aprendizado de Padrões Sintáticos utilizando Histórico de Versão para Sugestão Automatizada de Modificações de Código

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação - Área de Concentração Sistemas de Informação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: CAPES

BANCA EXAMINADORA

Prof. Dr. Rogério Eduardo Garcia
UNESP–Presidente Prudente
Orientador

Prof. Dr. Reginaldo Ré
Professor Doutor
Universidade Tecnológica Federal do Paraná

Prof. Dr. Danilo Medeiros Eler
Professor Assistente Doutor
UNESP–Presidente Prudente

Presidente Prudente
Janeiro – 2021

Dedico este trabalho
à minha família

Agradecimentos

Agradeço, primeiramente, a Jesus Cristo, pelo dom da vida, e que permitiu que pudesse superar os desafios a cada dia.

Agradeço aos meus pais, Miguel e Rosileni, que me ensinaram e me formaram sob valores pelo quais luto e defendo a cada dia, por todos os esforços em momentos de necessidade e dificuldade, assim como os momentos de alegria partilhados. Obrigado por tudo.

Agradeço à minha irmã Suzi, e ao meu cunhado Rafael, por todo o apoio em diversos momentos e pelas alegrias, lanches e aprendizados.

Agradeço ao meu orientador, Rogério, pelo auxílio, acompanhamento dedicado a mim e todo conhecimento partilhado ao longo de todos esses anos, como professor e orientador. Obrigado, Professor.

Agradeço aos meus colegas de laboratório, pelos momentos partilhados no LaPESA, principalmente ao Bruno, Wilson, Ingrid e Rafael; que aprendi muitas coisas, e pude partilhar o pouco que sei ajudando os outros.

Agradeço aos meus colegas de graduação que fizeram parte dessa trajetória. Também agradeço aos funcionários do Departamento de Matemática e Computação, dando suporte durante meu período como docente substituto, assim como suporte técnico.

Agradeço ao Prof. Danilo Medeiros Eler, por todo auxílio no crescimento da minha formação tanto na graduação quanto na pós-graduação.

Agradeço a todos que, direta ou indiretamente, me apoiaram e me auxiliaram no decorrer deste trabalho. Obrigado a todos vocês.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Uma característica de sistemas de software é o seu constante estado de evolução, em que diferentes tipos de modificações são aplicadas, as quais podem aumentar a complexidade do código-fonte, e consequentemente, reduzir a manutenabilidade do software. A refatoração de software é uma prática amplamente reconhecida para a redução de complexidade e reestruturação do código-fonte. Contudo, as abordagens existentes de identificação de oportunidades de refatorações apresentam diversas limitações, tanto em relação ao número de técnicas de refatoração disponíveis quanto à complexidade de modificações contempladas, o que pode não suprir as reais necessidades por parte dos desenvolvedores. Nesse sentido, este trabalho propõe um processo de aprendizado de modificações de código-fonte baseado em exemplos, com o intuito de identificar padrões sintáticos de modificações que propiciem a melhoria contínua do código-fonte, para que esses possam ser reaproveitadas e replicados no mesmo e em outros repositórios. O processo de aprendizado de modificações foi conduzido de modo individualizado em cada um dos repositórios. A partir dos resultados extraídos de um conjunto de repositórios, foi possível identificar um conjunto de instâncias de modificações simples, principalmente relacionadas com movimentação e renomeação de entidades de código-fonte, e também um conjunto de refatorações de software simples e compostas.

Palavras-chave: Histórico de versão. Padrões de modificação de código-fonte. Manutenabilidade de software.

Abstract

The continuous evolution state is a common to any software system, which developers conduct different types of code changes and may increase code complexity, resulting in the reduction of software maintainability. Software refactoring is a widely known practice to the reduction of complexity by the reestructuring of source code. However, the existing approaches to identify opportunities of refactorings present plenty of limitations, such as the number of available refactorings techniques and the complexity of their transformations, which may not supply the indeed needs of developers. From that, this study proposes a process of learning source code changes based on examples, to identify code patterns which further the continuous improvement of source code, and they allow the reuse and replication of them through the same and other repositories. We conducted this process trough each repository separately. From the obtained results, we identify a set of simple code changes, mainly related to move and rename modifications of code entities, and also, some of simple and composed software refactorings.

Keywords: History version. Patterns of source code changes. Software maintainability.

Lista de Figuras

Lista de Tabelas

Sumário

CAPÍTULO

1

Introdução

1.1 Contextualização

Uma característica comum em sistemas de software é a evolução. Modificações de código-fonte são realizadas com diferentes propósitos, como correção de defeitos, adição de novas funcionalidades e refatorações (Mens e Tourwé, 2004); e impactam diretamente a qualidade interna do software. Tais modificações, se conduzidas de modo inadequado, podem aumentar a complexidade do software, degradando sua estrutura interna, aumentando a propensão a defeitos e, como consequência, passam a exigir esforço e tempo para manutenção.

Com intuito de contribuir no processo de evolução de sistemas de software, estudos têm sido conduzidos para apoiar diferentes etapas do processo de desenvolvimento. Nesse contexto, as refatorações de software têm papel importante para a recuperação da organização interna do código-fonte. As refatorações de software consistem em modificações que reestruturam o código-fonte, permitindo a melhoria da organização interna, sem alterar o comportamento do software (Opdyke, 1992). As refatorações têm sido adotadas para a redução da complexidade de código-fonte (Fowler, 2018).

No contexto de refatoração, algumas iniciativas têm sido desenvolvidas como o Catálogo de Refatorações (Fowler, 2019) e abordagens automatizadas para a identificação de pontos suscetíveis a refatorações (Dig et al., 2006, Prete et al., 2010, Silva e Valente, 2017, Tsantalis et al., 2018). Tais abordagens identificam modificações de código-fonte que caracterizam técnicas de refatoração disponíveis na literatura (Fowler, 2019).

Contudo, as abordagens automatizadas, propostas na literatura para a detecção de refatorações, são limitadas em relação à complexidade das modificações, assim como, ao número de técnicas de refatoração contempladas. Al Dallal (2015) aponta que a maioria das abordagens de identificação automatizada não contemplam cerca de 72% das técnicas de refatoração presentes no Catálogo de Refatoração. Além disso, na maioria dos casos, as abordagens auto-

matizadas identificam modificações simples dentre as técnicas já catalogadas, utilizando regras predefinidas e heurísticas (Meng et al., 2011, Raychev et al., 2013). De modo geral, essas abordagens tendem a não suprir as reais necessidades dos desenvolvedores, especialmente quando modificações e/ou refatorações mais complexas são necessárias.

Nesse sentido, há trabalhos (Meng et al., 2013, Rolim et al., 2017, 2018a) que adotam uma estratégia na identificação de modificações, o uso de trechos de código-fonte como exemplos, para o aprendizado de modificações de código-fonte, e posterior reutilização (Rolim et al., 2017). Tais abordagens, baseadas na utilização de exemplos de modificações, são empregadas em contextos como: correção de exercícios (Head et al., 2017) e apresentação de dicas em cursos de programação (Phothilimthana e Sridhara, 2017), correção de defeitos em repositórios (Osman et al., 2014, Hanam et al., 2016, Zhong e Meng, 2018) e aplicação de modificações em geral (Martinez et al., 2013, Negara et al., 2014, Molderez et al., 2017).

Outra vertente, que também utiliza modificações préexistentes, é o reparo automático de programas (*APR – Automated Program Repair*). O foco do *APR* consiste em encontrar, de modo automatizado, uma solução (correção) para problemas de software (defeitos) sem intervenção humana. Basicamente, envolve quatro passos: 1) identificação da falha (*Fault Identification*); 2) localização do defeito causador da falha (*Fault localization*); 3) geração da correção (*Patch Generation*); 4) avaliação (*Patch Evaluation*) para determinar se a correção gerada realmente corrige o defeito. O terceiro passo, provavelmente o mais difícil e desafiador, consiste em gerar a correção para as linhas de código que contêm o defeito.

Há diversas técnicas diferentes, mas específicas para defeitos conhecidos (Monperrus, 2019, Wang et al., 2019). *APR* tem como foco defeitos – *bugs*, sem entrar na discussão de termos como *defect*, *fault*, *error*, *failure*, *mistake*, dentre outros (Monperrus, 2018) – o que limita a aplicação das técnicas disponíveis. Como exemplo de limitação do foco, Durieux et al. (2017) propõem a geração de correções especificamente para exceções geradas por ponteiros nulos (*null pointer exceptions*).

1.2 Motivação e Justificativa

O histórico de versão de um repositório de software armazena as operações realizadas sob o código-fonte entre duas versões (*commits*) subsequentes, de modo que a versão anterior pode ser obtida a partir da versão atual, e assim sucessivamente, compondo o histórico completo do repositório (Chacon e Straub, 2014). A partir dessas operações, pode-se obter as modificações realizadas em cada versão, as quais podem ser identificadas e reaproveitadas em futuras alterações no repositório. Contudo, cada modificação pode impactar degradando ou melhorando a arquitetura interna do código-fonte em repositórios.

Na literatura, as modificações adaptativas (ex. melhorias de código-fonte) estão restritas somente ao uso de refatorações de software, cuja automação, nas maioria das abordagens, se baseia no uso de regras predefinidas ou heurísticas, que resulta em um número restrito de técnicas de refatorações e em modificações sintaticamente simples (Al Dallal, 2015). Por outro lado, as abordagens que utilizam o aprendizado de modificações baseado em exemplos focam,

principalmente, em modificações preventivas ou corretivas, as quais são relacionadas a defeitos no código-fonte. Desse modo, por meio de um processo de aprendizado de modificações de código-fonte, pretende-se ampliar o conjunto de modificações identificadas que permitam a melhoria do código-fonte, não somente refatorações de software conhecidas, mas também modificações adicionais que sejam comumente aplicadas em um ou mais repositórios.

A utilização de modificações passadas pode permitir avanços na identificação de modificações que possibilitem a melhoria do código-fonte, de modo que essas possam ser reutilizadas ao longo do processo de desenvolvimento. De modo complementar, modificações que impactem de modo negativo ao código-fonte, degradando-o, possam ser evitadas.

1.3 Objetivos

Este trabalho de mestrado tem como objetivo geral aprimorar o processo de evolução da qualidade do código-fonte, por meio do aprendizado automatizado de modificações que possibilitem a melhoria da qualidade interna do software, com base em exemplos extraídos do histórico de repositórios de software. A proposta consiste em, por meio da análise do histórico de modificações em diferentes repositórios de código-fonte, identificar padrões sintáticos de modificações caracterizadas como melhorias de código-fonte. Uma melhoria de código-fonte é uma modificação que propicia um aumento qualitativo das propriedades do código-fonte. Cada modificação é classificada com base no impacto sob a manutenabilidade do código-fonte, cujo cálculo é feito utilizando um conjunto de métricas de qualidade de código-fonte.

Como objetivo específico, busca-se identificar um conjunto de padrões de modificações, que ao serem aplicadas que aumentam a qualidade interna do código-fonte. De modo complementar, também pretende-se identificar um conjunto de anti-padrões (*anti-patterns*), ou seja, modificações que ao serem aplicadas tendam a degradar o código-fonte. Além de identificar os padrões sintáticos das codificações, é importante ter esse mapeamento (impacto positivo ou negativo) sobre a qualidade interna do código-fonte, para subsidiar o processo de manutenção.

1.4 Organização

Este trabalho engloba diferentes conceitos dentro da Engenharia de Software como modificações de código-fonte e o processo de diferenciação, assim como qualidade de código-fonte para avaliação da manutenabilidade de software. Desse modo, para contextualizar a proposta de trabalho e os respectivos resultados, o presente documento está organizado como descrito a seguir:

- No Capítulo 2 são apresentados os conceitos relativos ao foco deste trabalho quanto a análise de repositórios de software, especialmente quanto a modificações adaptativas, como refatorações e algumas das principais abordagens na literatura. Na sequência é explorado o processo de diferenciação de código-fonte e como esse pode ser aplicado com o intuito de identificar e aprender tais modificações.

- No Capítulo 3 é apresentado o conceito de manutenabilidade de software, juntamente com

as principais métricas e algoritmos utilizados na avaliação do impacto de modificações em repositórios.

- No Capítulo 4 é descrito o processo de extração e análise dos padrões de modificações obtidos de repositórios, desde a identificação das modificações em cada *commit*, extração dos mapeamentos e os respectivos padrões sintáticos e, por fim, o processo de avaliação do impacto de cada modificação.
- No Capítulo 5 são apresentados os resultados e descobertas obtidas da análise dos repositórios.
- No Capítulo 6 são apresentadas as considerações finais e os trabalhos futuros.

CAPÍTULO

2

Modificações de Código-Fonte e sua Identificação

Uma modificação de código-fonte pode ser caracterizada como qualquer alteração na estrutura de um código-fonte. Essa modificação estrutural é definida na perspectiva sintática do código-fonte, e pode ou não impactar no comportamento do programa. Uma modificação de código-fonte é definida pela diferença entre duas Árvores Sintáticas Abstratas (*AST – Abstract Syntax Tree*) (SEBESTA, 2011), uma representando o código-fonte antes de alterações nele efetuadas (AST_i), e outra representando o código-fonte após a modificação (AST_o). As modificações aplicadas entre ambas *ASTs* são definidas por $AST_i \ \Delta \ AST_o$.

Modificações de código-fonte podem ser empregadas para diferentes propósitos. Por exemplo, para a correção de defeito, adição de novas funcionalidades, adaptação de código-fonte existente, melhoria da organização interna e redução de complexidade (Mens e Tourwé, 2004). Em relação às modificações adaptativas, é possível observar que são executadas de modo sistemático durante o processo de evolução do software – ou seja, são similares, mas não idênticas, em muitas localidades do código (Meng et al., 2013). Refatorações de software se enquadram como modificações sistemáticas, as quais permitem a melhoria da qualidade interna do código (Fowler, 2018).

Independentemente do tipo, o foco deste projeto é o aprendizado de modificações e a avaliação de seus impactos sob o código-fonte. Portanto, é mais relevante o estudo e, consequentemente, a apresentação das características das modificações e seus impactos, do que critérios para aplicação de modificações sugeridas na literatura, como é o caso das refatorações. O presente capítulo encontra-se organizado nas seguintes seções:

- Na Seção 2.1 é apresentada a definição de refatoração de software, juntamente com as principais iniciativas na literatura quanto às técnicas de refatoração, pois trata-se de um ponto de partida para análise de mudanças, e deve subsidiar o estudo de suas características. Abordagens automatizadas para detecção de pontos candidato à refatoração, es-

pecificamente aplicados a repositórios de software, também são apresentados – uma vez identificados, tais pontos são pontos de análise, assim como o impacto da aplicação das modificações neles aplicados;

- Na Seção 2.2 é apresentada a diferenciação de código-fonte: antes e depois de modificações, é possível identificá-las de modo automatizado. Uma vez identificadas, as modificações podem ser mapeadas e "aprendidas", de modo a auxiliar na melhoria do código-fonte;

- Na Seção 2.3 é apresentado o aprendizado de modificações efetuadas sistematicamente e as principais abordagens relacionadas.

2.1 Refatoração de Software

Diversas modificações são aplicadas ao longo de processo de desenvolvimento de um software, que podem tornar cada vez mais complexa a estrutura, em alguns casos modificando inclusive a arquitetura inicialmente projetada. Para evitar o aumento descontrolado da complexidade, faz-se necessária a aplicação de técnicas que permitam a sua redução e propiciem a melhoria da qualidade interna do software (Mens e Tourwé, 2004). O domínio de pesquisa que investiga esse problema é a reestruturação (*restructuring*) (Arnold, 1986).

Segundo Chikofsky e Cross (1990), a reestruturação consiste em transformar de uma representação de código-fonte em outra, com o mesmo nível de abstração, enquanto se preserva o comportamento externo do sistema, tanto em termos funcionais quanto semânticos. Uma reestruturação é geralmente uma modificação de aparência, na qual o código alterado para melhorar a sua estrutura em termos do senso tradicional de projeto. Em geral, essa modificação não é resultado de um novo requisito, mas pode levar a melhores observações do software, e sugerir novas modificações que melhorem aspectos do sistema. Para sistemas de software orientados a objetos, o termo mais específico consiste na refatoração.

Segundo Opdyke (1992), refatorar (*refactoring*) consiste no processo de modificar um sistema de software orientado a objeto, de modo que não seja alterado o seu comportamento externo, porém melhore sua estrutura interna. A ideia básica desse conceito consiste em reorganizar classes, atributos e métodos para facilitar adaptações e extensões do software (Mens e Tourwé, 2004).

Fowler (2019) apresenta um catálogo de técnicas de refatoração amplamente conhecidas por profissionais da área de Engenharia de Software. O emprego dessas técnicas permite melhores práticas de programação, de modo a reduzir a complexidade do software e, consequentemente, aumento na qualidade interna do código-fonte – objetivo deste projeto. Na Tabela 2.1 [1] é apresentada a lista completa de técnicas de refatoração contempladas no catálogo.

Conforme originalmente propostas, as técnicas de refatoração apresentadas na Tabela 2.1 devem ser aplicadas em pontos específicos de um código-fonte, para produzir o efeito pretendido na refatoração. Na literatura, trabalhos têm visado a identificação de oportunidades de refatorações, por meio da análise do histórico de versão em repositórios de software, tais como:

[1] As refatorações apresentadas se referem à primeira versão do catálogo.

Tabela 2.1: Técnicas de Refatoração contempladas no Catálogo de Refatoração.

Extract Method	*Inline Method*	*Inline Temp*
Replace Temp with Query	*Introduce Explaining Variable*	*Split Temporary Variable*
Remove Assignments to Parameters	*Replace Method with Method Object*	*Substitute Algorithm*
Move Method	*Move Field*	*Extract Class*
Inline Class	*Hide Delegate*	*Remove Middle Man*
Introduce Foreign Method	*Introduce Local Extension*	*Self Encapsulate Field*
Replace Data Value with Object	*Change Value to Reference*	*Change Reference to Value*
Replace Array with Object	*Duplicate Observed Data*	*Change Unidirectional Association to Bidirectional*
Change Bidirectional Association to Unidirectional	*Replace Magic Number with Symbolic Constant*	*Encapsulate Field*
Encapsulate Collection	*Replace Record with Data Class*	*Replace Type Code with Class*
Replace Type Code with Subclasses	*Replace Type Code with StateStrategy*	*Replace Subclass with Fields*
Decompose Conditional	*Consolidate Conditional Expression*	*Consolidate Duplicate Conditional Fragments*
Remove Control Flag	*Replace Nested Conditional with Guard Clauses*	*Replace Conditional with Polymorphism*
Introduce Null Object	*Introduce Assertion*	*Rename Method*
Add Parameter	*Remove Parameter*	*Separate Query from Modifier*
Parameterize Method	*Replace Parameter with Explicit Methods*	*Preserve Whole Object*
Replace Parameter with Method	*Introduce Parameter Object*	*Remove Setting Method*
Hide Method	*Replace Constructor with Factory Method*	*Encapsulate Downcast*
Replace Error Code with Exception	*Replace Exception with Test*	*Pull Up Field*
Pull Up Method	*Pull Up Constructor Body*	*Push Down Method*
Push Down Field	*Extract Subclass*	*Extract Superclass*
Extract Interface	*Collapse Hierarchy*	*Form Template Method*
Replace Inheritance with Delegation	*Replace Delegation with Inheritance*	

Fonte: Adaptada de Fowler (2019).

- a coleta de metadados da interação com o ambiente de desenvolvimento (Murphy-Hill et al., 2012);
- a análise de termos predefinidos nas mensagens de *commit* (Ratzinger et al., 2008);
- a geração e a execução automática de casos de teste (Soares et al., 2010), para identificar modificações que preservem o comportamento original do software.

Além disso, outros trabalhos buscam identificação de oportunidades de refatoração de modo mais preciso, por meio da análise direta do código-fonte, conhecida como *Análise Estática*. A seguir, são descritas algumas técnicas de identificação de oportunidades de refatoração.

2.1.1 Refactoring Crawler

Dig et al. (2006) propõem a técnica *Refactoring Crawler*, cuja estratégia consiste na combinação de análise sintática e de análise referenciada de grafos, de modo a permitir a detecção de pontos no código-fonte candidatos à refatoração, e o mapeamento das dependências e refinamentos dos pontos candidatos encontrados.

Inicialmente, a estrutura sintática do programa, expressa em uma *AST*, é analisada. Nesse contexto, é suficiente considerar que nós da *AST* representam as entidades de programa (pacotes, classes, métodos e atributos). Em seguida, aplica-se a técnica *Shingles Encoding* (Broder, 1997), com o intuito de encontrar pares de entidades similares que sejam potencialmente suscetíveis a refatorações. Nessa técnica, cada elemento sintático é convertido em valor numérico, de modo que conjuntos numéricos semelhantes representem trechos de código semelhantes.

Por fim, realiza-se a identificação da técnica de refatoração de cada modificação. Esse processo utiliza um conjunto de regras semânticas, em que são verificados relacionamentos definidos para cada técnica de refatoração analisada, de modo a calcular uma probabilidade da modificação analisada corresponder a uma dessas, em detrimento de um limiar predefinido. A técnica que obtiver a maior probabilidade, desde que maior que o limiar, é definida como a refatoração aplicada na modificação. Um ponto importante nessa abordagem consiste na possibilidade de mais uma técnica obter probabilidades relevantes na modificação. Por exemplo, as técnicas *move method* e *push-down method* são similares, contudo a segunda introduz a necessidade de um relacionamento de herança entre a classe de origem e destino. Nesses casos, sempre é selecionada a refatoração mais específica por meio do cálculo de probabilidade.

Essa técnica abrange sete tipos de refatoração para o processo de detecção: *Rename Package*, *Rename Class*, *Rename Method*, *Push Down Method*, *Pull Up Method*, *Move Method* e *Add Parameter*.

2.1.2 Ref-Finder

A técnica *Ref-Finder*, proposta por Prete et al. (2010), baseia-se em lógica de programação e é capaz de detectar um conjunto de 63 tipos de refatorações. Para tanto, cada uma das refatorações é definida por um conjunto de regras baseada em um modelo pré e pós-definição. A estratégia adotada pela técnica é composta pelos seguintes passos:

- Inicialmente, realiza-se uma varredura das *ASTs* de todos as classes existentes no repositório com o objetivo de extrair fatos sobre as entidades de código – por exemplo, dependências estruturais entre classes, métodos e atributos; e seus identificadores – de modo a representar as entidades de código em termos de fatos lógicos. Tais fatos são mantidos em uma base de dados.

- Em seguida, busca-se inferir instâncias concretas de refatoração presentes no código-fonte,

juntamente com os tipos específicos de cada uma das refatorações, por meio dos fatos lógicos obtidos na etapa anterior.

- Por fim, refatorações mais simples são combinadas de modo a representar possíveis refatorações complexas.

2.1.3 RMiner

A técnica *RMiner*, proposta por Tsantalis et al. (2018), tem como intuito de minerar refatorações em grandes repositórios de código-fonte. Essa técnica identifica um conjunto de 14 tipos de refatorações, apresentado na Tabela 2.2.

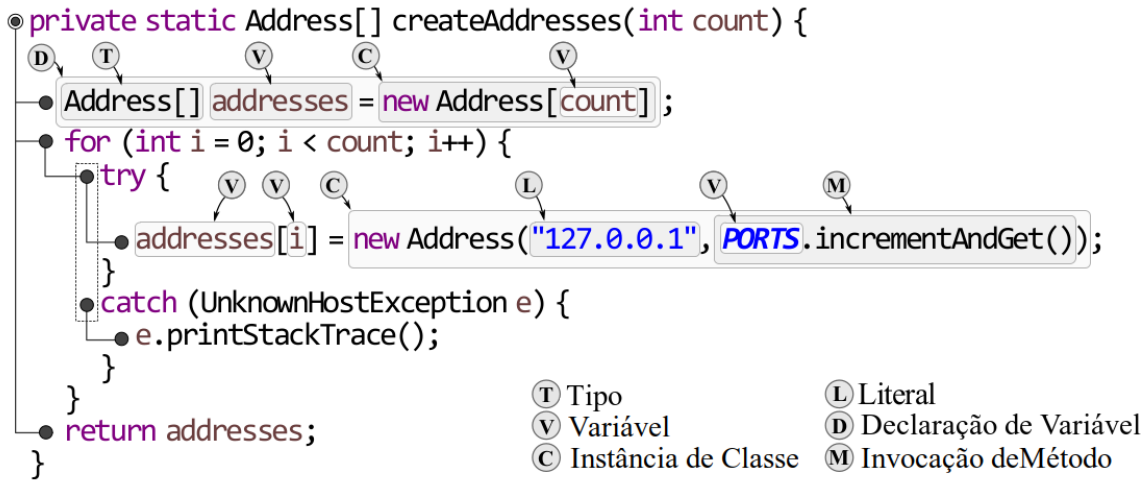
Tabela 2.2: Refatorações detectadas pela *RMiner*.

Categoria	**Refatorações**
Package	Rename Package
Class	Move Class, Rename Class, Extract Superclass, Extract Interface
Method	Extract Method, Move Method, Rename Method, Inline Method, Push Down Method, Pull Up Method
Field	Push Down Field, Pull Up Field, Move Field

Fonte: Adaptada de Tsantalis et al. (2018).

A estratégia da técnica se baseia no algoritmo *UMLDiff* (Xing e Stroulia, 2005), que visa diferenciar entidades de código orientados a objeto a partir da análise das classes, métodos e atributos de duas versões de código-fonte subsequentes, e que passaram por operações de adição, remoção e/ou movimentação de código.

Figura 2.1: Representação do corpo de um método com uma árvore sintática abstrata.



Fonte: Adaptada de Tsantalis et al. (2018).

A estratégia se inicia com uma análise *top-down* (a partir de classes, métodos até a variáveis e expressões) na árvore sintática abstrata, identificando e classificando cada um dos elementos sintáticos da linguagem, para identificar o conjunto de classes, métodos e atributos que foram adicionados, modificados e removidos entre duas versões subsequentes. Figura 2.1 é ilustrada

a classificação sintática de cada elemento do código-fonte em relação ao exemplo (trecho de código-fonte) representado. Por fim, cada uma das modificações é analisada em relação a um conjunto de regras estruturais, para verificar se a modificação em questão é compatível com o conjunto de regras de algum tipo de refatoração.

2.1.4 Ref-Diff

A técnica *Ref-Diff*, proposta por Silva e Valente (2017), utiliza um conjunto de heurísticas e cálculo de similaridade de código para identificar os tipos de refatorações aplicadas entre versões em repositórios. A quantidade de tipos de refatorações de software contempladas pela técnica é o mesmo da técnica *RMiner* (Seção 2.1.3). O funcionamento do técnica é dividido em duas etapas: *Análise do Código-Fonte* e *Análise de Relacionamento*.

Na *Análise de Código-Fonte*, o código-fonte é inspecionado com o propósito de identificar entidades de código (classes – *Type (T)*, métodos – *Method (M)* e atributos – *Field (F)*) e compor uma representação de alto nível que contenha todas as entidades, resumidamente $E = (T_i U M_i U F_i)$. Como a análise é conduzida entre versões subsequentes. Desse modo, para cada versão é extraído um modelo de alto nível antes das modificações E_a e outro após as modificações E_b.

Na *Análise de Relacionamento*, o objetivo consiste em identificar os relacionamentos entre as entidades de código E_a e E_b. O processo consiste em construir um grafo bipartido com dois conjuntos de vértices utilizando informações entre as entidades, resultando em um conjunto de relacionamentos R que interliga ambas. Por fim, as modificações são classificadas nos respectivos tipos de refatoração utilizando a mais abrangente combinação dentre os relacionamentos identificados antes e depois da modificação.

2.1.5 Limitações Observadas

Apesar dos avanços no processo de identificação de oportunidades de refatoração, as presentes técnicas possuem algumas limitações. De modo majoritário, tais técnicas focam em um conjunto restrito de técnicas de refatoração. Al Dallal (2015) identificou que 72,2% das técnicas de refatoração presentes no Catálogo de Fowler não são contempladas por essas técnicas. Esse comportamento resulta, em parte, da dificuldade da implementação de alguns tipos de refatorações, mas também da necessidade de comparação de resultados entre estudos, de modo que ambas as partes acabem se restringindo mutuamente a um conjunto pequeno de refatorações.

Tais limitações também estão presentes no processo de aplicação de refatorações. Com o intuito de identificar as motivações da condução de refatorações, Silva et al. (2016) obteve que as técnicas refatorações mais amplamente aplicadas são: *extract-method*, *move class*, *move attribute*, *rename package* e *move method*; o que demonstra um baixo nível de complexidade das transformações. Além disso, também foi obtido que a aplicação de refatorações em 55% dos casos são aplicadas de forma manual, em média, para as diferentes técnicas de refatoração, e que o percentual aumenta conforme a complexidade da alteração.

2.2 Diferenciação de Código-Fonte

Antes de tratar de Diferenciação de Código-Fonte, é necessário introduzir o conceito de árvore sintática abstrata. Uma árvore sintática abstrata T é formada por conjunto de nós, de modo que somente um desses seja a raíz $root(T)$. Cada nó $t \in T$, com exceção da raíz $root(T)$, possui um nó-pai $p \in T \cup \emptyset$, que é denotado por $parent(p)$. Também, cada nó $t \in T$ possui uma sequência de nós-filhos $(children(t))$, assim como um rótulo em um alfabeto $l \in \sum (label(t) = l)$, e um valor $v \in String$, que pode ser vazio $(value(t) = v \vee \epsilon)$ (Falleri et al., 2014).

Desse modo, formalmente, uma modificação de código-fonte é definida pela diferença entre duas *ASTs*, em que AST_i refere-se à árvore sintática abstrata antes da modificação, enquanto AST_o refere-se à estrutura sintática após a modificação, portanto, tem-se $AST_i \ \Delta \ AST_o$. A partir da obtenção das modificações, o próximo passo consiste em identificar as operações que compõem a modificação. Esse processo é conhecido como *Diferenciação de Código-Fonte*.

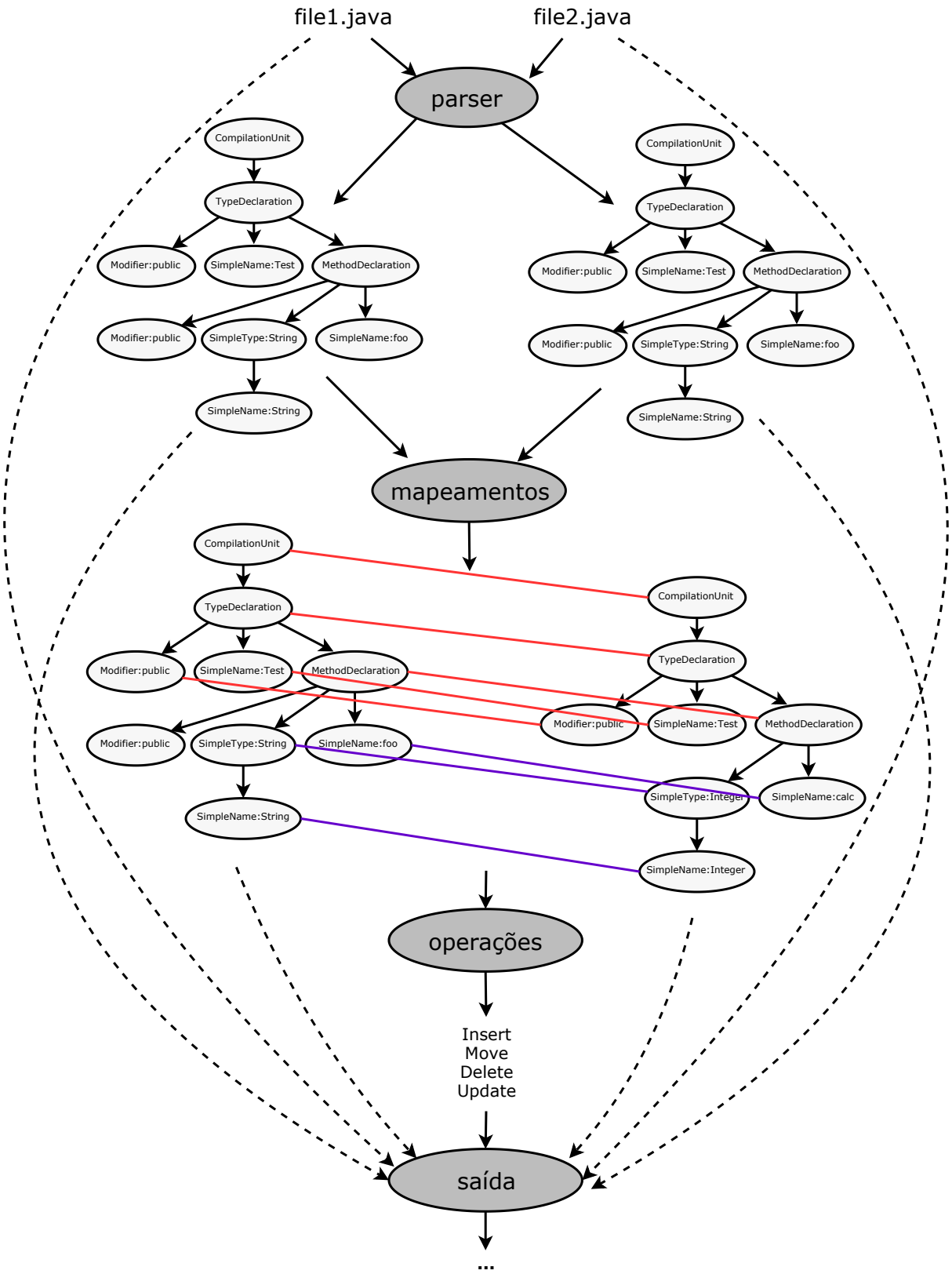
Modificações podem ser obtidas por diferentes estratégias: fornecidos pelos usuários, em exercícios de programação e em repositórios de software. Em repositórios de software, pode-se obter as modificações a partir dos sistemas de controle de versão. Os sistemas comumente empregados permitem o registro de pontos que delimitam cada versão: os pontos de *commits* registram as modificações executadas (Chacon e Straub, 2014). Em cada *commit* são registrados os estados de todos os arquivos (*snapshot*). Desse modo, a partir da comparação de dois *commits* subsequentes, é possível obter as modificações entre as versões, e se efetuado ao longo de todo o histórico de repositório, possibilita a extração de modificações aplicadas no repositório. As operações que compõem a modificação são mantidas pelo sistema de controle de versão, em nível de arquivo.

Similarmente, o processo de diferenciação de código-fonte visa à extração do conjunto de operações aplicadas em uma modificação, responsável por transformar a AST_i na AST_o. Esse conjunto é conhecido como sequência de edição (*edit-script*) (Falleri et al., 2014). De modo geral, esse processo é dividido em duas fases, a *Fase de Mapeamento* e a *Fase de Geração da Sequência de Edição*, como apresentado na Figura 2.2.

Na *Fase de Mapeamento*, o intuito consiste em identificar os pares de elementos sintáticos que têm correspondência a partir das *ASTs*. Nesse processo existem quatro possíveis casos:

- Elementos não-modificados: para os elementos que não foram modificados, o mapeamento vincula um elemento de código na AST_i com outro na AST_o.
- Elementos inseridos: para esses elementos não existe um correspondente na AST_i, desse modo, o elemento é classificado “elemento novo”.
- Elementos removidos: para esses elementos não existe um corresponde na AST_o, desse modo, o elemento é classificado “elemento antigo”.
- Elementos modificados: são elementos que possuem correspondente entre as AST_o e AST_i, contudo o valor associado ao elemento é diferente, desse modo, o elemento é classificado como “elemento atualizado”.

Figura 2.2: Processo de diferenciação.

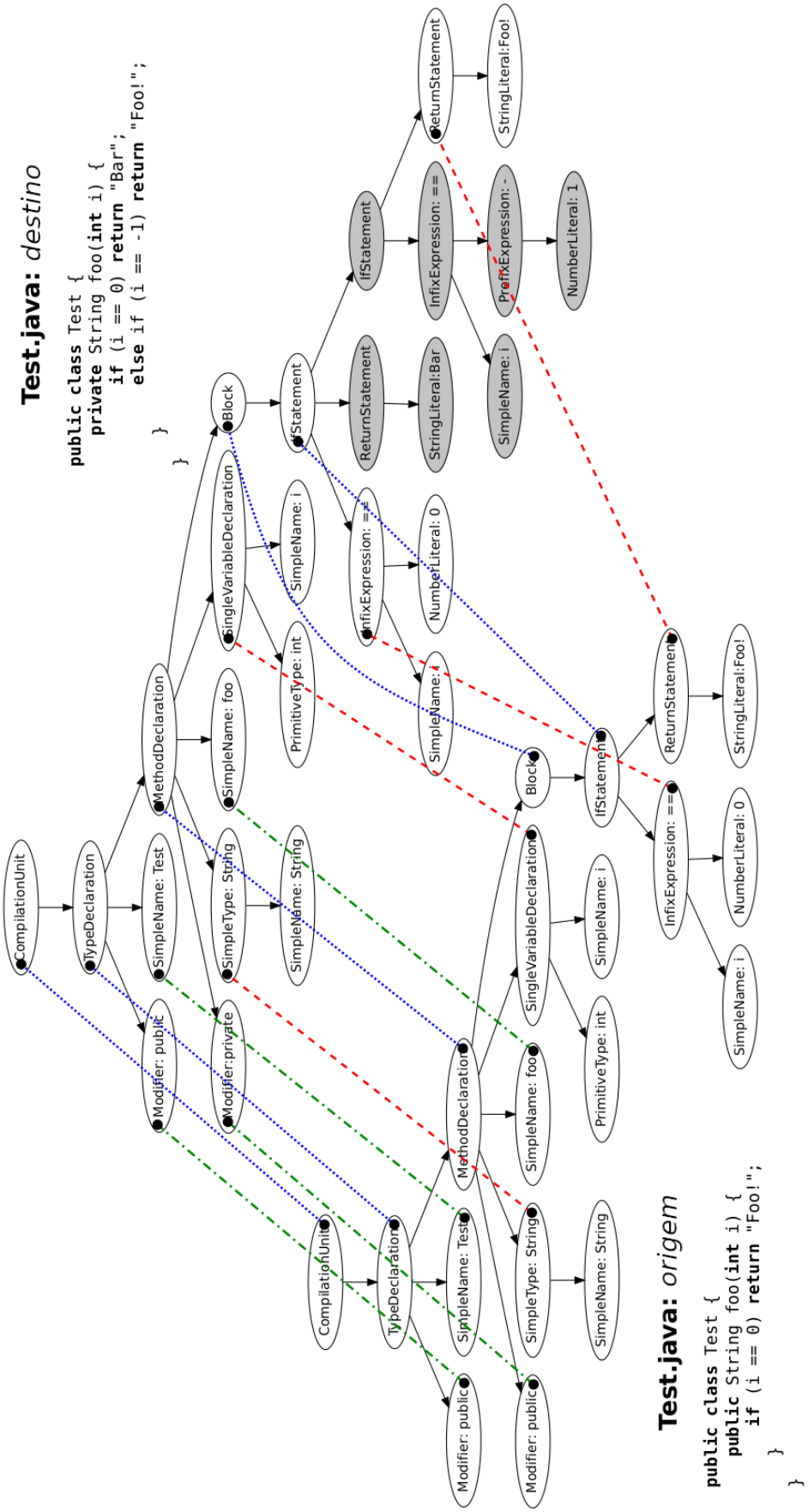


Fonte: Adaptada de Falleri et al. (2014).

Um exemplo do processo de mapeamento entre duas *ASTs* é apresentado na Figura 2.3.

Em seguida, na *Geração de Sequência de Edição*, esses mapeamentos são utilizados para identificar quais alterações foram aplicadas. Nessa fase, a maioria das abordagens de diferen-

Figura 2.3: Mapeamento de duas *ASTs* em versões consecutivas.



Fonte: Adaptada de Falleri et al. (2014).

ciação têm utilizado o algoritmo de Chawathe et al. (1996) para diferenciação de estruturas hierárquicas (Falleri et al., 2014, Dotzler e Philippsen, 2016, Frick et al., 2018). Esse algoritmo tem como base o produto cartesiano $n = max(|T_1|, |T_2|)$ entre as duas *ASTs*, o que resulta em um desempenho de complexidade $O(n^2)$. Como resultado final, a sequência de edição é obtida, a qual, geralmente, é composta pelas seguintes operações:

- $add(t, t_p, i, l, v)$: adiciona o nó t na i-ésima posição da sub-árvore t_p, o qual é rotulado por l e recebe o valor v.
- $move(t, t_p, i)$: movimenta/translada o nó (sub-árvore) identificado por t para a sub-árvore t_p na posição i.
- $update(t, v_n)$: atualiza o valor do nó identificado por t para v_n.
- $remove(t)$: remove o nó ou sub-árvore identificado por t.

A *Fase de Mapeamento* tem relevância, pois um mapeamento incorreto pode influenciar diretamente na sequência de edição, tornando-a mais longa e imprecisa (Dotzler e Philippsen, 2016, Frick et al., 2018).

Na literatura, o principal algoritmo de diferenciação é o *GumTree*, proposto por Falleri et al. (2014), cuja estratégia simula o processo de diferenciação de um desenvolvedor ao analisar uma modificação. O funcionamento do algoritmo é dividido em duas etapas principais:

- Um algoritmo guloso realiza análise *top-down* com intuito de encontrar todas as árvores isomórficas, de modo a estabelecer ligações entre estas.
- Aplica-se um algoritmo *bottom-up* em pares de nós correspondentes de forma a detectar mapeamentos adicionais entre os nós descentes identificados como não-correspondentes na etapa anterior.

O algoritmo apresentado por Chawathe et al. (1996) é utilizado para gerar a sequência de edição. Os algoritmos das etapas de *top-down* e *bottom* são apresentados nos algoritmos 1 e 2, respectivamente.

Na primeira etapa (*top-down*), o algoritmo segue uma estratégia gulosa para a identificação, nas árvores sintáticas T_1 e T_2, da árvore sintática comum (isomórfica) de maior altura, de modo que todas as demais sub-árvores possíveis (também isomórficas) estejam contidas nessa árvore sintática principal.

O processo é iniciado a partir dos nós-raiz de ambas sub-árvores (pois são os nós de maior altura), e são executados recursivamente para o nós-filhos, até que uma sub-árvore seja compatível. Cada par de sub-árvores isomórficas (compatíveis) são armazenadas em uma lista de candidatos a mapeamentos. Em seguida, os mapeamentos da listagem de candidatos são extraídos um a um, e priorizados de modo decrescente conforme a função de dados.

A segunda etapa (*bottom-up*) possui como entrada os mapeamentos identificados na etapa anterior. A partir desses mapeamentos, o primeiro passo consiste em identificar grupos de nós na estrutura sintática que apresentam um número significante de nós mapeados.

Entrada: Árvores sintáticas T_1 e T_2, Listas de
Saída: Mapeamento M
início
$M = \emptyset, A = \emptyset$
$push(root(T_1), L_1), push(root(T_2), L_2)$
enquanto $min(peekMax(L_1), peekMax(L_2)) > alturaMinima$ **faça**
se $peekMax(L_1) \neq peekMax(L_2)$ **então**
se $peekMax(L_1) > peekMax(L_2)$ **então**
para todo $l \in pop(L_1)$ **faça**
$open(t, L_1)$
senão
para todo $t \in pop(L_2)$ **faça**
$open(t, L_2)$
senão
$H_1 = pop(L_1), H_2 = pop(L_2)$
para todo $(t_1, t_2) \in H_1 \times H_2$ **faça**
se $isomorphic(t_1, t_2)$ **então**
se $\exists t_x \in T_2 | isomorphic(t_1, x) \wedge t_x \neq t_2 || \exists t_x \in T_1 | isomorphic(t_x, t_2) \wedge t_x \neq t_1$ **então**
$add(A, (t_1, t_2))$
senão
adicionar todos pares de nós isomórficos em $s(t_1)$ para M
para todo $t_1 \in H_1 | (t_1, t_x) \notin A \cup M$ **faça**
$open(t_1, L_1)$
para todo $t_2 \in H_2 | (t_2, t_x) \notin A \cup M$ **faça**
$open(t_2, L_2)$
$sort(t_1, t_2) \in A$ usando $dice(parent(t_1), parent(t_2), M)$
enquanto $size(A) \neq 0$ **faça**
$(t_1, t_2) = remove(A, 0)$
adicionar todos os pares de nós isomórficos em $s(t_1)$ e $s(t_2)$ para M
$A = A \backslash \{(t_1, t_x \in A)\}, A = A \backslash \{(t_x, t_2 \in A)\}$
fim

Algoritmo 1: Algoritmo *Top-down*.

Dentro de cada grupo, o próximo passo consiste em reduzir o número de nós não-mapeamentos, por meio da busca no conjunto da nós não-mapeados da outra estrutura sintática, considerando aspectos como rótulos e equivalência de nós descentes. Essa análise pode resultar um conjunto com vários candidatos a mapeamentos, cuja escolha é feita pela maior similaridade de mapeamento (dado um valor mínimo de limiar).

Uma das principais contribuições do algoritmo *GumTree* consiste no aprimoramento da identificação de operações de movimentação (*move*), as quais em trabalhos anteriores como Fluri et al. (2007), eram identificadas majoritariamente como deleções seguidas inserções.

Chawathe et al. (1996) propõem uma abordagem para cálculo da sequência de edição, a qual é dividida em duas etapas: o mapeamento, o qual se baseia na similaridade entre nós folha, e também entre nós não-folha; e a geração de sequência de edição mínima, que utiliza os ma-

Entrada: Árvores sintáticas T_1 e T_2 e Mapeamento M
Saída: Mapeamento M
início
  para todo $t_1 \in T_1 | t_1 \notin M \wedge t_1$ *tem nós-filhos mapeados, em pós-ordem* **faça**
    $t_2 = candidate(t_1, M)$
    se $t_2 \neq null$ *e* $Dice(t_1, t_2, M) > minDice$ **então**
      $add(M, (t_1, t_2))$
      se $max(|s(t_1)|, |s(t_2)| < maxSize)$ **então**
        $R = opt(t_1, t_2)$
        para todo $(t_a, t_b) \in R$ **faça**
          se $t_a, t_b \notin M \wedge l(t_a) - l(t_b)$ **então**
            $add(M, (t_a, t_b))$
fim

Algoritmo 2: Algoritmo *Bottom-up*

Entrada: Árvores sintáticas T_1 e T_2
Saída: Mapeamento M
início
  $M = \emptyset$
  para todo $l \in rotulos$ **faça**
    $S_1 = chain(T_1, l), S_2 = chain(T_2, l)$
    $lcs = LCS(S_1, S_2, equal)$
    para todo $(x, y) \in lcs$ **faça**
      $add(x, y)\ para\ M$
    para todo $x \notin M\ \&\ x \in S_1$ **faça**
      se $y \notin M\ \&\ Y \in S_2\ \&\ equal(x, y)$ **então**
        $add(x, y)\ para\ M$
fim

Algoritmo 3: Algoritmo de Mapeamento

peamentos previamente definidos juntamente com as AST de entrada e saída. Os algoritmos utilizados para resolver ambos subproblemas são apresentados nos algoritmos 3 e 4, respectivamente.

O algoritmo 3 define a estratégia para identificação dos mapeamentos entre duas árvores sintáticas, T_1 e T_2, fornecidas como entradas. A partir disso, cada nó-folha da árvore sintática T_1 é comparado a outro da T_2, verificando a compatibilidade de rótulos e se a similaridade é superior a um dado limiar definido, desse modo, definindo um mapeamento entre T_1 e T_2. Para nós não-folha, o processo é conduzido de modo recursivo, sendo que taxa de ocorrência de nós comuns entre as sub-árvores comparadas deve ser de pelo menos 50% ($\lambda > 0.5$). O processo de cálculo de similaridade entre nós tem como base o algoritmo da Subsequência Comum Máxima (*LCS – Longest Common Subsequence*).

O Algoritmo 4 realiza a extração da sequência do edição. Os mapeamentos obtidos anteriormente são dados como entrada, juntamente com as duas árvores sintáticas . Na primeira fase,

Entrada: Árvores sintáticas T_1 e T_2 e Mapeamento M
Saída: Sequência de Edição E
início
$M = M, E = \emptyset$
para todo $x \in T_2$ *por busca em largura* **faça**
$y = p(x)$ e z parceiro de y em M'
se x *sem parceiro em* M' **então**
$k = FindPos(x)$ e criar um no baseado em x
adicionar $Insert(w, z, k)$ para E, adicionar (w, x) para M', e aplicar $Insert(w, z, k)$ para T_1
senão
w = o parceiro de x em M' e v igual ao parceiro de $p(w)$ em M'
se $v(x)$ *differs from* $v(w)$ **então**
adicionar $Update(w, v(x))$ em E e aplicar $Update(w, v(x))$ em T_1
se $(y, v) \notin M$ **então**
$k = FindPos(x)$ e z parceiro de y em M'
adicionar $Move(w, z, k)$ em E e aplicar $Move(w, z, k)$ em T_1
alinhar nós filhos (w, x)
para todo $w \in T_2$ *em percurso pos-ordem* **faça**
se w *nao tem parceiro em* M' **então**
adicionar $Delete(w)$ em E e aplicar $Delete(w)$ em T_1
fim

Algoritmo 4: Algoritmo de Geração de Sequência de Edição

o algoritmo percorre T_2 com percurso em largura. Para cada nó mapeado que não possuem correspondem é registrada uma operação de inserção (*insert*). Caso o par de nós do mapeamento apresente diferenças entre si é gerado uma operação de atualização (*update*). Caso os nós do mapeamento possuam elementos-pai diferentes é gerada uma operação de movimentação (*move*). Por fim, para cada nó pertencente a T_1, que não teve mapeamento com T_2, é gerada uma operação de remoção (*delete*).

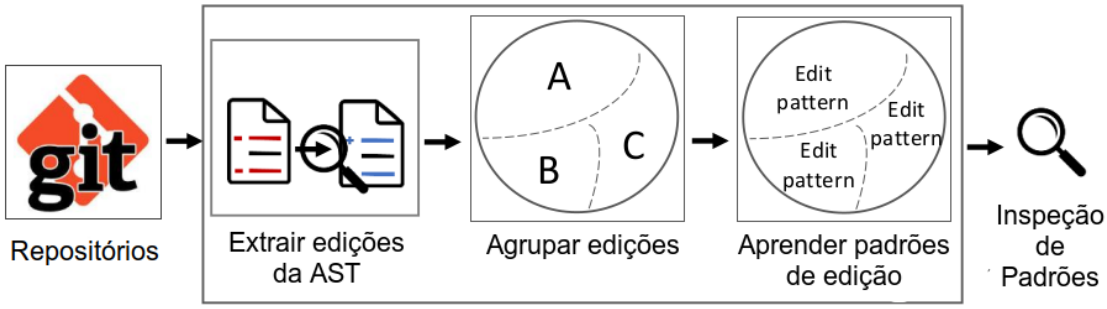
Outros trabalhos também são relevantes neste contexto. Dotzler e Philippsen (2016) apresentam o algoritmo *Move-optimized Tree Differencing (MTDIFF)*, o qual se baseia no algoritmo *Change Distiller* (Fluri et al., 2007), de modo a obter uma sequência de edição mais curta, por intermédio de pequenas otimizações, para resolver imprecisões em situações recorrentes identificadas.

De modo semelhante, Frick et al. (2018) apresentam o algoritmo *Iterative Java Matcher (IJM)* , com o intuito de otimizar a fase de mapeamento e, consequentemente, produzir sequências de edição mais compreensíveis ao desenvolvedor. Para isso, foi elaborado um conjunto de mapeadores parciais, os quais analisam pontos específicos do código-fonte de modo a definir a alteração realizada.

2.3 Aprendizado de Modificações

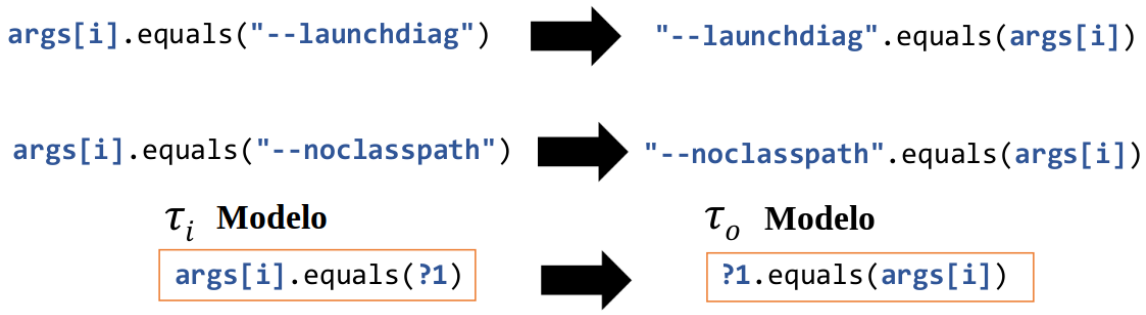
Modificações de código-fonte são aplicadas em diversos pontos do programa (Rolim et al., 2017). Kim e Notkin (2009) apontam que modificações repetidas ou muito similares são

Figura 2.4: Funcionamento da técnica *Revisar*.

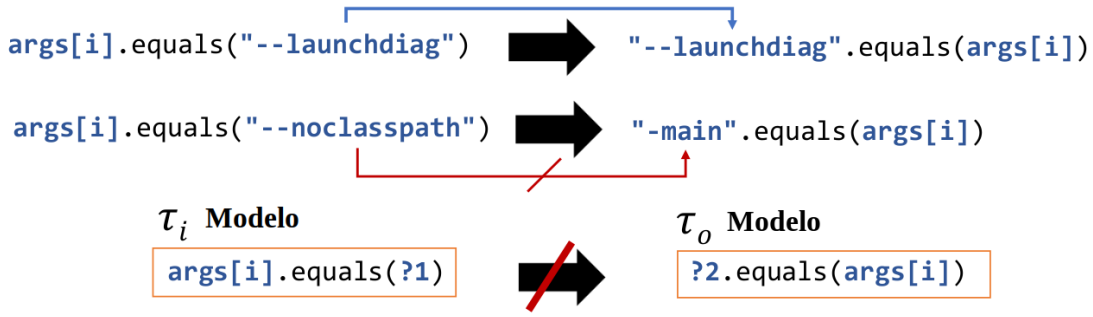


Fonte: Adaptada de Rolim et al. (2018a).

Figura 2.5: Comparativo entre duas modificações de código-fonte.



(a) Modificações compatíveis.



(b) Modificações incompatíveis.

Fonte: Adaptada de Rolim et al. (2018a)

conduzidas de modo sistemático, e estão relacionadas a aspectos estruturais em termos de código-fonte, o que caracteriza a presença de padrões de modificações.

Para confirmar o caráter repetitivo, o estudo de Nguyen et al. (2013) apresentou algumas características comuns de modificações de código-fonte e correções de defeitos em repositórios de software:

- Independente de uma modificação em geral ou uma correção de defeitos, pode-se observar que tanto dentro do mesmo repositório ou entre repositórios, as modificações com menor número de operações tendem a apresentar uma frequência maior, resultando em um relação inversamente proporcional entre o tamanho (em números de operações para

execução da modificação) e quantidade de repetições de cada padrão de modficação, conforme identificado por Nguyen et al. (2013) e apresentado na Figura 2.6.

Cada gráfico apresenta a frequência de repetição (eixo y), em relação ao tamanho da modificação (eixo x), em cada cenário analisado.

- A repetição também depende de aspectos sintáticos, de modo que determinar instruções tem uma maior recorrência, principalmente estruturas mais simples como acesso a vetores, chamadas de método em detrimento de estruturas como laços de repetição ou condicionais.

- A generalização de modificações tem maior frequência e estável (maior similaridade) entre repositórios do que quando um único repositório é analisado de modo isolado.

Nesse contexto, de modo a usufruir o caráter repetitivo e sistemático em diversas modificações de código-fonte, trabalhos têm aplicado diferentes abordagens no aprendizado de padrões de modificações. A técnica *Revisar*, proposta por Rolim et al. (2018a), tem como objetivo a identificação automática de padrões de edição em repositórios de software. O fluxo de execução da técnica é apresentado na Figura 2.4.

O funcionamento inicia com a análise do repositório, em que são obtidas as modificações de código-fonte, no formato de pares de exemplos (entrada-saída). De cada modificação, são extraídas as AST_i e AST_o, e a sequência de edição utilizando o algoritmo GumTree (apresentado na Seção 2.2).

Entrada: Conjunto de modificações concretas $\{(i_1, o_1) \dots (i_n, o_n)\}$
Saída: Conjuntos de modelos
início
  $clusters = \emptyset$
  para todo $(i_k, o_k) in \{(i_1, o_1) \dots (i_n, o_n)\}$ **faça**
    $cluster\_candidates = \emptyset$
    para todo $clusterc \in clusters$ **faça**
      $r = \tau_i \mapsto \tau_o := compute\_template((i_k, o_k), c)$
      se r *é uma transformação compatível* **então**
        $cluster\_candidates.add(c)$
        $cost_c; = add_c ost((i_k, o_k), c)$
    $best = argmin(cluster\_candidates)$
    se *um cluster melhor existe* **então**
      $best.add((i_k, o_k))$
    senão
      $cluster.append(newcluster((i_k, o_k)))$
  Retornar clusters
fim

Algoritmo 5: Algoritmo de Agrupamento Guloso.

Em seguida, cada instância de modificação é analisada de modo a agrupar modificações sintaticamente semelhantes. A partir de cada grupo, um padrão sintático comum é obtido entre

as instâncias, para generalizar todas as instâncias presentes no mesmo agrupamento (Rolim et al., 2018b).

Como exemplo, na Figura 2.5 são apresentadas quatro modificações de código-fonte em que é realizada a chamada do método `equals()` para a comparação entre um literal e uma variável do tipo de dados `String`.

No primeiro caso (Figura 2.6(a)), as duas primeiras modificações realizam a inversão entre variável e literal, permanecendo os mesmos elementos do programa. Essa modificação se enquadra na categoria de "preferir um literal na utilização do método `equals()`", desse modo, é possível criar um padrão (entrada – saída). Contudo, no segundo caso (Figura 2.6(b)) a modificação insere um novo literal `--main"`, que não estava presente no contexto inicial da modificação e não é igual ao literal anterior `--noclasspath"`, desse modo não é possível extrair um padrão dessa modificação.

Um modelo (*template*) é extraído de cada padrão sintático, representando a estrutura sintática da modificação-base, porém com a substituição de todos os identificadores e constantes. Formalmente, um modelo τ é uma AST em que os nós folha podem ser *holes*, de modo que uma AST t seja compatível somente se é possível atribuir todos os valores concretos de t nos *holes* definidos em τ e a AST resultante é identica a original. No exemplo da Figura 2.6(a), os literais `--launchdiag` e `--noclasspath` são substituídos pelo *hole* `?1`. A definição dos agrupamentos nessa técnica é feito pelo Algoritmo 5.

A partir do conjunto de modificações identificadas, extrai-se o modelo de cada padrão sintático para sua representação. Em seguida, para cada modelo verifica-se se há um ou mais agrupamentos compatíveis com a modificação, em caso positivo, seleciona o agrupamento cuja transformação tenha menor custo; caso negativo, cria-se um novo agrupamento em que modelo é adicionado. Com o término, após a análise de todas as modificações concreta, obtém-se um conjunto de agrupamentos de modificações semelhantes, os quais formam os padrões de modificação.

Outros trabalhos relacionados ao aprendizado de padrões de modificação são apresentados, mas não detalhados. Meng et al. (2013) propõem a técnica *LASE* (*Locating and Applying Systematic Edits*), que objetiva auxiliar desenvolvedores no processo de evolução de programas pela aplicação de operações sistemáticas. Essa técnica, a partir do fornecimento de exemplos, extrai a generalização mais específica de cada modificação. Com a análise do contexto em que as respectivas modificações foram aplicadas, são identificados os potenciais trechos do código-fonte em que as alterações podem ser aplicadas. A etapa de recomendação é baseada em detecção de clones e análise de dependências.

Rolim et al. (2017) propõem a técnica *Refazer*, que tem como objetivo a síntese de transformações de programa com base em exemplos. Essa técnica utiliza como *framework PROSE* (Polozov e Gulwani, 2015), comumente utilizado para a definição de exemplos e especificações. Nessa técnica, cada modificação é definida por um conjunto de regras de reescrita, que por sua vez, possuem uma operação que pode ser aplicada a alguns pontos no código-fonte. A descrição completa de cada modificação é definida por uma linguagem de domínio específico (*DSL* –

Domain Specified Language). O processo de síntese de modificações se baseia no agrupamento e ranqueamento de cada uma das modificações identificadas.

2.4 Considerações Finais

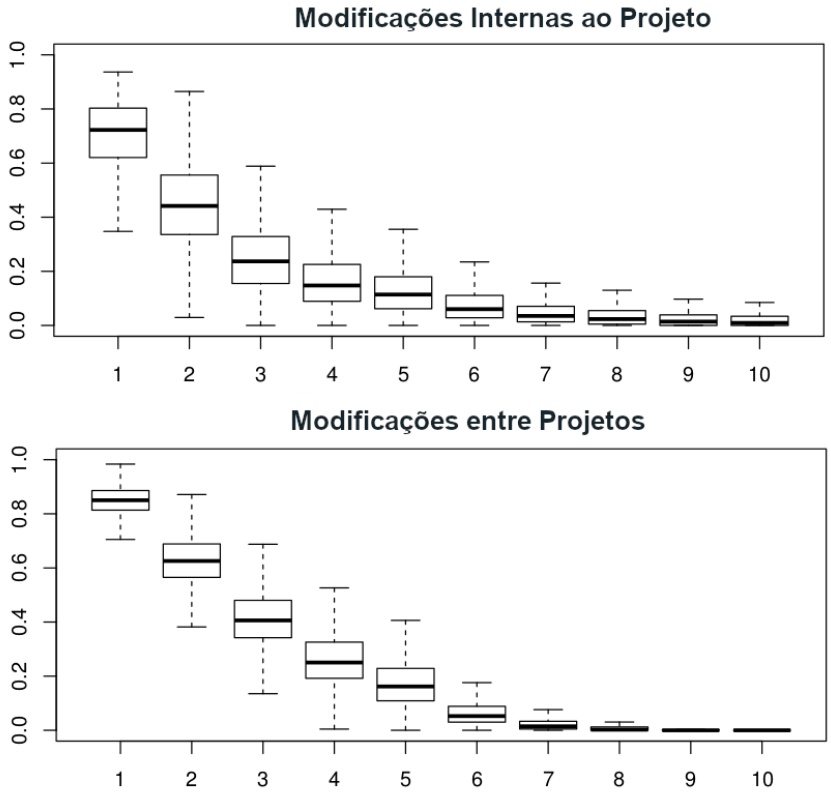
Neste capítulo foi apresentado o contexto principal deste trabalho quanto a Mineração de Repositórios de Software, em que, visa-se modificações de código-fonte, principalmente modificações adaptativas, como principal exemplos as refatorações de software, as quais permitem a reestruturação do código, de melhorar a organização interna do software, contudo sem impacto observável de comportamento.

Dentre os estudos de detecção de oportunidades de refatoração, as abordagens comumente se baseiam em regras predefinidas ou heurísticas, o que restringe a amplitude das refatorações detectadas. Por outro lado, alguns trabalhos na literatura – principalmente relacionados a modificações corretivas – têm se utilizado do processo de diferenciação de código-fonte para extração do passo a passo da execução da modificação e sua estrutura. Uma vez extraídos esses dois elementos, padrões de modificações relacionadas a extraída são obtidos, compondo, desse modo, um padrão de modificação considerada um defeito.

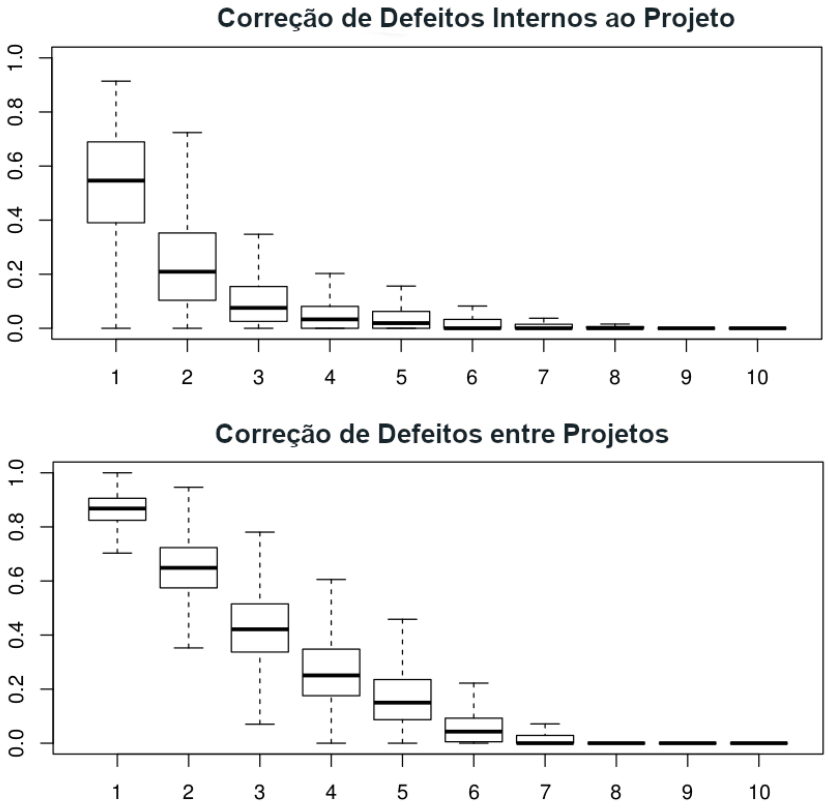
No presente trabalho, a intenção consiste em um utilizar a mesma estratégia, porém voltada para refatorações e modificações adaptativas em geral, de modo a avaliá-las com base no impacto de cada com base na manutenabilidade.

Contudo, antes disso, faz-se necessário o entendimento sobre Manutenabilidade de Software, além de algumas abordagens que possibilitam sua mensuração, que são descritas no próximo capítulo.

Figura 2.6: Repetição de modificações e correções de defeitos em detrimento do tamanho.



(a) Modificações de Código.



(b) Correção de Defeito.

Fonte: Adaptada de Nguyen et al. (2013)

CAPÍTULO

3

Manutenabilidade de Software

Durante o desenvolvimento – e ao longo da vida de um software –, modificações para a implementação de funcionalidades desejadas, melhorias ou correção de defeitos têm impacto direto na qualidade interna do software, na maioria das vezes aumentando sua complexidade (Sharma et al., 2015).

Um grande desafio no processo evolutivo consiste em como agregar novas funcionalidades a um sistema de software, oriundas da alteração ou adição de requisitos, sem prejudicar a estruturação interna do software. Essa tarefa está diretamente relacionada à propriedade conhecida como *Manutenabilidade de Software* (ISO/IEC-9126, 2001).

A fase de manutenção de um projeto é muito crítica, devido a sua duração e custos envolvidos. Desse modo, o quanto antes desenvolvedores considerarem aspectos relacionados a manutenabilidade, menores tendem a ser os impactos na qualidade do software e custos financeiros (Dubey e Rana, 2011).

Nesse contexto, as métricas de software são instrumentos que permitem a mensuração de aspectos relativos à qualidade do software e ao código-fonte. A utilização de métricas está comumente relacionada ao monitoramento e à avaliação de características do software, seja para mensuração de propriedades (tamanhos, complexidade, etc.), identificação e/ou predição da ocorrência de falha e defeitos, ou outros elementos de código-fonte, por meio de modelos estatísticos ou de aprendizado de máquina.

No presente trabalho, as métricas (especificamente relacionadas ao códigofonte) têm papel fundamental na mensuração dos impactos de cada modificação no código-fonte do software, possibilitando a identificação de padrões de modificações que, recorrentemente, contribuam para a melhoria ou não da qualidade do código-fonte e, consequentemente, da manutenabilidade do software.

Nesse sentido, o capítulo encontra-se organizado do seguinte modo:

- na Seção 3.1 são apresentados alguns conceitos e trabalhos relacionados à qualidade de

código-fonte, principalmente quanto a métricas de qualidade de código-fonte.

- na Seção 3.2 são apresentados modelos de avaliação de qualidade e manutenabilidade de código-fonte presentes na literatura.

3.1 Métricas de Software

Conforme um software cresce em número de recursos computacionais oferecidos ao usuário, também tende a aumentar sua complexidade e a importância do projeto de software. Nesse cenário, as consequências de quaisquer alterações são diretas sobre os atributos de qualidade do software, como flexibilidade, testabilidade, manutenibilidade, desempenho e segurança (Meirelles, 2013). Segundo Del Esposte (2014), a complexidade consiste no principal fator de risco de um software, uma vez que a evolução é comprometida por falta de legibilidade e flexibilidade do sistema.

O monitoramento desses atributos é uma atividade fundamental e pode apoiar técnicas de desenvolvimento que promovam a melhoria contínua do código-fonte. Essa atividade pode ser apoiada pelo uso de *Métricas de Software*.

Métricas de software expressam um valor associado a um dado atributo ou propriedade do software. A sua empregabilidade em estudos na literatura é multivariada, como exemplo, na identificação de propensão a defeitos em arquivos com base no histórico do repositório (Liu et al., 2018), a detecção de defeitos e suas respectivas correções (Osman et al., 2014), a análise de tipos de modificações específicas como métodos clonados (Islam e Zibran, 2018), e a construção de modelos de predição de defeitos por meio da combinação de métricas e algoritmos de aprendizado de máquina (Yuan et al., 2013, Nisa e Ahsan, 2015).

Em geral, as métricas de software podem ser divididas em duas categorias: métricas de produto de software e métricas de processo de software. A primeira categoria está relacionada aos artefatos de software, tais como o código-fonte e a documentação; enquanto a segunda categoria foca na carga de trabalho necessária para a condução das atividades do processo de software (Li e Henry, 1993). Somente métricas relacionadas a primeira categoria (métricas de produto de software) são consideradas neste trabalho, devido ao objeto de análise do estudo, no caso o código-fonte ao longo de diversas versões do histórico.

Dentre as principais métricas presentes na literatura que visam a mensuração da complexidade de um software, podem ser observadas diferentes estratégias. As métricas de Halsted (1977) e Bail e Zelkowitz (1988) utilizam de medidas baseadas em componentes léxicos, enquanto outras, como a métrica de complexidade ciclomática de McCabe (McCabe, 1976), utilizam-se do fluxo de controle do programa em funções/procedimentos. As métricas acima citadas têm como foco o paradigma procedural de programação.

Na orientação a objetos, as métricas mensuram a relação entre os elementos de software, de modo a observar os principais aspectos do paradigma como herança, polimorfismo e encapsulamento. Nesse contexto, as métricas mais amplamente referenciadas são as métricas de CK, propostas por Chidamber e Kemerer (1994), as quais permitem a análise precisa de aspectos como complexidade, coesão e acoplamento.

Outros trabalhos relevantes também podem ser destacados, como as métricas propostas por Lorenz e Kidd (1994), as quais permitem a extração de características estáticas do projeto do software, principalmente quanto ao tamanho e herança das classes, e a presença/ausência de problemas no código-fonte. Abreu e Brito (1995) definiram um conjunto de métricas para mensurar o uso de mecanismos relacionados a orientação a objetos como herança, ocultação de atributos e métodos, polimorfismo e acopladas.

As métricas utilizadas no presente trabalho abrangem as métricas propostas por Chidamber e Kemerer (1994), as quais estão descritas na seção a seguir.

3.1.1 Métricas de CK

As métricas propostas por Chidamber e Kemerer (1994) consistem em um conjunto total de seis métricas. Cada métrica avalia um aspecto do sistema de software, sob a visão principalmente focada em aspectos de projeto, ao invés da implementação (Dubey e Rana, 2011). Apesar disso, esse conjunto de métricas tem se demonstrado útil, tanto para projetistas quanto desenvolvedores, pois permite a avaliação do sistema a nível de arquivos (Harrison et al., 1998).

Cada uma métricas de CK está explicada brevemente a seguir:

- **Depth Inheritance Tree – DIT**: essa métrica mensura o nível de profundidade de uma classe dentro de uma hierarquia de herança. Por meio da herança, classes podem compartilhar atributos e comportamentos de uma para outra de modo hierárquico, desse modo, quanto maior número de classes envolvidas, maior o número de elementos compartilhados, resultando em um código-fote mais complexo de ser manutenido.

  O nível de profundidade é medido com base nos níveis da hierarquia, desse modo, a classe raiz assume valor 0, enquanto as demais classes variam até N, sendo N inteiro e positivo.

- **Number of Children – NOC**: essa métrica mensura a quantidade de elementos-filhos (atributos e métodos) diretos de uma classe específica. Conforme maior o número de elementos de uma classe, a tendência é que a classe seja maior e mais complexa, e consequentemente, um maior número de classes pode depender da lógica implementada nessa classe.

  A contagem de elementos pode variar desde 0 a N, sendo N um número positivo e inteiro.

- **Response For Class – RFC**: essa métrica mede a cardinalidade do conjunto de resposta de uma classe. Um conjunto de resposta consiste em todas as chamadas a métodos locais e de outras classes. Quanto maior o número de métodos envolvidos em uma iteração, maior pode ser a dificuldade de identificar a origem de um determinado erro.

  O cálculo da métrica é feito através da soma todas as chamadas de métodos únicos dentro de uma classe, o que pode variar de 0 a N, sendo N inteiro e positivo.

- **Lack of Cohesion of Class – LCOM**: essa métrica mensura a falta de coesão entre os elementos de uma classe. A coesão é caracterizada pela interligação forte dos métodos

com os atributos da classe, logo uma classe que os elementos seja fracamente interligados, ou mesmo, possam ser agrupados em conjuntos desconexos, tem baixa coesão.

O cálculo da métrica é obtido pelo número de conjuntos disjuntos de métodos que podem ser obtidos na classe. Para que dois métodos pertençam ao mesmo conjunto, deve haver pelo menos uma variável ou atributo em comum.

- **Weighted Method per Class – WMC**: essa métrica mensura a complexidade estática da lógica implementada em todos os métodos da classe. Uma classe com maior número de métodos, e com métodos mais complexos individualmente (grande quantidade de instruções de controle de fluxo), tende a ser mais difícil de ser manutenida.

  A obtenção do valor da métrica é feito pela somatória da complexidade ciclomática de McCabe (McCabe, 1976) de cada método da classe. O resulado obtido compreende um valor de 0 a N, sendo N inteiro e positivo.

- **Coupling Between Objects – CBO**: essa métrica mensura o nível de acoplamento de uma classe com as demais. De modo geral, essa métrica permite avaliar o quanto uma classe é dependente dos métodos e atributos de outras classes. A ocorrência de classes muito acopladas pode ser consequência de uma tentativa de reaproveitamento de código, contudo isso pode atrapalhar a modularização do software, tornando os componentes do software mais dependentes entre si. Adicionalmente, em decorrência dessa dependência, o impacto de uma modificação em um módulo pode se propagar para outras partes do sistema, podendo resultar na introdução de defeitos.

  O cálculo dessa métrica é feito pela contagem do número de classes acopladas à classe em análise, cujo valor pode variar de 0 a N, sendo N inteiro e positivo.

Um ponto de destaque quanto ao conjunto de métricas de CK, consiste em sua utilização em diversos trabalhos na avaliação de manutenabilidade de software. Inclusive, Dubey e Rana (2011) aponta uma relação inversamente proporcional da medidas extraídas pelas métricas de CK e o nível de complexidade de um sistema. Ou seja, conforme a um aumento nos valores dessas métricas, registra-se uma redução na manutenabilidade do software.

Por fim, deve ser salientado que uma métrica não pode ser utilizada de modo isolado, porque essa não é capaz de quantificar todas as características relevantes de um sistema de software. Desse modo, um conjunto de métricas deve ser utilizando, permitindo a avaliação sob diferentes perspectivas (Malhotra e Khanna, 2013).

3.2 Modelos de Manutenabilidade de Software

Os avanços no processo de desenvolvimento de software têm produzido sistemas maiores e mais complexos. Como consequência, surge a necessidade monitoramento qualitativo desse, de modo a evitar perda de qualidade e garantir de modo constante sua manutenabilidade. Apesar da sua relevante importância, avaliar a manutenabilidade de um software ainda é uma tarefa desafiadora (Dubey e Rana, 2011).

Segundo Seref e Tanriover (2016), uma grande variedade de técnicas têm sido aplicadas na mensuração e predição de manutenabilidade de software, desde algoritmos estatísticos e de aprendizado de máquina, técnicas baseadas na natureza, em avaliações de especialistas e abordagens híbridas. A relação entre métricas de software e manutenabilidade também tem sido investigada em diversos estudos.

Seref e Tanriover (2016) observaram que, em sistemas de software baseados em linguagens imperativas, existe a predominância de métricas mais tradicionais, como pontos por função, métricas de Halstead (Halsted, 1977) e a métrica de Complexidade Ciclomática de McCabe (McCabe, 1976). Dentre os principais trabalhos, tem-se como destaque o Índice de Manutenabilidade inicialmente proposto por Oman e Hagemeister (1992).

Com o advento da orientação a objetos, as métricas tradionais perderam relevância em detrimento da análise características como herança, coesão, acoplamento e polimorfismo. A partir disso, a necessidade de um conjunto de métricas que suprisse tais características foi destacado por (Rombach, 1990). Nesse contexto, as Métricas de CK (Chidamber e Kemerer, 1994) foram propostas e se tornaram amplamente utilizadas em diversos estudos (Seref e Tanriover, 2016).

Mediante a variedade de estratégias e estudos propostos neste âmbito, na seções 3.2.1 e 3.2.2 são apresentadas duas abordagens que utilizam estratégias baseadas mensuração de manutenabilidade de software no contexto do paradigma de programação imperativo e orientado a objetos, respectivamente.

3.2.1 Índice de Manutenabilidade

O Índice de Manutenabilidade (*Manutenability Index*) é uma das métricas mais bem estabelecidas na literatura para a avaliação de manutenabilidade de software. Essa métrica foi introduzida por Oman e Hagemeister (1992), com o intuito de mensurar a facilidade de manutenção de um programa. A fórmula inicialmente proposta do MI está representada na Equação 3.1:

$$MI = 171 - 5.2 * ln(aveV) - 0.23 * ave(G) - 16.2 * ln(aveLOC) \tag{3.1}$$

Em que, são definidos como:

- $aveV$ – volume médio de *Halsted*;
- $ave(G)$ – complexidade ciclomática média;
- $aveLOC$ – número médio de linhas de código-fonte.

O cálculo do Índice de Manutenabilidade é dependente do tamanho do programa, que é expresso pelo número de linhas, juntamente com duas métricas de complexidade. A métrica do volume de *Halstead* se refere ao número de operandos e operadores existentes em cada arquivo; enquanto a complexidade ciclomática indica a quantidade de fluxos de execução possíveis (Molnar e Motogna, 2017).

De modo adicional, na literatura foram propostas algumas variações do Índice de Manutenabilidade, as quais consideram aspectos adicionais como comentários (Coleman et al., 1994)

e diferentes limiares de precisão nos pesos de cada fator, como na implementação em softwares como Visual Studio (Microsoft, 2011).

Porém a utilização do Índice de Manutenabilidade também possui ressalvas a serem consideradas. Os estudos e ferramentas que o implementam se baseiam diferentes fórmulas para cálculo do índice, seja em termos de limiares e parâmetros considerados. Um outro ponto consiste que as linguagens de programação, desde a criação do MI, passaram por grandes modificações, sendo agregados novos recursos e alterações no paradigma de programação, em que fatores como estrutura, coesão e acoplamento são relevantes (Welker, 2001, Heitlager et al., 2007, Molnar e Motogna, 2017).

Adicionalmente, o cálculo do índice é comumente feita para análise a nível de pacotes ou mesma aplicação com um todo a cada versão, considerando um nível de granularidade alto. Essa perspectiva resulta em uma avaliação muito genérica, o que pode não contribuir na identificação dos elementos que impactaram na manutenabilidade do sistema (Molnar e Motogna, 2017).

3.2.2 Variação de Manutenabilidade

Molnar e Motogna (2017) apresentaram uma abordagem para identificação de modificações significativas em sistemas orientados a objeto, por meio da análise de atributos como profundidade da árvore de herança, encapsulamento e nível de acoplamento entre módulos.

A estratégia da abordagem consiste em identificar tais modificações entre versões, a nível de classes, e prover uma representação númerica simples. A seleção de métricas se baseia no conjunto de métricas de CK, dentre as quais são adotadas as métricas *DIT*, *WMC*, *CBO* e *LCOM*; de modo que cada uma represente as características de tamanho/estrutura, complexidade, coesão e acoplamento em um software, respectivamente.

O maior contraste em relação às demais estratégias reside em avaliar o impacto da manutenabilidade que ocorre durante o desenvolvimento do software. Desse modo, o algoritmo proposto recebe como entrada os valores das métricas indicadas de cada uma das classes presentes em duas versões do software, visando comparar o impacto das modificações por meio de um valor númerico inteiro. Um resultado positivo indica um aumento na manutenabilidade, enquanto um negativo indica redução. No Algoritmo 6 é apresentado o pseudocódigo do algoritmo da abordagem.

O funcionamento do algoritmo consiste na comparação, versão a versão, de todas as classes do sistema. Caso uma classe não apareça em uma das versões comparadas, o valor atribuído é zero. Em versões que uma determinada classe não está presente, o valor adotado é zero para cada uma das métricas. Para cada classe, se o valor de uma métrica aumenta entre as versões, então a pontuação de manutenabilidade é reduzida em uma unidade, em caso de redução é aumentada em uma unidade, e se não for alterada, a pontuação também é mantida. Cada uma das métricas avaliadas tem o mesmo peso sob a pontuação final do arquivo em uma dada versão.

Os resultados obtidos na avaliação da abordagem indicaram uma maior sensibilidade do cálculo da Variação da Manutenabilidade quanto a ocorrência de modificações em detrimento do Índice de Manutenabilidade. Além disso, foi identificada uma maior precisão da abordagem em versões com ocorrência de um grande número de inserções/deleções de código-fonte.

Entrada: (sv_1, sv_2): par de versões
$ms = \{DIT, WMC, CBO, LCOM\}$: conjunto de métricas
$m(c, sv)$: valor da métrica para classe c na versão sv, dado que:
se $(c \in sv_2) and (c \notin sv_1)$ **então**
$m(c, sv_1) = 0$

Saída: Variação de Manutenabilidade

início
mc = 0 **para todo** $metricinms$ **faça**
para todo $classc \in sv_1 \cup sv_2$ **faça**
se $m(c, sv_1) < m(c, sv_2)$ **então**
$mc = mc - 1$
se $m(c, sv_1) > m(c, sv_2)$ **então**
$mc = mc + 1$
Retornar mc
fim

Algoritmo 6: Algoritmo de Variação de Manutenabilidade.

3.3 Considerações Finais

Neste capítulo foi apresentado o conceito de manutenabilidade de software, e como essa tem sido estimada por meio da análise do código-fonte. Diversos estudos têm estudo a correlação de manutenabilidade e métricas de software. O Índice de Manutenabilidade, proposto por Oman e Hagemeister (1992), consiste em uma das abordagens de mensuração de manutenabilidade mais amplamente utilizadas e avaliadas na literatura. Ao mesmo tempo, estudos apontam críticas quanto a capacidade de avaliação do Índice de Manutenabilidade, tanto em sua definição, como limitações em sistemas baseados no paradigma orientado a objetos.

De modo a suprir tais limitações, estudos têm investigado a manutenabilidade em sistemas orientados a objetos, como o estudo de Molnar e Motogna (2017), o qual visa identificar modificações de código-fonte significativas quanto a manutenabilidade de software, utilizando métricas relacionadas a propriedades relevantes da orientação a objetos.

Desse modo, a identificação de padrões de modificações (descrita no Capítulo 2), juntamente com abordagens de avaliação da manutenabilidade podem permitir a identificação de modificações presentes, de modo recorrente, em arquivos/versões cuja manutenabilidade tenha sido aumentado, assim como modificações que comumente ocorrem em momentos de redução de manutenabilidade. A combinação desses elementos permite a elaboração de um processo de aprendizado de modificações de código-fonte, de modo a classificá-las entre melhorias e degradações, cujo processo está descrito no capítulo a seguir.

CAPÍTULO

4

Processo de Aprendizado de Padrões Sintáticos

Neste capítulo é apresentado o processo de aprendizado de padrões de modificações de código-fonte, obtidos por meio de modificações extraídas do histórico de versão de repositórios de software desenvolvido por esse trabalho.

Como visto nos capítulos anteriores, os estudos relacionados a modificações adaptativas como refatorações apresentam limitações em decorrência da estratégia adotada para identificação de modificações. Nesse sentido, o processo proposto combina algoritmos de diferenciação de código-fonte e modelos de mensuração de manutenabilidade, para a identificação de padrões sintáticos de modificação que propiciem a melhoria do código-fonte, assim como, de modo complementar, de modificações que resultem na degradação do código-fonte. A visão geral do processo é apresentada na Figura 4.1.

De modo geral, o processo consiste na obtenção das modificações realizadas entre duas versões (no caso deste trabalho são considerados *commits*). Paralelamente, um conjunto de métricas de software relacionadas ao código-fonte é extraído, dos arquivos de ambas versões, e utilizado para o cálculo da manutenabilidade, e consequentemente, identificar a variação da manutenabilidade resultante. Essa variação, dependendo das modificações implementadas, pode ser resultado do impacto de uma ou mais modificações. Para identificar quais modificações realmente tiveram impacto sob a manutenabilidade é aplicado um modelo de regressão.

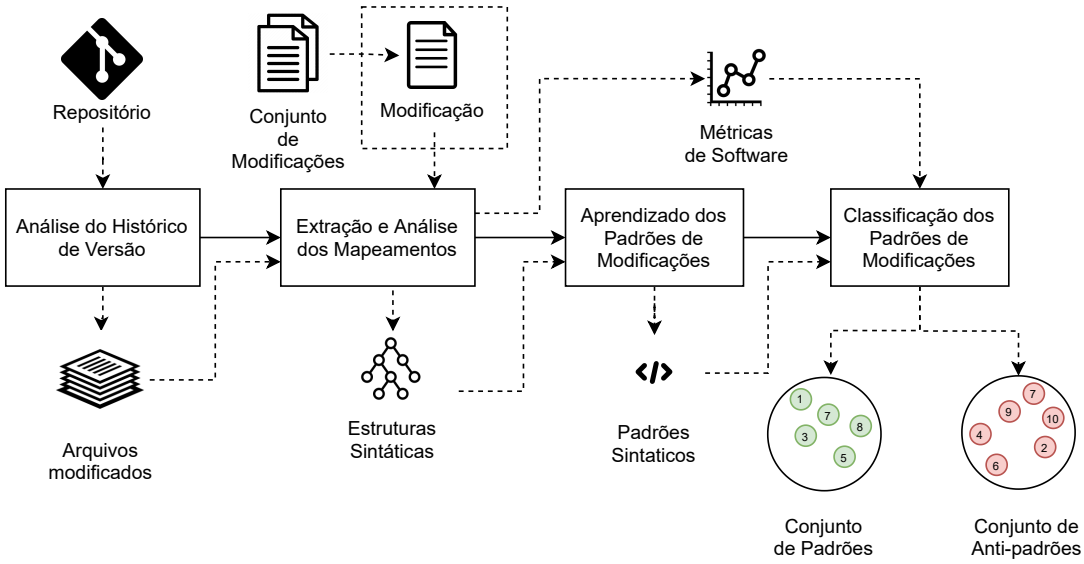
A análise dos repositórios de software foi conduzida com apoio da ferramenta de linha de comando *Learning Code Extractor*, que foi implementada neste projeto, cujo código-fonte está disponível para acesso[1].

O passo a passo do processo é descrito mais detalhadamente nas seções a seguir:

- Na Seção 4.1 é detalhada a etapa de análise do histórico de versão dos repositórios para a

[1]Disponível em: https://github.com/leandroungari/learning-code-extractor/

Figura 4.1: Visão geral do processo de aprendizado de modificações.



Fonte: Elaborada pelo autor.

extração dos arquivos modificados;

- Na Seção 4.2 é descrita a etapa de identificação e análise dos dos mapeamentos das modificações;
- Na Seção 4.3 é descrito o processo de aprendizado dos padrões sintáticos;
- Na Seção 4.4 é explicitada a estratégia de classificação dos padrões sintáticos com base nas métricas extraídas.

4.1 Análise do Histórico de Versão

A primeira etapa do processo consiste na análise do repositório de código-fonte para obtenção dos arquivos modificados em cada *commit*. O fluxo geral dessa etapa é apresentada na Figura 4.2.

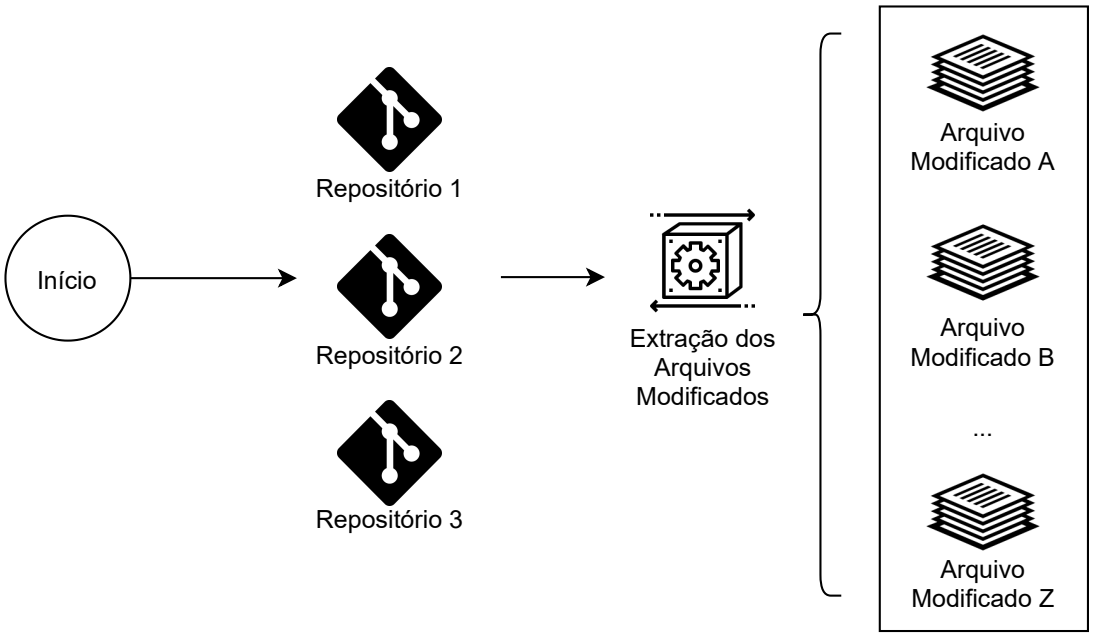
Um fator relevante é que o conjunto de repositórios selecionados neste estudo é baseado no sistema de controle de versão Git [2], que permite a consulta do registro total e detalhado das modificações conduzidas ao longo do histórico.

O processo se inicia, em cada repositório, com a extração dos *commits* realizados na principal *branch* em cada repositório, comumente a *branch master* ou equivalente. Em seguida, obtém-se a relação dos arquivos modificados em cada *commit* para utilização na próxima etapa. Cada arquivo modificado, possui um conjunto de uma ou mais modificações, as quais são extraídas na próxima etapa.

A identificação dos arquivos modificados foi feita por meio da biblioteca *JGit* [3], a qual permite a extração de dados e a manipulação do repositório. Por meio dessa biblioteca, cabe destacar que também é possível identificar as modificações realizadas em cada arquivo, contudo

[2]Disponível em: https://git-scm.com/
[3]Disponível em: https://www.eclipse.org/jgit/

Figura 4.2: Análise do histórico de repositórios.



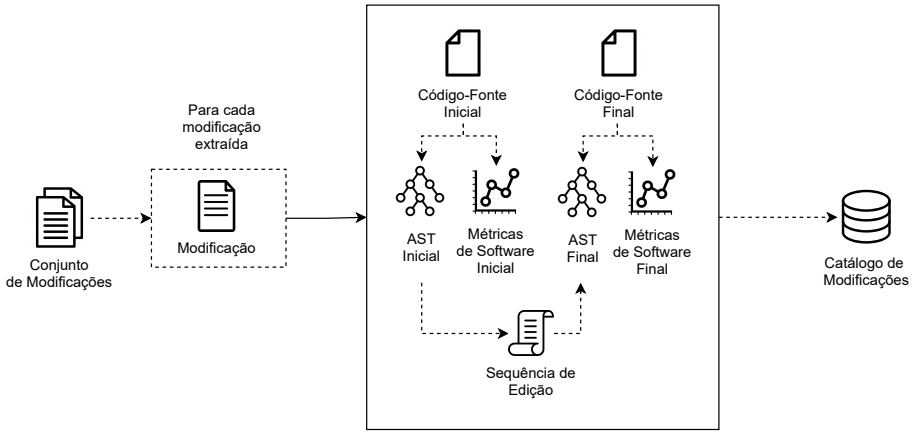
Fonte: Elaborada pelo autor.

a granularidade da representação das modificações oferecida, no caso diferenciação baseada em linhas, não foi empregada neste trabalho, pois foi adotadas a diferenciação baseada em árvore sintática.

4.2 Extração e Análise dos Mapeamentos

A segunda etapa consiste na extração e análise das modificações obtidas dos arquivos em cada *commit*, de modo que sejam construídos os mapeamentos, assim como a obtenção das métricas de software. Na Figura 4.3 é apresentada uma visão geral da condução dessa etapa.

Figura 4.3: Extração e análise de modificações.



Fonte: Elaborada pelo autor.

Nas seções a seguir, são apresentadas as atividades que compõem essa etapa de modo detalhado.

4.2.1 Extração dos Mapeamentos

A partir da relação de arquivos modificados de cada *commit*, as árvores sintáticas abstratas referentes ao arquivo modificado e sua versão prévia (*commit* imediatamente anterior) são construídas para a comparação entre ambas. Em cada árvore sintática T, o algoritmo de diferenciação *GumTree* é aplicado com base T_{v-1} e T_v, para identificar os mapeamentos de inserção, deleção, atualização e movimentação existentes.

Cada mapeamento é composto pelos elementos sintáticos envolvidos antes (E_{v-1}) e depois (E_v) da modificação aplicada. O par ordenado (E_{v-1},E_v) representa a estrutura sintática de uma determinada modificação, o qual é utilizado posteriormente para identificação de padrões de sintáticos.

Na extração dos mapeamentos foi utilizada a biblioteca *Gumtree Spoon AST Diff* [4], que fornece um cojunto de classes para manipulação das *ASTs*. A extração dos mapeamentos é feita por meio da análise do código-fonte de cada arquivo, em ambas versões AST_{v-1} e AST_v, das classes modificadas.

Durante a condução da extração de mapeamentos nos repositórios iniciais, foram observadas algumas deficiências de identificação. Nesse sentido, foram implementados processamentos adicionais e melhorias em relação à implementação do algoritmo pela biblioteca, com o objetivo de aprimorar os mapeamentos identificados e torná-los mais próximos das modificações de código-fonte reais. Tais melhorias e processamentos são descritos a seguir:

4.2.1.1 Conversão de mapeamentos deleção/inserção para atualização

O primeiro aprimoramento consiste na conversão de mapeamentos de deleção seguidos de mapeamentos de inserção em um único mapeamento de movimentação, em casos que as operações se referem ao mesmo elemento sintático. Esse ajuste foi necessário em situações em que, como exemplificado na Figura 4.4, uma entidade de código-fonte é movida entre arquivos (ou mesmo dentro do mesmo arquivo), porém são registrados dois mapeamentos resultantes (deleção/inserção). Essa falha de identificação de mapeamento teve maior ocorrência em modificações entre arquivos.

Em alguns mapeamentos foi observado também que os mapeamentos se referiam ao mesmo elemento sintático, contudo havia pequenas alterações no código-fonte como ajustes de variáveis, uma nova instrução, entre outros. Nesse caso, a comparação direta (estritamente igual) não identificaria que os mapeamentos se referiam ao mesmo elemento, desse modo, foi adotado um limiar de precisão que varia entre 75% a 85% de similaridade (número de lexemas comuns entre os trechos de código-fonte) para verificação de compatibilidade.

A solução aplicada consistiu na verificação de correspondência entre os elementos modificados dos mapeamentos de inserção e deleção dentro do mesmo *commit* entre todos os arquivos. Em caso de correspondência do mesmo elemento sintático, foi realizada a união do mapeamento de deleção e inserção em um mapeamento de movimentação, resultando em uma representação mais próxima da conduzida pelo desenvolvedor.

[4]Disponível em: https://github.com/SpoonLabs/gumtree-spoon-ast-diff

Figura 4.4: Conversão para mapeamento de atualização.

```
  //antes da modificacao
  public class A {
-   public void metodoA() {
-   ...
-   }
  }
  //depois da modificacao
  public class A {
    ...
  }
```

```
  //antes da modificacao
  public class B {
    ...
  }
  //depois da modificacao
  public class B {
+   public void metodoA() {
+     ...
+   }
  }
```

Fonte: Elaborada pelo autor.

Figura 4.5: União de mapeamento relacionados.

```
  public void exemplo() {
-   int temp = 0;    //renomeia a variavel
+   int soma = 0;
    for (var i = 0; i < 10; i++) {
-     temp += i;     //modifica a atribuicao
+     soma += i;
    }
-   System.out.println(temp);
+   System.out.println(soma);    //substitui a variavel
  }
```

Fonte: Elaborada pelo autor.

4.2.1.2 Agrupamento de mapeamentos relacionados

A segunda melhoria conduzida consistiu no agrupamento de mapeamentos com dependências e elementos sintáticos relacionados. Essa melhoria foi necessária em situações que, como exemplificado na Figura 4.5, uma mesma modificação de código-fonte (não-contínua em termos de linhas de código) é dividida em dois ou mais mapeamentos, porém com elementos sintáticos relacionados.

Nesse contexto, a melhoria aplicada consistiu em verificar entre mapeamentos sintaticamente próximos a existência de dependências entre as partes envolvidas. As dependências verificadas consistiram variáveis, atributos de classe e parâmetros em comum; e também relação hierárquica em termos de bloco de código, como estruturas de seleção e laços de repetição. Caso existam dependências em comum, os mapeamentos são unidos.

4.2.1.3 Composição de mapeamentos baseado nos elementos sintáticos

A terceira melhoria consistiu na composição dos mapeamentos em um conjunto específico de tipos de sintáticos. Esse ajuste foi necessário para a composição de modificações estruturalmente significantes, devido a ocorrência de mapeamentos relacionados a modificações com granularidade muita baixa (como alterações de operadores, adição de parenteses, etc.), as quais não definem uma modificação com semântica completa.

Na Figura 4.6 é ilustrada a representação de um cenário de aplicação dessa melhoria. A modificação realizada na linha (3-4) inicialmente geraria um mapeamento indicando a substi-

Figura 4.6: Exemplo de composição de mapeamento.

```
  public void exemplo() {
    int temp = 0;
-   for (var i = 0; i <= 10; i++) {
+   for (var i = 0; i < 10; i++) {
      temp += i;
    }
    ...
  }
```

Fonte: Elaborada pelo autor.

tuição do operador condicional $\leq$ por $<$, contudo o impacto da modificação não está restrita somente a esse elemento. Desse modo, é aplicado uma estratégia recursiva de caminhamento ascendente na AST até que seja identificado o primeiro elemento significante. Nesse exemplo, após o operador sintático, o próximo elemento a ser avaliado seria uma expressão, contudo está também não é significativa. Nesse caso, o processo deve ser repetido até encontrar nó do elemento sintático de declarações (*statement*).

Os tipos de elementos sintáticos considerados na composição foram:

- Classes, Enumeráveis e Interfaces;
- Atributos, Métodos e Construtores;
- Declarações e Anotações;

4.2.2 Análise dos Mapeamentos

O processo de análise dos mapeamentos consiste na extração de um conjunto de métricas de software dos arquivos modificados em cada *commit*. A métricas selecionadas são referentes ao conjunto de Métricas de CK, as quais focam em aspectos relevantes a este estudo, principalmente quanto a complexidade e propriedades do paradigma de orientação a objetos.

O processo consiste em, para cada mapeamento, são extraídas as métricas de cada arquivo, em ambas as versões. Um aspecto importante consiste que as métricas utilizadas estimam propriedades à nível de arquivo. Isso permite maior precisão quanto a avaliação da qualidade do código-fonte, se comparado a abordagens de maior granularidade (pacotes ou versões).

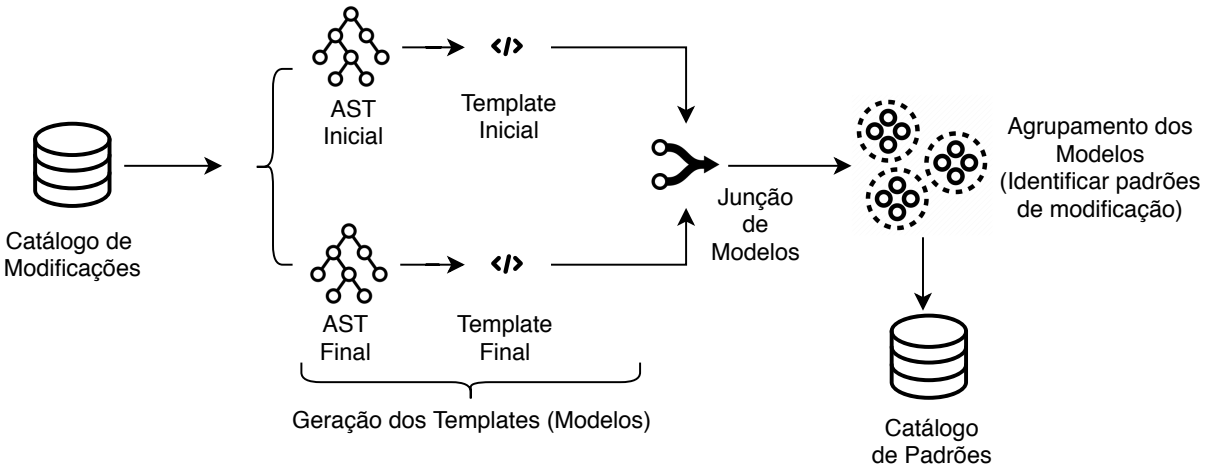
Na sequência, as medidas extraídas são fornecidas para o cálculo da variação da manutenabilidade nos arquivos envolvidos em cada mapeamento. O cálculo é feito utilizando o algoritmo proposto por Molnar e Motogna (2017) (apresentado na Seção 3.2.2), em que, como entrada, são fornecidas as métricas de antes e depois da modificação em cada arquivo, e como resultado, a saída indica a tendência de aumento ou diminuição da manutenabilidade nos arquivos.

Por fim, os valores de variação da manutenabilidade de cada mapeamento são armazenados para uso na avaliação de impacto dos padrões de modificações.

4.3 Aprendizado dos Padrões de Modificações

A terceira etapa consiste no aprendizado de padrões sintáticos de modificações de código-fonte, com base nos mapeamentos previamente identificados. Na Figura 4.7 é apresentada uma visão geral dessa etapa.

Figura 4.7: Extração e agrupamento dos modelos.



Fonte: Elaborada pelo autor.

Inicialmente, dada as *ASTs* abstratas dos elementos E_{v-1} e E_v envolvidos em cada mapeamento, obtém-se os modelos sintáticos de cada um, os quais combinados formam o padrão sintático da modificação. O padrão sintático consiste junção das *ASTs* em ambas as versões (antes e depois da modificação), desconsiderando os nomes de variáveis e símbolos literais, os quais são substituídos por referências.

Na sequência, é realizado o agrupamento de modificações com padrão sintático compatível. Inicialmente, é definido um conjunto vazio de agrupamentos para armazenamentos do padrões de modificações identificados. Para cada mapeamento é verificado se existe um grupo de modificações cujo padrão sintático é compatível. A verificação de compatibilidade é feita por um processo de caminhamento em todos os nós de ambas *ASTs* verificando a igualdade da função sintática de cada um dos nós. Somente se todos forem iguais, as *ASTs* são consideradas compatíveis. Caso seja encontrado um agrupamento compatível, o mapeamento é adicionado ao conjunto de mapeamentos vinculados ao padrão. Caso contrário, um novo agrupamento é criado, cujo padrão é definido pelo padrão sintático do mapeamento correspondente. Esse processo adotada uma estratégia gulosa, resultando em um primeiro (e único) agrupamento compatível para cada padrão sintático.

O Algoritmo 7 representa o processo descrito acima, o qual se baseia no algoritmo proposto por Rolim et al. (2018a) (descrito na Seção 2.3).

Na implementação desse algoritmo, algumas otimizações foram adicionadas quanto a busca e comparação de padrões sintáticos referentes ao mesmo tipo de elemento sintático (ex. somente classes com classes, métodos com métodos).

Ao final desse etapa, o resultado é composto pela relação de padrões sintáticos encontrados,

Entrada: $map = \{m_1, m_2, ..., m_n\}$: conjunto de mapeamentos
Saída: Conjunto de agrupamentos
início
    $groups = \{\}$
    para todo $minmap$ **faça**
        $flag_add = false$
        $template_i = extract(m \rightarrow ev_i)$
        $template_o = extract(m \rightarrow ev_o)$
        para todo $group \in groups$ **faça**
            se $(template_i, template_o)$ *é compatível com* $group \rightarrow base$ **então**
                $add(group, m)$
                $continue$
        se $flag_add == false$ **então**
            $add(groups, new\_group(m))$
    Retornar $groups$
fim

Algoritmo 7: Algoritmo de extração dos padrões de modificação.

juntamente com a lista de mapeamentos compatíveis com cada.

4.4 Classificação dos Padrões de Modificações

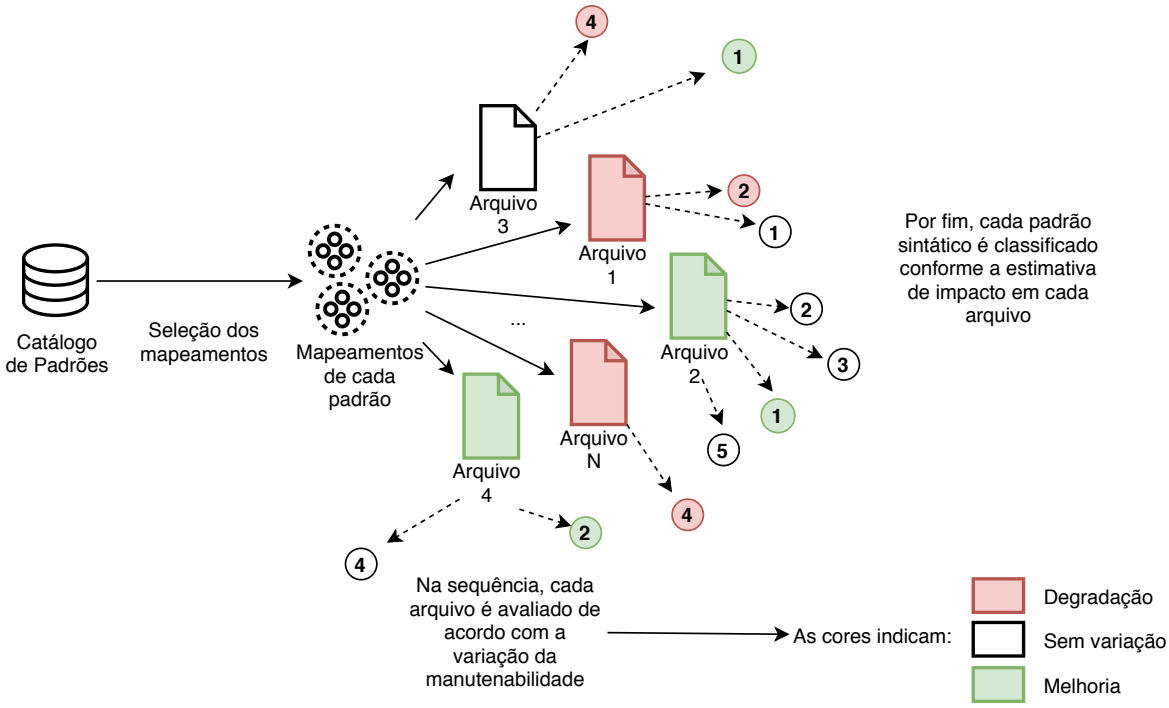
Por fim, a última etapa consiste na classificação dos padrões de modificações entre melhorias e degradações de código-fonte. O processo de classificação utiliza os produtos das duas etapas anteriores, o conjunto de padrões sintáticos (cada qual com o conjunto de mapeamentos relacionados) e as medidas de manutenabilidade de cada mapeamento. A visão geral dessa etapa está representada na Figura 4.8.

O primeiro passo para a classificação consiste no cálculo do impacto médio na manutenabilidade. Basicamente, o cálculo consiste na obtenção da média da variação de manutenabilidade do conjunto de mapeamentos para cada padrão. O resultado representa o impacto médio dos padrões nos respectivos arquivos de ocorrência.

Para representação visual e análise estatística foi adotada a representação por *box-plot*. Nessa representação, cada gráfico *box-plot* é obtido pela distribuição da variação da manutenabilidade das ocorrências nos arquivos de cada padrão sintático. Dado que, o domínio dos resultados de variação está restrito majoritariamente ao intervalo de $[-4, 4]$ em decorrência do cálculo de variação de impacto, foram definidos os seguintes intervalos, proporcionalmente iguais, para a classificação de impacto nas seguintes categorias:

- $\Delta man \leq -2$: Degradação forte;
- $-2 < \Delta man < 0$: Degradação fraca;
- $\Delta man = 0$: Sem impacto;
- $0 < \Delta man < 2$: Melhoria fraca;

Figura 4.8: Classificação da manutenabilidade dos padrões sintáticos.



Fonte: Elaborada pelo autor.

- $\Delta man \geq 2$: Melhoria forte.

Cabe ressaltar que, para os diferentes repositórios, foram considerados os padrões sintáticos com maior número de ocorrências dentre os disponíveis.

Contudo, a classificação inicial somente permite avaliar os padrões sintáticos quando ao comportamento dos arquivos de ocorrência, não permitindo avaliar individualmente cada padrão. Como exemplo, dois padrões sintáticos podem ter ocorrências em diversos arquivos em comum, e tais arquivos podem indicar uma tendência de melhoria no código-fonte. Contudo, existe a possibilidade de um padrão consistir em uma melhoria e outro uma degradação, o que poderia caracterizar uma falha de classificação.

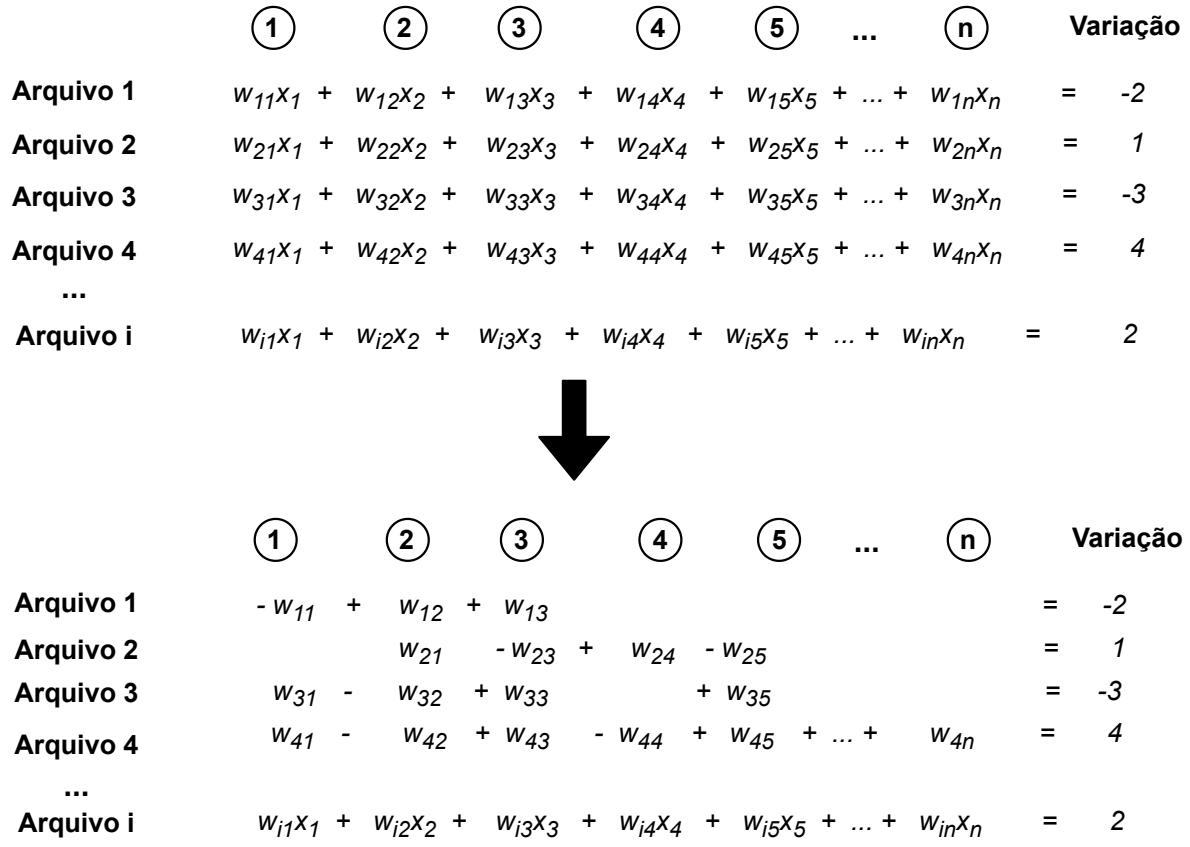
Nesse sentido, para avaliar individualmente o impacto de cada padrão sintático, optou-se pela utilização de um modelo de regressão, de modo a estimar a influência de cada padrão separadamente. A aplicação do modelo de regressão tem como objetivo a identificação das influências (pesos) das ocorrências dos padrões sintáticos em cada arquivo. O modelo de dados (base para o conjunto de treinamento e teste) foi construído com base na estratégia representada na Figura 4.9.

O modelo é basicamente composto pelo conjunto de conjunto de arquivos modificados do repositório ao longo do histórico de *commits*. Cada arquivo consiste em um entrada (linha) do modelo, que pode ser composta por melhorias ($\uparrow$), degradações ($\downarrow$) e modificações sem impacto ($-$).

Arquivo: $w_{11}x_1(\uparrow) + w_{12}x_2(\downarrow) + w_{13}x_3(-) + \cdots + w_{1n}x_n(\uparrow) = -1$

Onde:

Figura 4.9: Modelo de classificação dos padrões sintáticos.



Fonte: Elaborada pelo autor.

- w_{mn} – impacto (peso) de cada modificação no arquivo.
- x_n – presença (1), ausência (0) ou movimentação (-1/1).

Em caso de não-ocorrência, o valor atribuído é 0. Caso contrário, deve ser considerado o tipo de mapeamento registrado, em que mapeamentos de inserção ou deleção atribui-se o valor 1. Para os mapeamentos de atualização e movimentação, deve ser verificado se a modificação envolve diferentes arquivo; se sim então deve-se atribuir valor 1 ao arquivo de destino e -1 ao arquivo de origem (ambos relativos ao mesmo *commit*); caso contrário, deve-se atribuir o valor 1 ao respectivo arquivo.

A implementação do modelo adotou o algoritmo *XGBoost* (Chen e Guestrin, 2016). Esse algoritmo utiliza uma estratégia de gradiente descente hierárquico, que permite estimar a influência de diferentes elementos sob o resultado final. A seleção do algoritmo *XGBoost* ocorreu em decorrência da compatibilidade com modelo de dados esparsos e ponderados, como utilizado neste trabalho, além da alta empregabilidade em diferentes problemas de aprendizagem, sendo considerados em um muitos problemas como algoritmo estado da arte.

O código-fonte referente ao processo de construção e processamento do modelo de dados, juntamente com todos os dados coletados está disponível para consulta [5].

[5]Disponível em:http://bit.ly/2YXCxEM

4.5 Considerações Finais

Neste capítulo foi apresentado o processo de aprendizado de padrões sintáticos de modificações de código-fonte. Por meio desse processo objetiva-se a classificação de padrões de modificações entre melhorias ou degradações; ou seja, deseja-se identificar modificações que apresentem relevância, tanto de impacto quanto de repetição, sobre propriedades relativas a manutenabilidade de software. Cada padrão foi avaliado quanto a um modelo de variação de manutenabilidade que utiliza um conjunto de métricas de orientação a objetos para análise do impacto das modificações conduzidas.

No próximo capítulo é apresentada a condução desse processo em conjunto de repositório de código-aberto, assim como, a análise dos resultados e principais descobertas obtidas.

CAPÍTULO

5

Resultados

Neste capítulo são apresentados os resultados obtidos a partir da análise de cada um dos repositórios selecionados. Todos os mapeamentos foram extraídos de cada um dos repositórios, referentes às modificações de código-fonte realizadas ao longo de um período do histórico de versão. O objetivo dessa análise consiste em identificar quais tipos de modificações de código-fonte compõem os possíveis padrões sintáticos, assim como, seus respectivos impactos sob a manutenabilidade de software.

Cabe ressaltar que, para cada repositório, foram analisados conjunto de *commits* de tamanhos diferentes, em decorrência do número de modificações e distribuição dessas ao longo do histórico. De modo geral, para os repositórios menores foram analisados todo o histórico, enquanto para os demais repositórios, foi selecionado um período específico.

Com a obtenção dos mapeamentos, o próximo passo consiste no aprendizado dos padrões sintáticos mais relevantes (em quantidade de ocorrências) e analisá-los quanto ao impacto na manutenabilidade geral e nas métricas de software individualmente.

Nesse sentido, este capítulo encontra-se organizado do seguinte modo:

- na Seção 5.1 é descrito o conjunto de repositórios utilizados para a avaliação do processo apresentado neste estudo, juntamente com as informações relativas ao ambiente de execução.
- na Seção 5.2 é apresentada uma análise geral do processo de extração dos padrões sintáticos em todos os repositório.
- na Seção 5.3 são apresentadas as análises dos estudos de caso de cada um dos repositórios individualmente.
- na Seção 5.4 são analisados alguns pontos principais em relação ao processo, apresentando as principais descobertas, assim como limitações e possíveis soluções de modo a proporcionar melhores resultados.

5.1 Conjunto de Dados e Ambiente de Execução

O conjunto de dados neste estudo é composto por repositórios de software de código-aberto, implementados na linguagem de programação Java e que utilizam o sistema de controle de versão Git. Esses repositórios foram selecionados devido a sua recorrência em avaliações de estudos relacionados, assim como pela qualidade de codificação dos projetos.

A lista de repositórios está descrita na Tabela 5.1.

Tabela 5.1: Conjunto de repositórios.

Repositório	**Total de Commits**	**Commits Analisados**
Refactoring Toy Example [1]	65	65
Gson [2]	1,4k	1,4k
Truth [3]	1,4k	1,4k
JUnit 4 [4]	2,4k	2,4k
JUnit 5 [5]	6,2k	2,7k
CheckStyle [6]	9,5k	2k
FindBugs [7]	15,3k	2,1k

Fonte: Elaborada pelo autor.

A execução das análises de cada repositório foi conduzida em duas fases: a primeira de extração de mapeamentos e aprendizado dos padrões sintáticos; enquanto a segunda de classificação dos padrões sintáticos quanto ao impacto sob a manutenabilidade. O formato de execução foi por meio da submissão de trabalhos a um *cluster* distribuído, cuja configuração de *hardware* é a seguinte:

- CentOS Linux release 7.7
- 2x Intel®Xeon®CPU E5-2690 @2.90GHz ou 2x Intel®Xeon®CPU E5430 @2.66GHz
- 32 GB RAM

Para cada repositório, foram executados trabalhos para as seguintes tarefas: a extração dos mapeamentos da modificações, a extração das métricas de software de cada arquivo modificado, identificação dos agrupamentos de modificações que formam os mapeamentos, e por fim, a execução do modelo de regressão. O tempo de análise necessário para repositório variou de um dia até mais de duas semanas, conforme o tamanho dos repositórios e o número de mapeamentos (e consequentemente padrões sintáticos) identificados.

[1] https://github.com/danilofes/refactoring-toy-example
[2] https://github.com/google/gson
[3] https://github.com/google/truth
[4] https://github.com/junit-team/junit4
[5] https://github.com/junit-team/junit5
[6] https://github.com/checkstyle/checkstyle
[7] https://github.com/findbugsproject/findbugs

5.2 Análise do Processo de Aprendizado dos Padrões Sintáticos

Nesta seção é a apresentada a análise do processo de aprendizado dos padrões sintáticos, com o objetivo de detalhar os resultados gerais obtidos em cada etapa do processo.

A primeira etapa, a *Etapa de Extração e Análise de Mapeamentos*, consistiu na identificação das modificações realizadas ao longo do histórico de cada repositório e obtenção dos respectivos mapeamentos dessas modificações. Nesse processo foi obtido um conjunto total de 110 711 instâncias de mapeamentos, considerando todos os repositórios. Na Tabela 5.2 está descrita a quantidade total e por tipo de mapeamento obtidas de cada repositório.

Tabela 5.2: Quantidade de mapeamentos por tipo extraídos dos repositórios.

Repositório	**Inserção**	**Remoção**	**Moviment.**	**Atualiz.**	**Total**
Refactoring Toy Example	101	65	88	14	268
Gson	8260	4529	2094	4876	19759
Truth	5493	3106	1845	4076	14520
JUnit 4	12547	7037	2835	6961	29380
JUnit 5	8136	4573	1755	7950	22414
CheckStyle	7113	5020	3194	9043	24370
FindBugs	12250	4818	2505	2535	22108

Fonte: Elaborada pelo autor.

Na etapa seguinte, a *Etapa de Identificação dos Padrões Sintáticos*, os mapeamentos foram agrupados conforme a similaridade estrutural, resultando em um total de 2 405 padrões sintáticos com duas ou mais ocorrências. A Tabela 5.3 descreve o número de padrões obtidos em cada repositório.

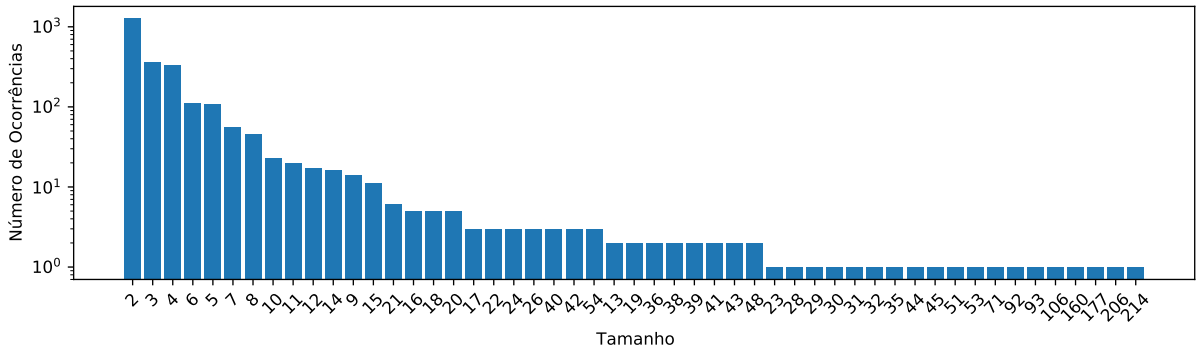
Tabela 5.3: Quantidade de padrões sintáticos por repositório.

Repositório	**Quantidade**
Refactoring Toy Example	60
Gson	686
Truth	253
JUnit 4	708
JUnit 5	227
CheckStyle	372
FindBugs	99

Fonte: Elaborada pelo autor.

O conjunto total de padrões sintáticos apresentou uma comportamento variado, tanto em instâncias com alto número de ocorrências (padrões com mais de 200 ocorrências), quanto em um elevado número de padrões sintáticos com um pequeno número de ocorrências. De modo a representar esse comportamento, a distribuição do número de padrões sintáticos pelo número de ocorrências está representado na Figura 5.1.

Figura 5.1: Quantidade de padrões sintáticos pelo número de ocorrências.



Fonte: Elaborada pelo autor.

Por fim, a última etapa, a *Etapa de Classificação dos Padrões Sintáticos*, cada um padrões sintáticos sintáticos foi analisado quanto ao comportamento médio dos arquivos de ocorrência. Contudo, o comportamento médio dos arquivos quanto a manutenabilidade pode não ser preciso para a avaliação do impacto. Nesse sentido, foi implementado um modelo de regressão para identificar o impacto individual (pesos) de cada padrão nos arquivos de ocorrência, e consequentemente, eliminar possíveis influências de outras modificações. O modelo de regressão, esse foi executado individualmente para cada repositório. A composição da amostra de treinamento do modelo utilizou 80% das instâncias de padrões, enquanto as demais instâncias foram destinadas ao processo de teste do modelo.

O cálculo da taxa de erro foi realizado utilizando o método dos mínimos quadrados, resultando no erro absoluto. Na sequência, a obtenção do erro percentual foi realizada pela divisão do erro absoluto por intervalo de possíveis valores do conjunto-imagem da variação da manutenabilidade, nesse caso, o intervalo de $[-4, 4]$.

De modo geral, a taxa de erro obtida em cada repositório foi satisfatória, sendo essa inferior a 8% em todos os casos. Nos repositórios com pelo menos mil *commits* analisados, a taxa de erro foi significativamente menor, de modo que, somente o repositório *Refactoring Toy Example* apresentou erro superior a 7%, enquanto os demais foram próximos ou inferiores a 2%. Os erros absolutos e percentuais de cada repositório estão descritos na Tabela 5.4.

Tabela 5.4: Taxa de erro do modelo de regressão em cada repositório.

Repositório	**Erro Absoluto**	**Erro Percentual**
Refactoring Toy Example	0,5910	7,38%
Gson	0,1671	2,08%
Truth	0,1423	1,77%
JUnit 4	0,0563	0,70%
JUnit 5	0,0135	0,17%
CheckStyle	0,1437	1,79%
FindBugs	0,0055	0,07%

Fonte: Elaborada pelo autor.

A partir da execução do modelo de regressão, o impacto individual de cada padrão sintático foi identificado. O impacto é resultado da combinação do impacto médio do arquivo (x_i) e a

influência (w_{in}) do padrão no respectivo arquivo, os padrões que não apresentaram influência na manutenabilidade do arquivo ($w_{in} = 0$) foram descartados. Desse modo, como resultado final, foram obtidos 74 padrões sintáticos com impacto relevante na variação da manutenabilidade, com base no processo de classificação. Nesse sentido, podemos observar uma redução significativa em detrimento do total, resultando em um conjunto de somente 3,07%.

O reduzido número de padrões decorreu, em parte, devido a alta taxa de modificações por arquivo em cada um dos repositórios. Por meio da análise das ocorrências dos padrões ao longo do histórico de versão dos repositório, foi possível identificar que, em média, os arquivos com instâncias de padrões apresentaram cerca de 7,91 modificações por arquivo. O repositório que apresentou a menor taxa de modificações por arquivo foi o repositório *Refactoring Toy Example* com aproximadamente 2,69 modificações por arquivo, enquanto a maior taxa pelo repositório *Gson* com 15,85. Essas modificações adicionais podem influenciar na variação de manutenabilidade, e consequentemente, aumentar a imprecisão da estimativa do impacto de cada padrão sintático.

5.3 Estudo de Caso

Nesta seção é apresentado um estudo de caso, que consiste em uma análise mais detalhada quanto aos padrões sintáticos e suas respectivas modificações identificadas em cada repositório. Em cada estudo de caso, uma visão geral é apresentada sobre o projeto do respectivo repositório, e também quanto ao comportamento da manutenabilidade ao longo do histórico analisado. Em seguida, os principais padrões sintáticos são analisados quanto ao tipo da modificação e o impacto resultante.

5.3.1 Refactoring Toy Example

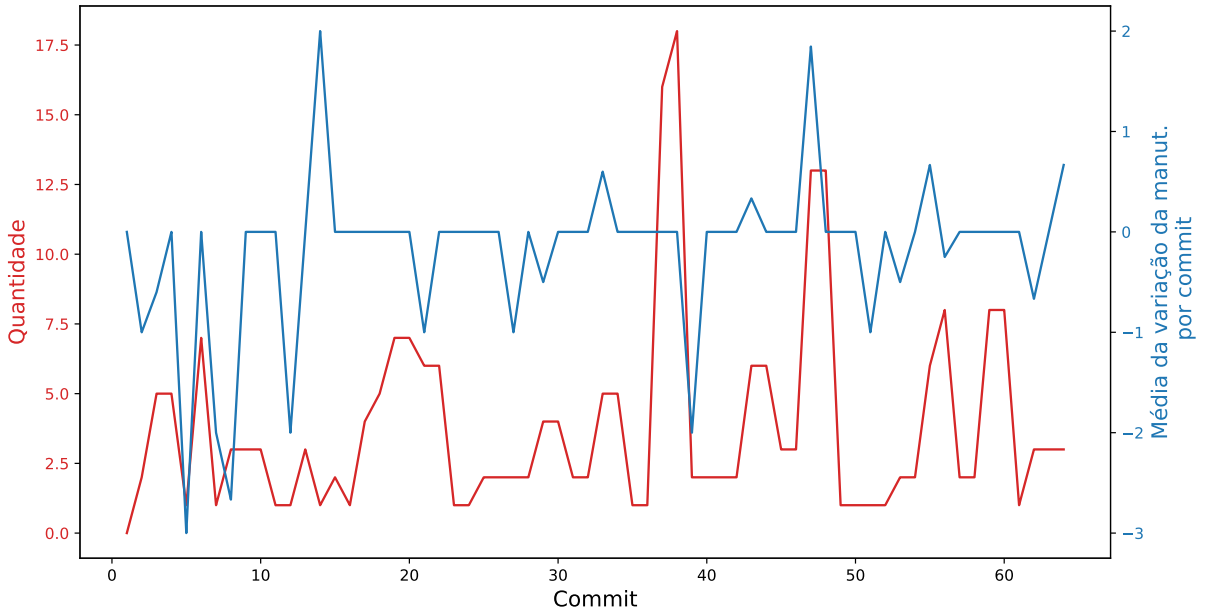
O repositório *Refactoring Toy Example* foi implementado no estudo de Silva e Valente (2017), o qual consiste em um projeto de instâncias de refatorações de software. Em seu estudo original, o projeto foi utilizado como base para o comparativo de técnicas para identificações de pontos suscetíveis a aplicação de refatorações. No presente estudo, a análise desse repositório tem como objetivo a verificação do impacto de alguns tipos específicos de refatorações. A totalidade de *commits* da *branch* master foi analisada.

O primeiro aspecto a ser analisado consiste no comportamento da variação da manutenabilidade do repositório. A relação da variação média da manutenabilidade e o número de mapeamentos de modificações em cada versão é apresentada na Figura 5.2.

Nesse sentido, pode-se observar um comportamento estável com um baixo número de modificações ao longo do histórico, em grande parte inferior à dez modificações por arquivo, exceto em dois momentos do histórico. Por outro lado, a variação de manutenabilidade apresentou um comportamento mais instável, em que apresentaram oscilações maiores, predominante negativas, ou seja, indicando um maior número de modificações que degradam o código-fonte.

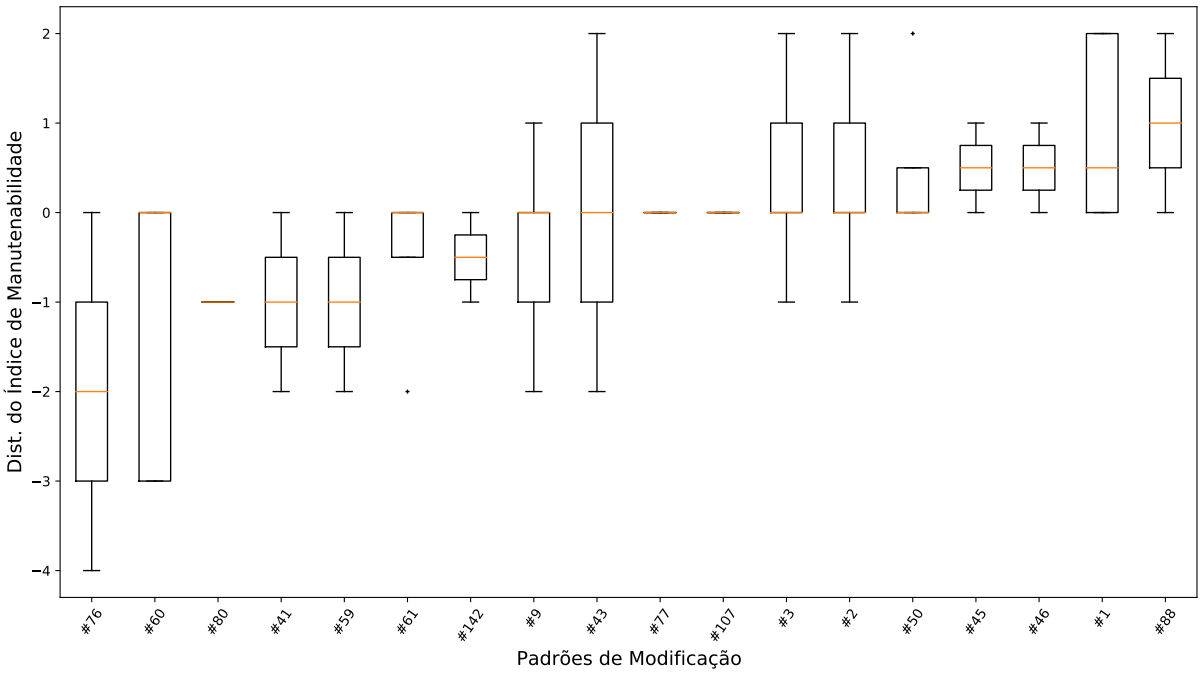
Em relação aos padrões sintáticos, foram identificados 18 padrões com pelo menos duas ocorrências. Na Figura 5.3 é ilustrado o comportamento quanto a variação da manutenabilidade das instâncias de cada padrão sintático com base nos arquivos de ocorrência.

Figura 5.2: Quantidade de mapeamentos e variação da manutenabilidade por Commit – Refactoring-Toy-Example.



Fonte: Elaborada pelo autor.

Figura 5.3: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – Refactoring Toy Example.

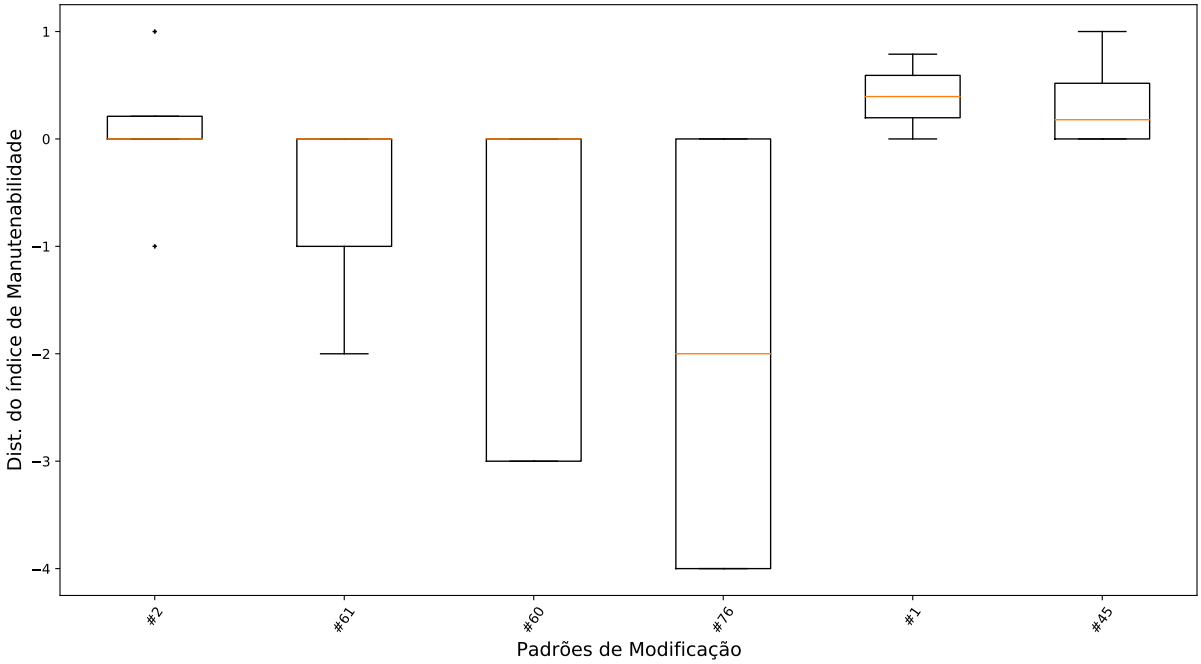


Fonte: Elaborada pelo autor.

A partir da execução do modelo de regressão, somente seis padrões sintáticos do conjunto inicial apresentaram impacto relevante sob a variação da manutenabilidade, conforme representado na Figura 5.4.

Pode-se observar que parte do conjunto de padrões apresentou um comportamento equilibrado entre melhorias e degradações, enquanto o restante registrou uma variação maior no

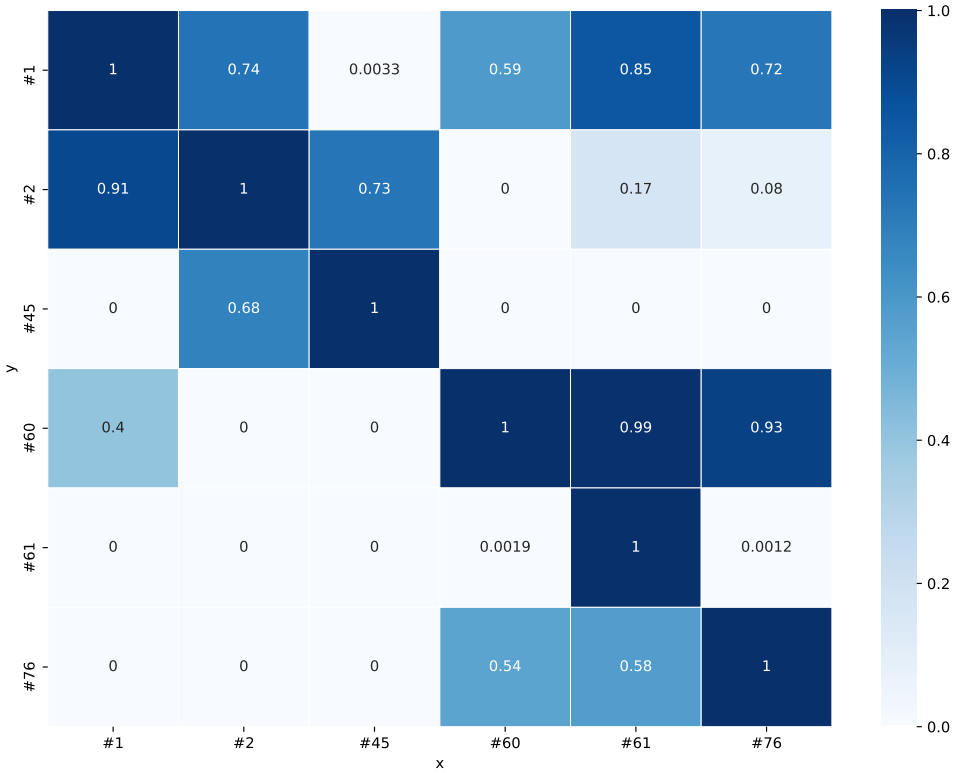
Figura 5.4: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos padrões) – Refactoring Toy Example.



Fonte: Elaborada pelo autor.

impacto. Contudo, cabe salientar que em ambas situações, todas as instâncias apresentaram a mediana de impacto próximo de zero, o que pode indicar uma baixa influência na maioria das ocorrências.

Figura 5.5: Relação entre os padrões sintáticos – Refactoring Toy Example.



Fonte: Elaborada pelo autor.

Outro aspecto consiste na relação entre as ocorrências dos padrões sintáticos. Nesse sentido, como representado na Figura 5.5, é possível identificar que os padrões $\#1$,$\#2$ e $\#45$; e também os padrões $\#60$,$\#61$ e $\#76$ apresentam uma alta correlação, indicando uma relação entre as modificações e/ou arquivos com ocorrências.

A seguir, alguns padrões sintáticos obtidos nesse repositório foram selecionados para um maior detalhamento. Adicionalmente, na Tabela 5.5 estão representadas as variações médias de cada métrica nas ocorrências de cada padrão sintático.

Tabela 5.5: Variação média das métricas em cada padrão sintático – Refactoring Toy Example.

Padrão Sintático	**DIT**	**WMC**	**CBO**	**LCOM**
#76	0,500	1,250	0,250	2,500
#1	0	-1,667	0	-0,917
#45	0	-0,250	-0,250	2,750

Fonte: Elaborada pelo autor.

- **Padrão #76**: a modificação consiste na criação de uma superclasse, a partir da extração de diversos métodos e atributos da classe. Adicionalmente, foi realizada a renomeação da classe-filha. Quanto ao impacto na manutenabilidade, apesar da remoção inicial dos métodos, foi identificado um aumento das métricas *DIT*, *WMC*, *CBO* e *LCOM*. Com o aumento das métricas, o padrão é classificado como degradação, contudo os métodos adicionados são provenientes de outras classes, nas quais as métricas foram reduzidas.

- **Padrão #1**: a modificação representada por esse padrão consiste em uma refatoração composta de *push-down method* e *push-down attribute*, em que os atributos e métodos são enviados para a classe-filha na hierarquia de herança. Quanto a manutenabilidade, o padrão sintático indicou uma melhoria do código-fonte, pelo redução das métricas *WMC* e *LCOM*.

  Na Figura 5.6 está representada a modificação do respectivo padrão sintático.

- **Padrão #45**: a modificação representada pelo padrão consiste na execução de uma refatoração *push-down method* de dois métodos. Paralelamente, no mesmo arquivo, foi conduzida a remoção de um método com várias chamadas a outros métodos, resultando em um impacto adicional na manutenabilidade. Quanto a manutenabilidade, pode-se observar uma melhoria devido a redução pequena nas métricas *WMC* e *CBO*, enquanto um aumento na métrica *LCOM*.

  Na Figura 5.7 está representada a modificação do respectivo padrão sintático.

5.3.2 Gson

O repositório *Gson* consiste em um projeto de códigoaberto, criado pela Google, para a implementação de uma biblioteca para serialização/desserialização de dados no formato *JSON* (*JavaScript Object Notation*). Este repositório foi escolhido devido a sua utilização em estudos relacionados, ao longo histórico de desenvolvimento (o projeto foi iniciado em 2008).

Figura 5.6: Exemplo de modificação representada pelo padrão sintático #1.

```
  public class AnimalAmarilho {

-   protected int age;
-   protected int name;
    ...
-   public int getName() {
-      return name;
-   }

-   public void setName(int name) {
-     this.name = name;
-   }
    ...
  }
```

```
  public class Reptile
    extends AnimalAmarilho {

+   protected int age;
+   protected int name;
    ...
+   public int getName() {
+      return name;
+   }

+   public void setName(int name) {
+     this.name = name;
+   }
    ...
}
```

Fonte: Elaborada pelo autor.

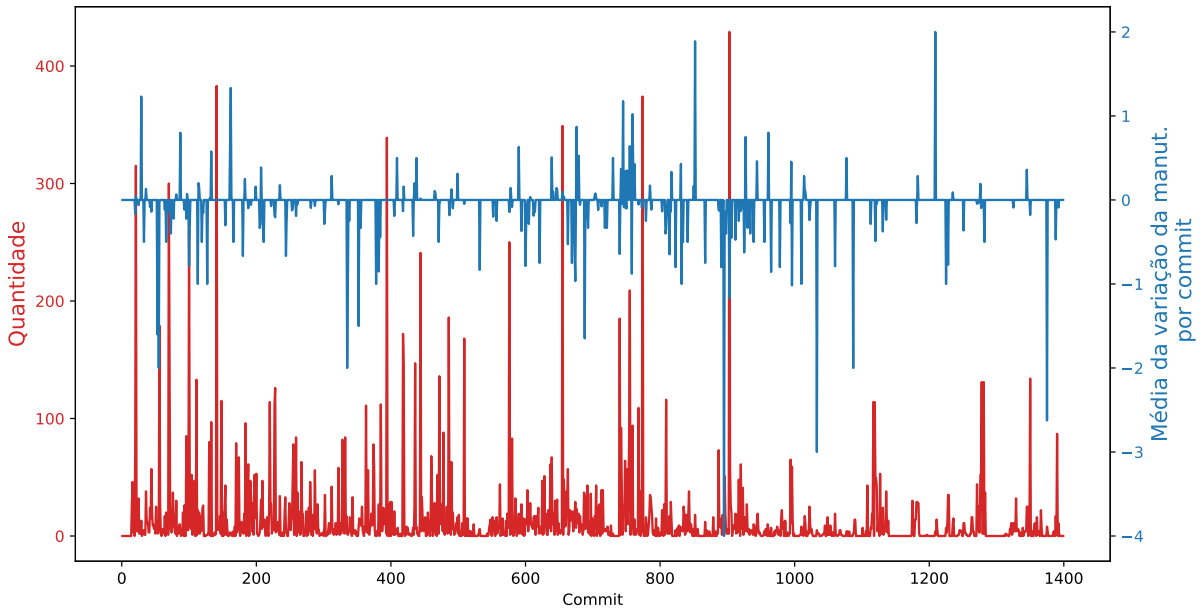
Figura 5.7: Exemplo de modificação representada pelo padrão sintático #45.

```
  public class Reptile {
    ...
-   public int getAge() {
-      return age;
-   }
-   public void setName(int name) {
-     this.name = name;
-   }
    ...
  }
```

```
  public class TurtleMarinha
    extends Reptile {
    ...
+   public int getAge() {
+      return age;
+   }
+   public void setName(int name) {
+     this.name = name;
+   }
    ...
  }
```

Fonte: Elaborada pelo autor.

Figura 5.8: Quantidade de mapeamentos e variação da manutenabilidade por Commit – Gson.



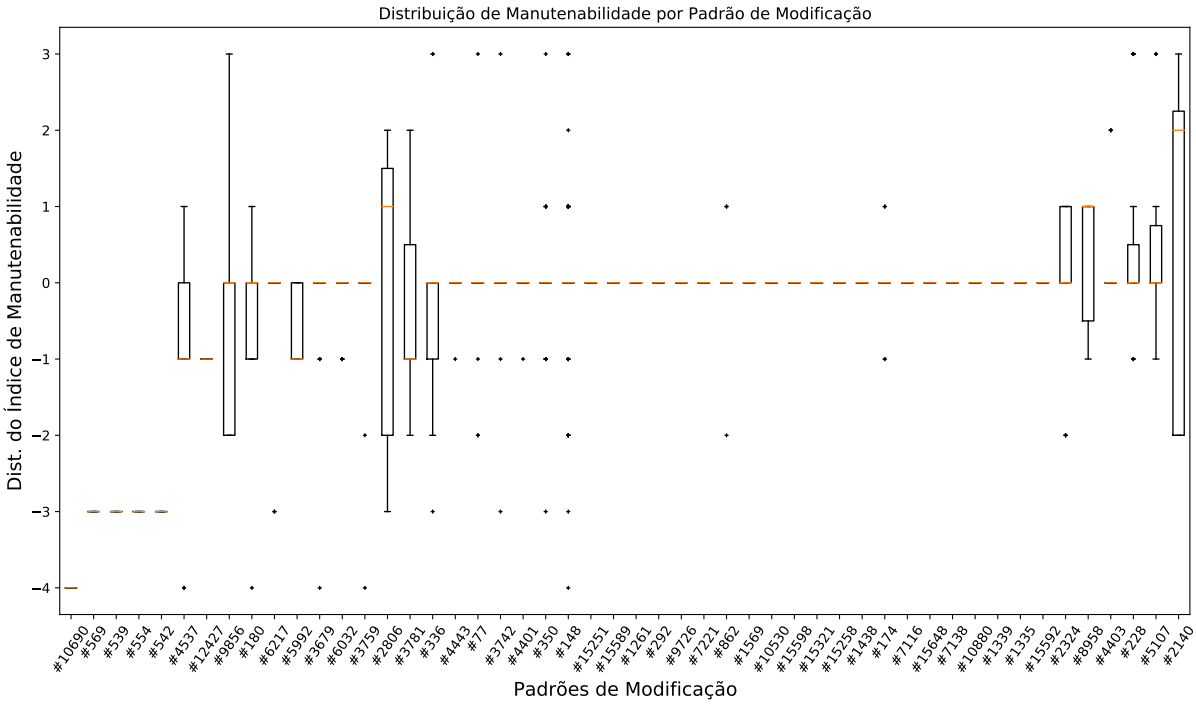
Fonte: Elaborada pelo autor.

O primeiro ponto de análise consiste no comportamento da manutenabilidade ao longo o histórico. Nesse sentido, o repositório apresentou uma alta taxa de modificações, principalmente até próximo o milésimo *commit*, com diversos picos de modificações, conforme representado na Figura 5.8.

Quanto a variação da manutenabilidade, o repositório apresentou oscilação ao longo dos *commits* com menor intensidade (número reduzido de melhorias e degradações fortes). Adicionalmente, pode-se observar a ocorrência de intervalos de *commits* com tendência maior para melhoria de código ou degradação. Por fim, ao final do histórico do repositório, pode-se observar um decréscimo na taxa de modificações e menor variação quanto a manutenabilidade do repositório.

Sobre os padrões sintáticos, o processo de agrupamento obteve um alto de número de instâncias, totalizando 686 padrões sintáticos. Na Figura 5.9 está representada uma amostra do conjunto total, sendo composta pelos padrões com maior número de ocorrências. O impacto na manutenabilidade descrito neste gráfico tem como base o comportamento médio dos arquivos de ocorrência.

Figura 5.9: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – Gson.



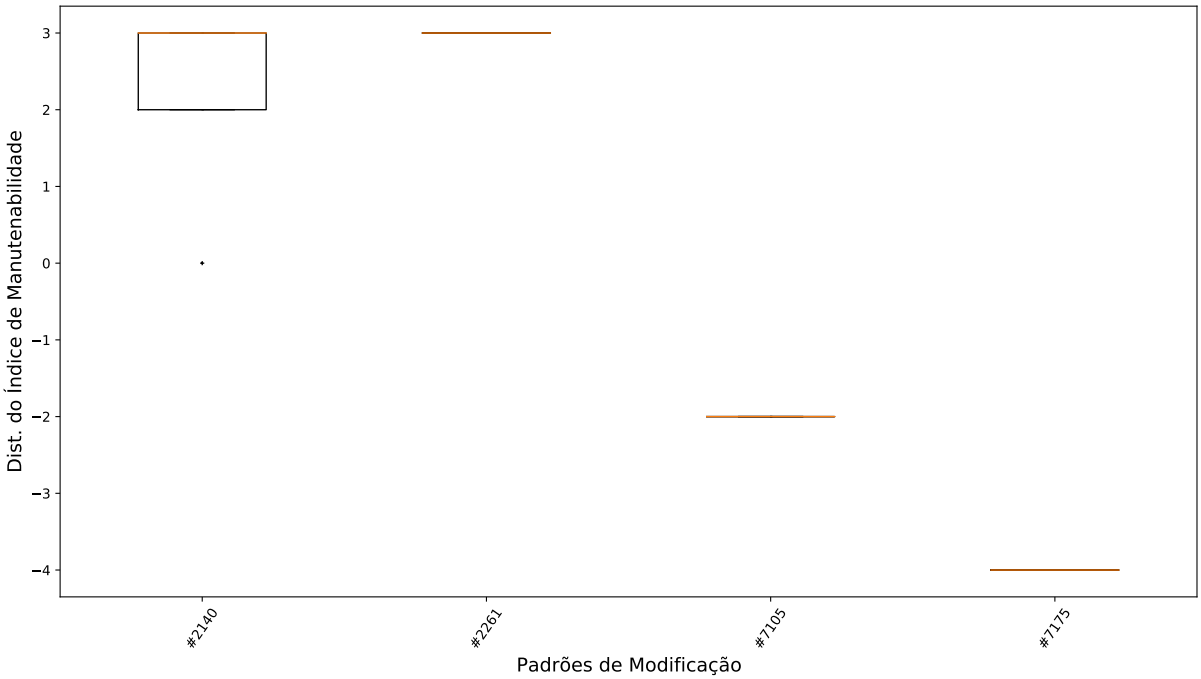
Fonte: Elaborada pelo autor.

A partir da amostra, pode-se observar uma relevante instabilidade no comportamento das ocorrências de cada padrão. Em alguns casos para o mesmo padrão sintático, o impacto na manutenabilidade indica uma melhoria ou degradação (amplo intervalo de variação). Em outros casos, o conjunto de ocorrências de um padrão sintático apresenta um resultado definido, contudo com a ocorrência de diversos pontos de exceção quanto ao impacto de valores incomuns.

Com a execução do modelo de regressão, um número reduzido de padrões foi identificado

com relevante, somente 4 instâncias (cerca 0,59% do conjunto inicial). Conforme representado na Figura 5.10, pode-se notar que os padrões mais significantes apresentam um comportamento bem definido quanto ao impacto sob a manutenabilidade, de modo que, as respectivas ocorrências registram pouca ou nenhuma variação entre si.

Figura 5.10: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos padrões) – Gson.



Fonte: Elaborada pelo autor.

A respeito da relação entre as ocorrências de cada padrão sintático, como representado na Figura 5.11, é possível notar a forte relação do padrão sintático $\#2140$ com os demais padrões, enquanto os demais são totalmente independentes entre si.

A seguir são detalhados alguns padrões sintáticos. Adicionalmente, na Tabela 5.6 estão representadas as variações médias de cada métrica nas ocorrências de cada padrão sintático.

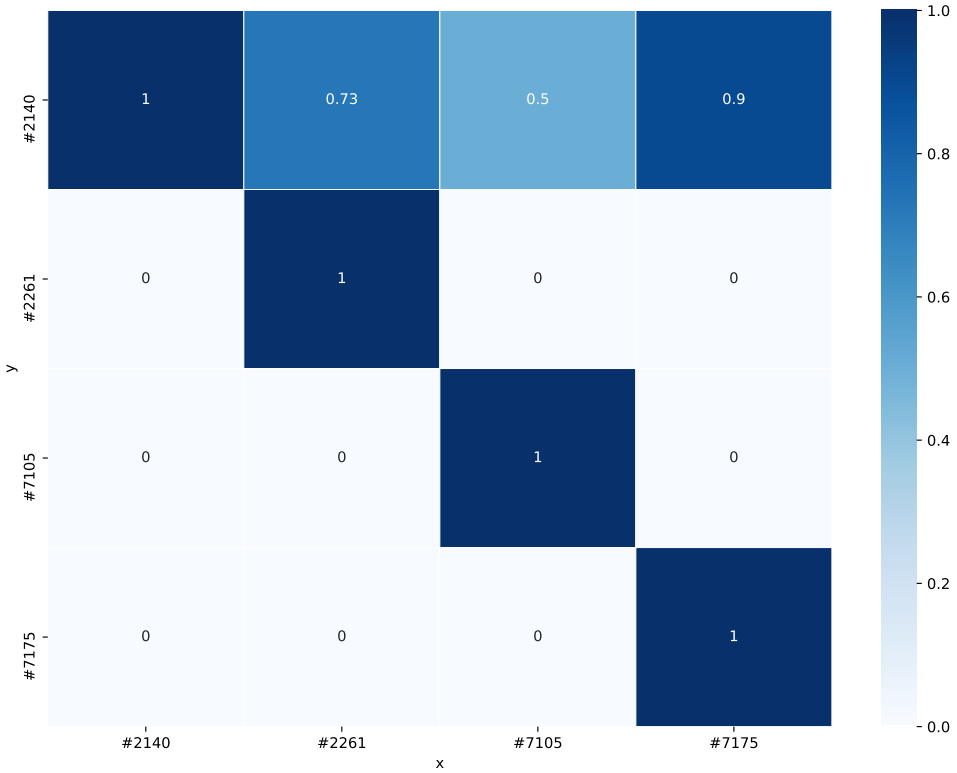
Tabela 5.6: Variação média das métricas em cada padrão sintático – Gson.

Padrão Sintático	DIT	WMC	CBO	LCOM
#2140	0	-0,875	-0,563	-19,625
#2261	0	0	1,5	-5,0
#7105	-1,0	3,667	2,667	27,334

Fonte: Elaborada pelo autor.

- **Padrão #2140**: a modificação realizou troca de uma atribuição de um tipo booleano por uma chamada a um método de uma classe externa. Essa classe externa foi extraída da classe da modificação original, resultando na redução do número de métodos e, consequentemente, melhoria na mensuração de qualidade da classe original.

Figura 5.11: Relação entre os padrões sintáticos – Gson.



Fonte: Elaborada pelo autor.

Figura 5.12: Exemplo de modificação representada pelo padrão sintático #2140.

```
  public GsonBuilder excludeFieldsWithoutExposeAnnotation() {
-   excludeFieldsWithoutExposeAnnotation = true;
+   excluder = excluder.excludeFieldsWithoutExposeAnnotation();
    return this;
  }
```

Fonte: Elaborada pelo autor.

- **Padrão #2261**: a modificação realizou a combinação de dois métodos com retorno de uma expressão com dupla condição, pela chamada de um método de uma classe externa. Contudo, a variação da mensuração de qualidade ocorre em decorrência da remoção de parte dos métodos e comandos presentes nos demais métodos.

- **Padrão #7105**: a modificação representada pelo padrão sintático consiste na substituição do tipo de dado do parâmetro do método. Quanto a manutenabilidade, foi registrado aumento significante das métricas *LCOM*, *WMC* e *CBO*, contudo esse resultado é proveniente de outras alterações nos mesmos arquivos.

5.3.3 Truth

O repositório *Truth* é um projeto de código-aberto, inicialmente desenvolvido pela Google, que consiste na implementação de uma biblioteca de asserções para testes automatizados, na linguagem de programação Java e com integrações adicionais com outras bibliotecas do mesmo desenvolvedor (ex.: *Guava*). A seleção do repositório ocorreu pela constante presença em

Figura 5.13: Exemplo de modificação representada pelo padrão sintático #2140.

```
- public boolean serializeField(Field f) {
-   return !serializationExclusionStrategy.shouldSkipClass(f.getType())
-       && !serializationExclusionStrategy.shouldSkipField(
-             new FieldAttributes(f)
-           );
- }

- private boolean deserializeField(Field f) {
-   return !deserializationExclusionStrategy.shouldSkipClass(f.getType())
-       && !deserializationExclusionStrategy.shouldSkipField(
-             new FieldAttributes(f)
-           );
- }

+ public boolean excludeField(Field f, boolean serialize) {
+   return !excluder.excludeClass(f.getType(), serialize)
+     && !excluder.excludeField(f, serialize);
+ }
```

Fonte: Elaborada pelo autor.

Figura 5.14: Exemplo de modificação representada pelo padrão sintático #7105.

```
  public static final Factory FACTORY = new Factory() {
    @SuppressWarnings("unchecked") // we use a runtime ...
-    public <T> TypeAdapter<T> create(MiniGson context,
      TypeToken<T> typeToken) {
+    public <T> TypeAdapter<T> create(Gson context,
      TypeToken<T> typeToken) {
      return typeToken.getRawType() == Date.class
        ? (TypeAdapter<T>) new DateTypeAdapter()
        : null;
    }
  };
```

Fonte: Elaborada pelo autor.

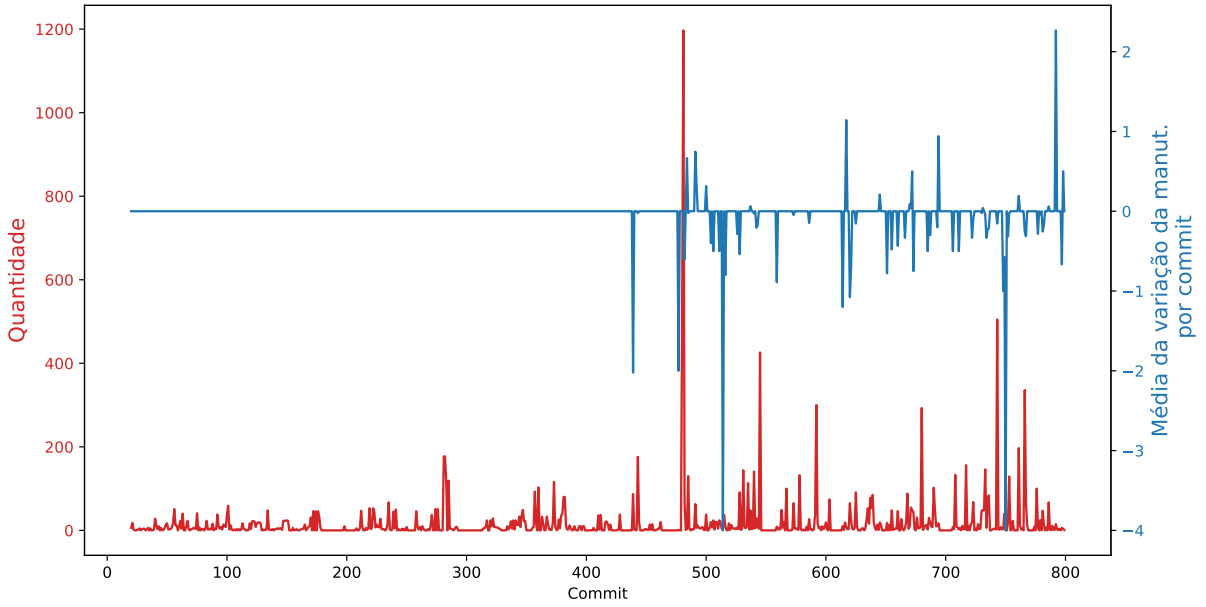
estudos relacionados, pelo longo histórico de desenvolvimento, além da análise quanto ao desenvolvimento por times semelhantes de programação em relação ao repositório *Gson*.

O primeiro aspecto de análise consiste no comportamento da manutenabilidade do repositório ao longo do histórico, como representado na Figura 5.15. A extração de métricas do repositório, entre os *commits* de número 0 a 450 aproximadamente, não apresentou registro de dados quanto às métricas devido a problemas de compatibilidade com a biblioteca utilizada. No restante do histórico, é possível notar momentos com intensas variações na manutenabilidade, os quais são espaçados por trechos de variações menores, tanto no sentido de melhorias quanto degradações.

Sobre o número de modificações por *commit*, é possível notar um número moderado durante a maior parte do histórico (inferior a 200 modificações), principalmente até o *commit* 500, em que foram registrados alguns picos de modificações (um caso superior a 1000 alterações).

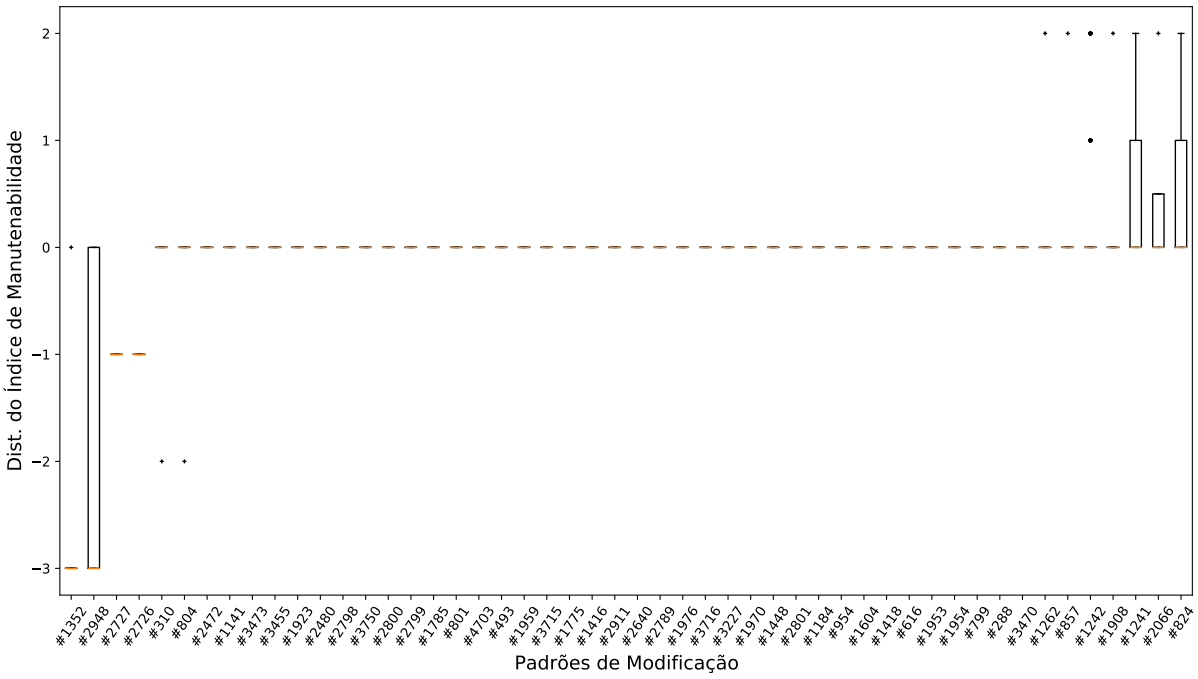
Quanto aos padrões sintáticos identificados, um conjunto inicial de 253 instâncias, pode-se notar que, conforme representado na Figura 5.16, as ocorrências da maioria dos padrões não

Figura 5.15: Quantidade de mapeamentos e variação da manutenabilidade por Commit – Truth.



Fonte: Elaborada pelo autor.

Figura 5.16: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – Truth.

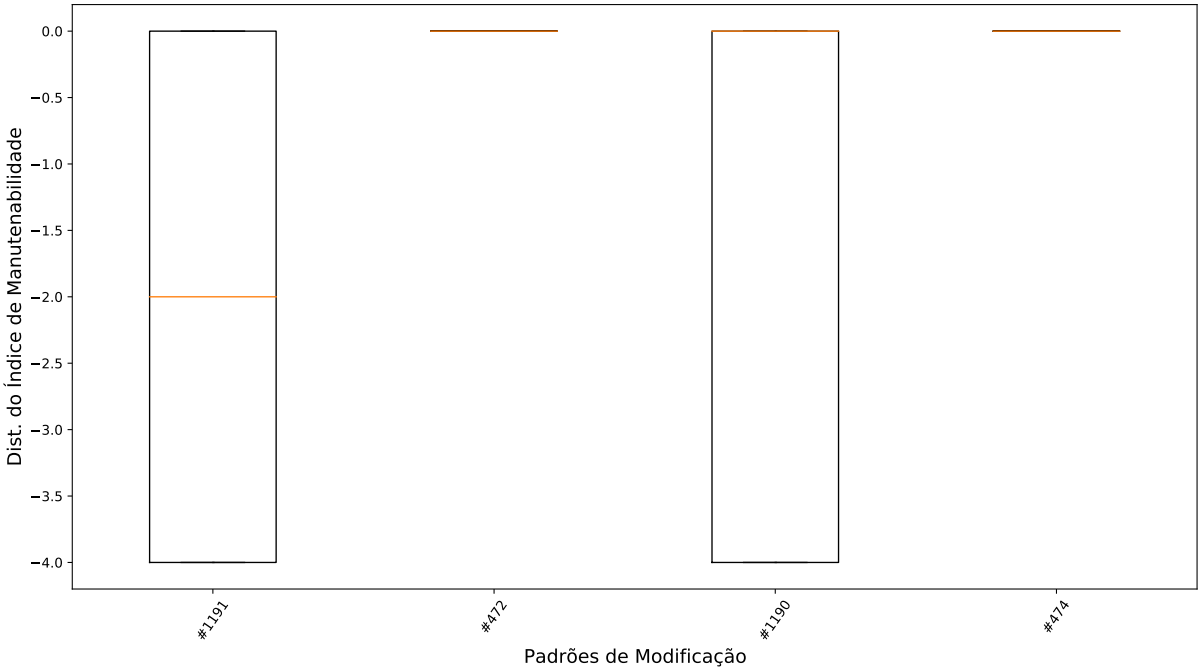


Fonte: Elaborada pelo autor.

apresentam relevante impacto nos arquivos, em que somente algumas exceções apresentam algum impacto, seja melhoria ou degradação, contudo ainda instável.

Considerando os impactos individuais de cada padrão, como apresentado na Figura 5.17, foram obtidos somente quatro padrões sintáticos significantes (cerca de 1,58%). A partir do gráfico, pode-se observar um comportamento instável e pouco significante dos padrões, em dois casos apresentam impacto na variação da manutenabilidade igual a zero, enquanto os demais

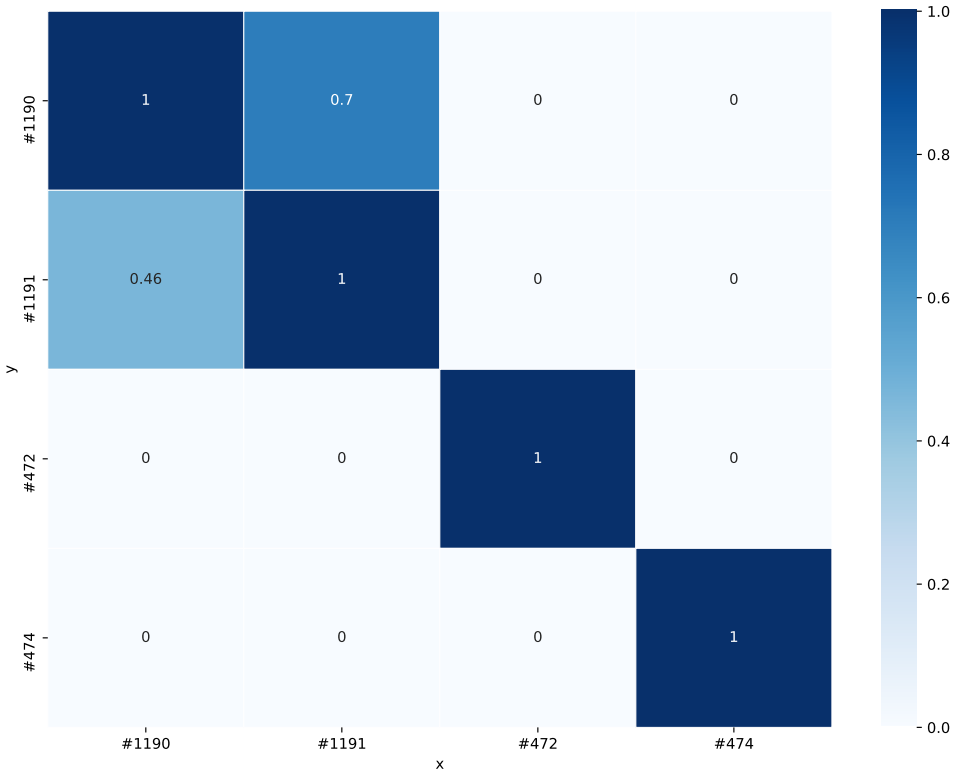
Figura 5.17: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos padrões) – Truth.



Fonte: Elaborada pelo autor.

apresentam tendência para degradação, contudo com impacto muito variante.

Figura 5.18: Relação entre os padrões sintáticos – Truth.



Fonte: Elaborada pelo autor.

Por fim, a respeito da relação entre os padrões sintáticos, conforme representado na Fi-

gura 5.18, pode-se identificar que os padrões #472 e #474 são totalmente independentes, enquanto os padrões #1190 e #1191 apresentam forte dependência.

Na sequência, alguns dos padrões sintáticos obtidos são apresentados de modo mais detalhado. Adicionalmente, na Tabela 5.7 estão representadas as variações médias de cada métrica nas ocorrências de cada padrão sintático.

Tabela 5.7: Variação média das métricas em cada padrão sintático – Truth.

Padrão Sintático	DIT	WMC	CBO	LCOM
#1191	0,667	3,000	1,334	2,000
#472	0	1,000	0	5,500
#1190	0,667	13,000	2,334	30,334
#474	0	1,000	0	5,000

Fonte: Elaborada pelo autor.

- **Padrões #1190 e #1191**: a modificação representada por ambos padrões consiste na substituição da superclasse da classe modificada. Quanto a manutenabilidade, ambos padrões foram classificados como degradações, porém com um grande variação em seus respectivos impactos. Esse comportamento decorre da forte variação das métricas em um arquivo de ocorrência (em ambos padrões), enquanto os demais arquivos mantiveram suas métricas inalteradas.

  Um exemplo de modificação dos padrões sintáticos #1190 e #1191 está apresentado na Figura 5.19.

Figura 5.19: Exemplo de modificação representada pelo padrão sintático #1190 e #1191.

```
- public class StringSubject extends Subject<StringSubject, String> {
+ public class StringSubject
    extends ComparableSubject<StringSubject, String> {

    public StringSubject(FailureStrategy failureStrategy, String string) {
      super(failureStrategy, string);
    }
    ...
```

Fonte: Elaborada pelo autor.

- **Padrão #472 e #474**: a modificação representada por ambos padrões consiste em um refatoração de renomeação de método (*rename method*), e consequentemente, o ajuste de todas as suas respectivas chamadas. Quanto a manutenabilidade, não foi registrada variação quanto às métricas consideradas em ambos os casos.

  Na Figura 5.20 está representada uma ocorrência de ambos padrões sintáticos.

5.3.4 JUnit4

O repositório *JUnit4* é um projeto de código-aberto, criado pelo *JUnit Team*, para implementação de um *framework* de testes unitários para a linguagem de programação Java. Esse repositório foi escolhido devido a recorrente presença em estudos relacionados e o longo período de desenvolvido, o qual permanecesse ativo, mesmo com uma versão mais recente do *framework*.

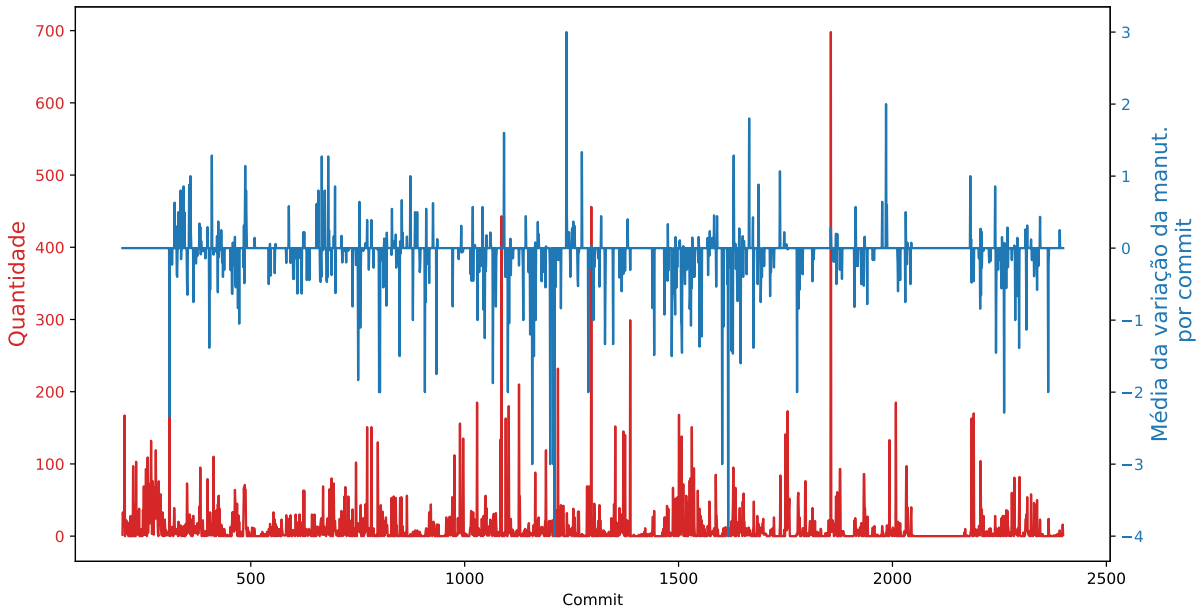
Figura 5.20: Exemplo de modificação representada pelos padrões sintáticos #472 e #474.

```
- public void comparesEqualTo(T other) {
+ public void isEquivalentAccordingToCompareTo(T other) {
    if (getSubject().compareTo(other) != 0) {
      failWithRawMessage(
        "%s should have been equivalent to <%s>", getDisplaySubject(),
        other
      );
      ...
    }
  }
```

Fonte: Elaborada pelo autor.

Inicialmente, quanto a variação da manutenabilidade ao longo do histórico do repositório, como representado na Figura 5.24 pode-se observar uma oscilação moderada entre melhoria ou degradação no comportamento médio de cada *commit*. Também cabe destacar que, diferentemente dos repositórios anteriores, o repositório apresenta intervalos de *commits* com comportamento mais uniforme, em ambos os sentidos.

Figura 5.21: Quantidade de mapeamentos e variação da manutenabilidade por Commit – JUnit4.



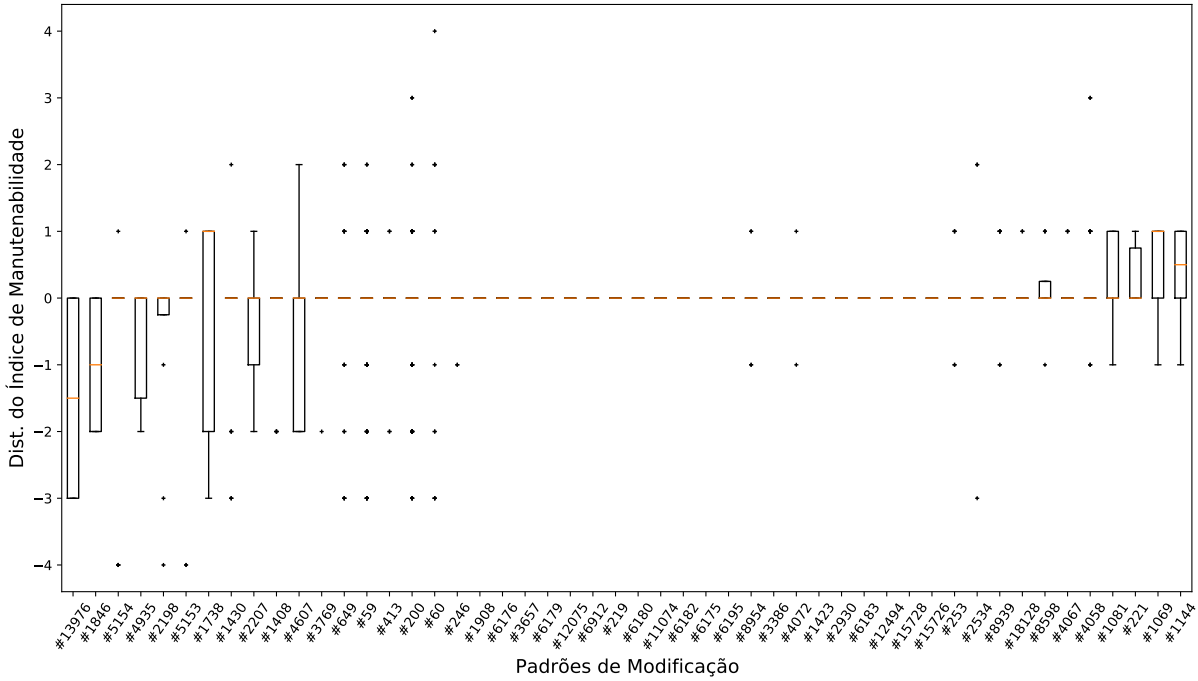
Fonte: Elaborada pelo autor.

Adicionalmente, quanto a taxa de modificações, pode-se notar no histórico a presença de períodos bem-definidos de aumento e diminuição na taxa de modificações, juntamente com um número reduzido de *commits* com alta taxa de modo isolado.

A respeito dos padrões sintáticos, o processo de agrupamento identificou um total de 708 instâncias. A partir desse conjunto inicial, os principais padrões (com base no número de ocorrências) foram selecionados e estão apresentados na Figura 5.22. Por meio da amostra, pode-se observar que as ocorrências dos principais padrões apresentam um comportamento com grande variação quanto a manutenabilidade, de modo que, a maioria das instâncias apresenta impacto

mediano igual a zero, e também significante variação de impacto, seja por um intervalo extenso ou pelo de valores incomuns pela análise do gráfico de *box-plot*.

Figura 5.22: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – JUnit4.



Fonte: Elaborada pelo autor.

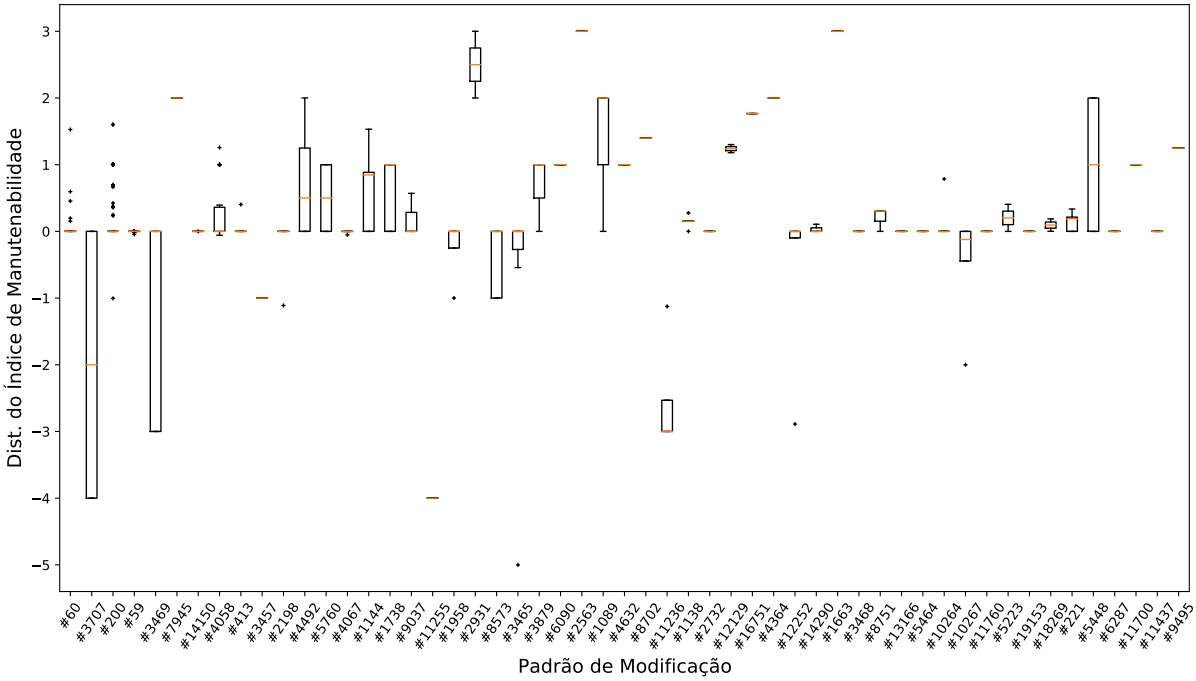
Com a execução do modelo de regressão, um conjunto de 44 padrões sintáticos relevantes foi identificado, aproximadamente 6,21% do conjunto inicial. Na Figura 5.23 é ilustrado o comportamento das ocorrências de cada um dos padrões sintáticos do conjunto resultante. Pode-se notar que o conjunto resultante obteve um comportamento mais bem definido, em detrimento do conjunto anterior; tanto na variação de impacto quanto de valores atípicos. Em alguns casos, como nos padrões #11255, #2931 e #1663, por exemplo, pouca ou nenhuma variação do impacto sob a manutenabilidade foi registrada. Apesar de um conjunto com padrões mais estáveis, ainda sim um número relevante de padrões apresentou a mediana de impacto próxima ou igual a zero.

Por fim, a respeito do relacionamento entre os padrões sintáticos, pode-se observar que, conforme representado na Figura 5.24, a maioria do conjunto apresentou baixa dependência entre si, em que menos de 10 instâncias (aprox. 22%) apresentaram ao menos um registro de correlação com outro padrão sintático superior a 0,5.

Na sequência, alguns padrões sintáticos são descritos mais detalhadamente. Para cada um padrões descritos a seguir, na Tabela 5.8 estão representadas as variações médias de cada métricas nas ocorrências de cada padrão sintático.

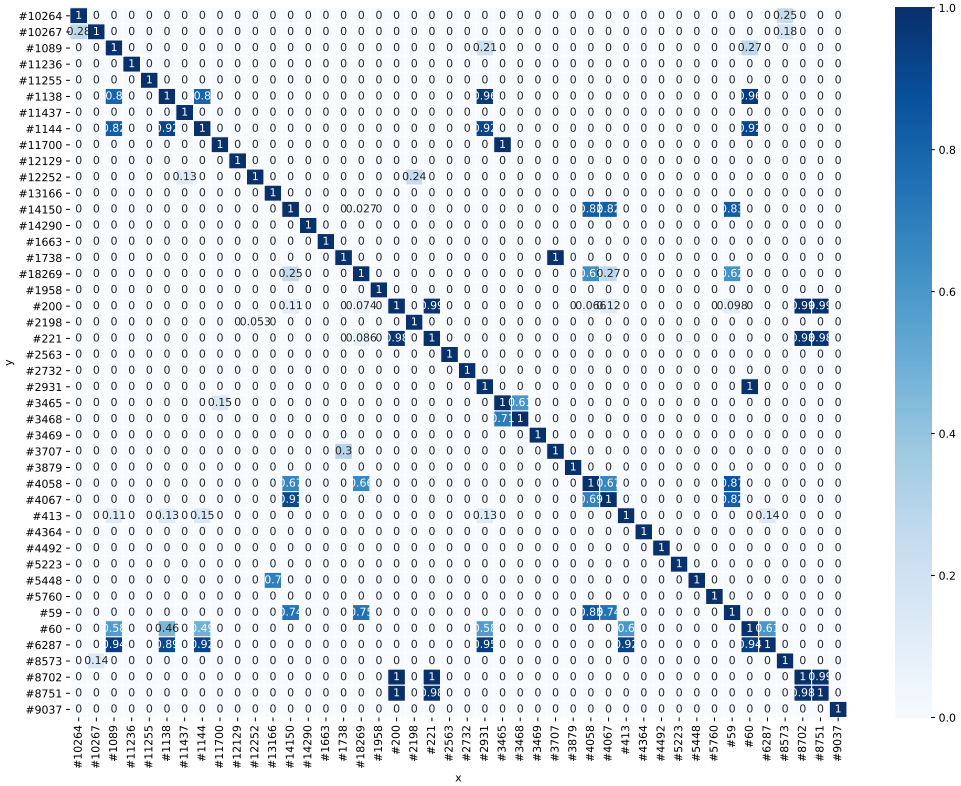
- **Padrão #2931**: a modificação representada por esse padrão consiste na renomeação da referência de importação de uma classe em decorrência de uma refatoração de *Move Class*. A variação da manutenabilidade identificada aconteceu em decorrência da não-identificação

Figura 5.23: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos padrões) – JUnit4.



Fonte: Elaborada pelo autor.

Figura 5.24: Relação entre os padrões sintáticos – JUnit4.



Fonte: Elaborada pelo autor.

de correspondência dos arquivos entre os diferentes pacotes, resultando em um "falso incremento"nas métricas de manutenabilidade.

Tabela 5.8: Variação média das métricas em cada padrão sintático – JUnit4.

Padrão Sintático	DIT	WMC	CBO	LCOM
#12129	-0,500	-7,000	-2,000	-69,500
#1663	0	-2,000	-2,000	-18,000
#11700	0	-5,000	3,500	-88,000

Fonte: Elaborada pelo autor.

- **Padrão #12129**: a modificação consiste na substituição da chamada do método construtor da classe. A troca da chamada foi realizada pela alteração do tipo do parâmetro. Quanto a manutenabilidade, uma redução de todas métricas foi identificada nas ocorrências do padrão sintático, contudo a variação foi resultante de modificações adicionais nos mesmos arquivos.

Figura 5.25: Exemplo de modificação representada pelo padrão sintático #12129.

```
- public static MaxCore forFolder(String storedResults) {
+ public static MaxCore forFolder(File storedResults) {
    return new MaxCore(storedResults);
  }
```

Fonte: Elaborada pelo autor.

- **Padrão #1663**: a modificação representada por esse padrão consiste na alteração da assinatura do método, em que o nome do método e o parâmetros são substituídos. Juntamente com a interface, alguns comandos do método são alterados de modo a manipular o novo tipo de dado passado via parâmetro (antes da modificação, o dado era extraído de modo interno ao método). Quanto ao impacto na manutenabilidade, a modificação foi identificada como melhoria, em que todas as métricas analisadas apresentaram forte redução, porém essa variação é resultante da remoção de comandos e métodos dos arquivos de ocorrência.

Figura 5.26: Exemplo de modificação representada pelo padrão sintático #1163.

```
- public List<Test> asTestList(Plan plan) {
-       Description description= plan.getDescription();
-       if (description.isTest())
-               return Arrays.asList(asTestCase(description));

+ public Test asTest(Description description) {
+       if (description.isSuite())
+      return createTest(description);
    else {
```

Fonte: Elaborada pelo autor.

- **Padrão #11700**: a modificação representada consiste na atualização da chamada de um método que foi movido entre classes (refatoração *move-method*). Em termos manutenabilidade, a modificação foi indicada como um pequena melhoria, em que as métricas

WMC e *LCOM* apresentaram redução, enquanto a métrica *CBO* aumentou. Pela análise das ocorrências, pode-se observar que a variação em parte foi resultado da alteração identificada, contudo a variação teve contribuição de modificações adicionais realizadas.

Figura 5.27: Exemplo de modificação representada pelo padrão sintático #11700.

```
  private void stopping() {
-   endNanos = currentNanoTime();
+   endNanos = clock.nanoTime();
  }
```

Fonte: Elaborada pelo autor.

- **Padrão #2563**: a modificação representada consiste na substituição da chamada de um método, cujo parâmetro provém de outra chamada de método, pela encadeamento de métodos em uma operação de atribuição. Quanto a manutenabilidade, a modificação foi classificada como uma melhoria forte, contudo esse resultado é proveniente da remoção de alguns métodos complexos das classes de ocorrência.

Figura 5.28: Exemplo de modificação representada pelo padrão sintático #2563.

```
  Arrays.sort(methods, new Comparator<Method>() {
    @Override public int compare(Method m1, Method m2) {
-     int i1 = _names.indexOf(nameAndDescriptor(m1));
+     int i2 = _names.indexOf(nameAndDescriptor(m2));
-     int i1 = m1.getName().hashCode();
+     int i2 = m2.getName().hashCode();
      return i1 != i2 ? i1 - i2 : m1.toString().compareTo(m2.toString());
    }
  });
```

Fonte: Elaborada pelo autor.

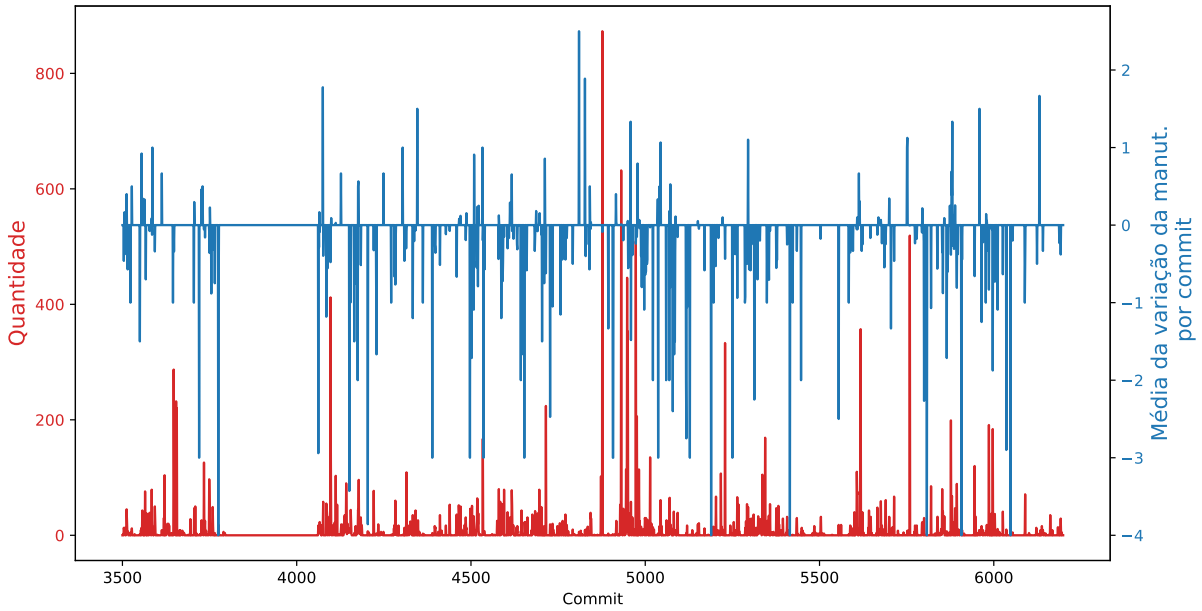
5.3.5 JUnit5

O repositório *JUnit5*, também desenvolvido pelo *JUnit Team*, consiste no projeto mais recente do *framework JUnit*. Esse repositório foi escolhido devido a possibilidade de comparação com entre repositórios com a mesma equipe de desenvolvimento, e também pelo elevado número de *commits* em um período reduzido de tempo, em detrimento dos demais repositórios.

O primeiro aspecto de análise quanto ao evolução da manutenabilidade ao longo do histórico. Na Figura 5.29 é ilustrado o histórico de versão entre os *commits* 3500 a 6200. De modo geral, é possível notar que ao longo de todo o período analisado, o repositório apresentou significantes oscilações quanto ao impacto médio sob a manutenabilidade, principalmente no sentido de modificações que promovam a degradação do código-fonte.

Sobre a taxa de modificação por *commit* no período, é possível notar um número significativamente baixo, na grande maioria dos casos inferior a 100, alternado por algumas altas fortes, porém menos frequentes. Nesse sentido, dado o número baixa de modificações e a forte oscilação do impacto sob a manutenabilidade, pode ser um indicativo que as modificações conduzidas têm alto impacto sob a estrutura do código-fonte.

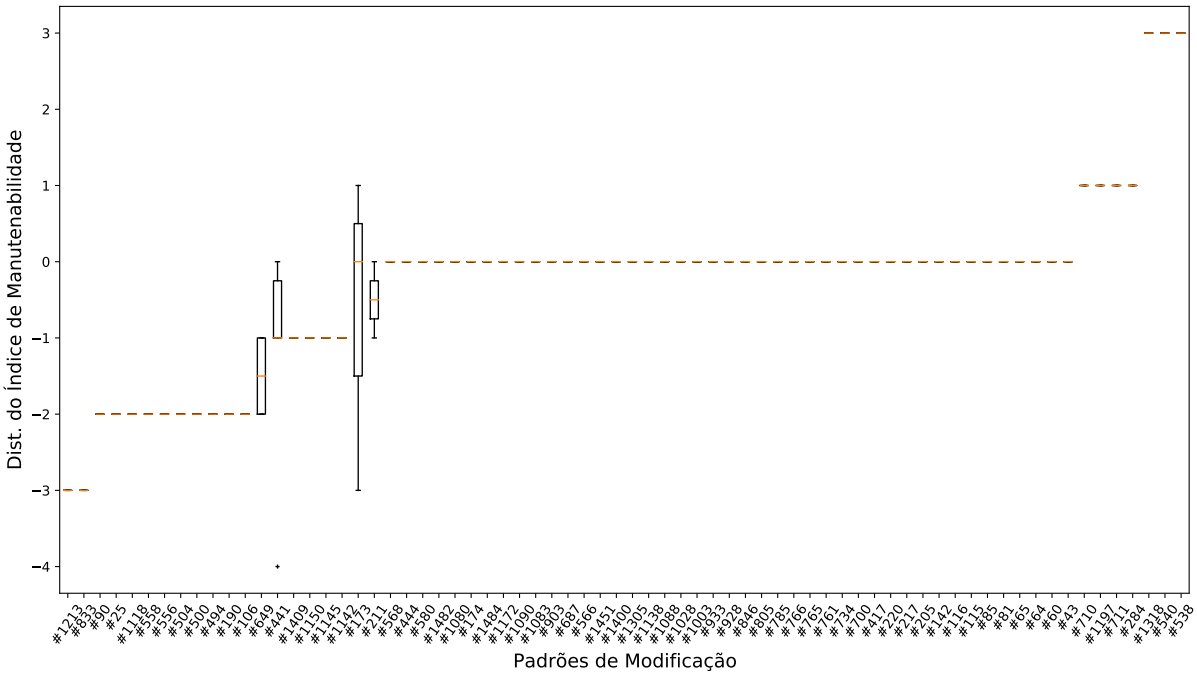
Figura 5.29: Quantidade de mapeamentos e variação da manutenabilidade por Commit – JUnit5.



Fonte: Elaborada pelo autor.

Quanto aos padrões sintáticos, o processo de agrupamento identificou um conjunto de 227 padrões sintáticos, cujas principais instâncias estão representadas na Figura 5.30.

Figura 5.30: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – JUnit5.



Fonte: Elaborada pelo autor.

A partir da amostra representada, pode-se notar que as ocorrências dos padrões apresentam considerável estabilidade quanto ao impacto médio sob a manutenabilidade nos arquivos de ocorrência. De modo geral, grande parte das instâncias obtidas apresentaram o impacto sob a

manutenabilidade sem nenhuma oscilação em termos de valor absoluto.

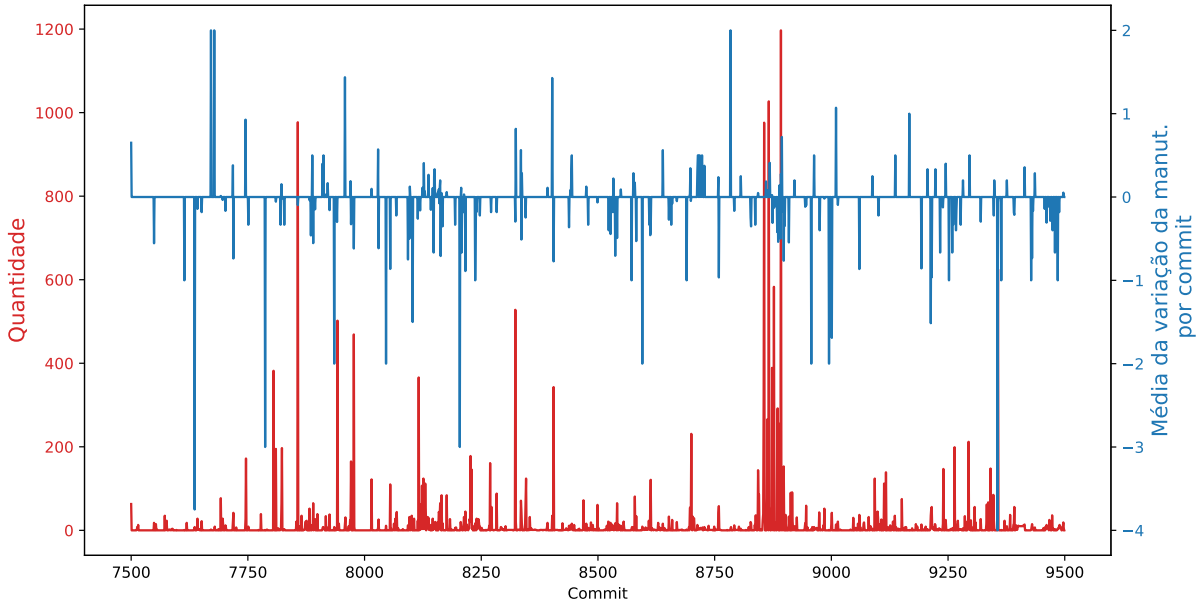
Após a execução do modelo de regressão, ao contrário da estabilidade indicada pelo conjunto inicial, nenhuma das instâncias obtidas obteve impacto individual significativo quanto a manutenabilidade, ou seja, todas tiveram peso médio sob os respectivos igual a zero.

5.3.6 CheckStyle

O repositório *CheckStyle* é um projeto de código-aberto, que consiste na implementação de uma ferramenta de verificação de regras e estilos de programação, de acordo com um conjunto de convenções e melhores práticas. Esse repositório foi selecionado pela presença em experimentos e análises de estudos relacionados e também devido a variação quanto a equipe de desenvolvimento. A análise do repositório contemplou os *commits* de 7500 a 9500 do histórico de versão.

Inicialmente, sobre a manutenabilidade ao longo do histórico, conforme representado na Figure 5.31, o repositório registrou uma grande variabilidade, tanto na alternância de períodos de tendência a melhoria ou degradação do código-fonte, e também quanto a variação de intensidade, em que *commits* próximos temporalmente registraram comportamentos opostos, como exemplificado no trecho entre o *commit* 7500 e 7750.

Figura 5.31: Quantidade de mapeamentos e variação da manutenabilidade por Commit – CheckStyle.



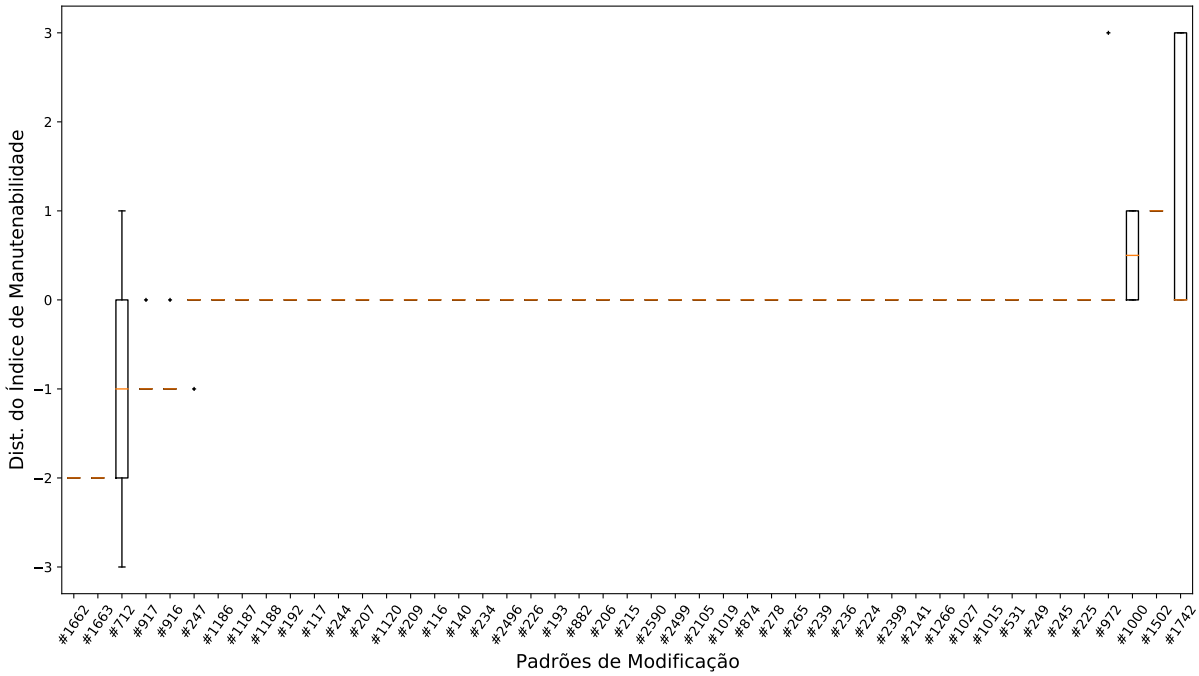
Fonte: Elaborada pelo autor.

Quanto a taxa de modificações por *commit*, o comportamento do repositório foi também similar ao da manutenabilidade, de modo que uma grande variação foi registrada com diversos altas fortes de modo isolado no período *commit* 8500. Além disso, entre os *commits* 8750 e 9000, o histórico registro uma sequência relevante de *commits* com alta taxa de modificação (superior a 200 modificações).

Em relação aos padrões sintáticos, o processo de agrupamento identificou um total 372

instâncias, das quais uma amostra com os padrões mais recorrentes está representada na Figura 5.33. A partir do gráfico, pode-se observar que a grande maioria das ocorrências apresentou impacto na manutenabilidade igual a zero, enquanto os demais apresentaram variações para melhorias ou degradações de código-fonte.

Figura 5.32: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – CheckStyle.



Fonte: Elaborada pelo autor.

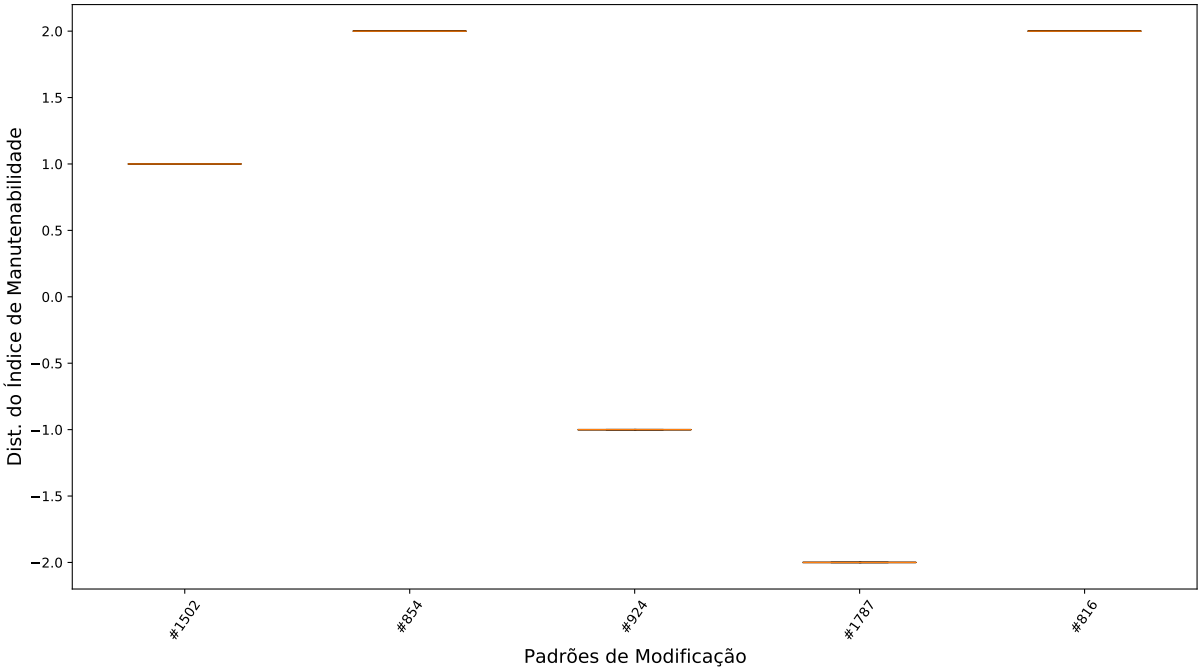
A partir da execução do modelo de regressão, foram obtidos cinco padrões sintáticos relevantes, aproximadamente 1,3% do conjunto inicial, como representado na Figura 5.33. De modo geral, as ocorrências dos padrões sintáticos apresentaram um comportamento estável (sem oscilação no valor de impacto) e bem definido quanto a sua classificação, em que os padrões #1502, #854 e #816 foram identificados como melhorias, enquanto #924 e #1787 como degradações.

Por fim, quanto a relação entre os padrões sintáticos, conforme representado na Figura 5.34, pode-se observar que cada uma das instâncias é totalmente independente, sem qualquer correlação com as demais.

A seguir, as modificações referentes aos seguintes padrões sintáticos são descritos detalhadamente. De modo adicional, na Tabela 5.9 estão representadas as variações médias das métricas analisadas nas ocorrências de cada padrão sintático.

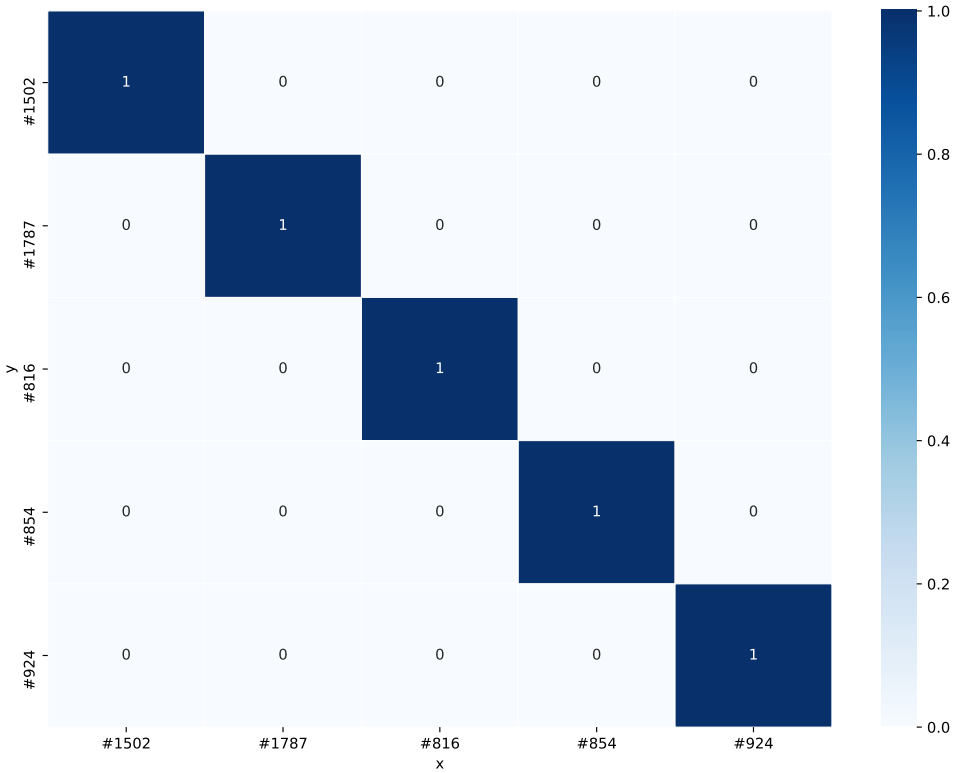
- **Padrão #1502**: a modificação representada por esse padrão consiste na remoção do tratamento de exceção (*try/catch*) de uma atribuição do retorno de um método. Essa modificação resulta em uma redução no número de fluxos de execução do método, o que resulta em uma pequena redução na métrica *WMC* apenas, enquanto as outras permaneceram inalteradas.

Figura 5.33: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos padrões) – CheckStyle.



Fonte: Elaborada pelo autor.

Figura 5.34: Relação entre os padrões sintáticos – CheckStyle.



Fonte: Elaborada pelo autor.

- **Padrão #854**: a modificação representada por esse padrão consiste em uma refatoração *Move Method*, juntamente com as atualizações das referências a esse, após a movimentação da classe original para uma classe utilitária. Quanto a manutenabilidade, a modifica-

Tabela 5.9: Variação média das métricas em cada padrão sintático – CheckStyle.

Padrão Sintático	DIT	WMC	CBO	LCOM
#1502	0	-1,000	0	0
#854	0	-1,000	0	-19,000
#816	0	-3,000	0	-13,000
#924	0	0	2,000	0
#1787	0	6,000	0	63,000

Fonte: Elaborada pelo autor.

Figura 5.35: Exemplo de modificação representada pelo padrão sintático #1502.

```
  public void setOption(String optionStr) {
-   try {
-     option = PadOption
-       .valueOf(optionStr.trim().toUpperCase(Locale.ENGLISH));
-   }
-   catch (IllegalArgumentException iae) {
-     throw new IllegalArgumentException(
-       "unable to parse " + optionStr, iae
-     );
-   }
+   option = PadOption
+      .valueOf(optionStr.trim().toUpperCase(Locale.ENGLISH));
  }
```

Fonte: Elaborada pelo autor.

ção foi classificada como um melhoria, em que, pode-se observar uma redução uniforme das métricas *WMC* e *LCOM* em decorrência da movimentação do método para outra classe.

Figura 5.36: Exemplo de modificação representada pelo padrão sintático #854.

```
  ...
  boolean result = false;
  if (literalDo.getParent().getType() == TokenTypes.SLIST) {
      final DetailAST block = literalDo.getFirstChild();
-     result = isOnSameLine(block, literalDo);
+     result = TokenUtil.areOnSameLine(block, literalDo);
  }
  ...
```

Fonte: Elaborada pelo autor.

- **Padrão #816**: a modificação representada pelo padrão consiste na substituição da chamada de um método específico por outro de mesma interface, após a renomeação desse no arquivo de declaração. Quanto a manutenabilidade, de modo geral, foi registrada redução nos valores das métricas *WMC* e *LCOM*, indicando a presente modificação como melhoria. Contudo essa diferença provém de alterações adicionais realizadas no mesmo arquivo, em que um método foi removido juntamente com comandos de outro método, resultando na diminuição da complexidade de lógica implementada.

Figura 5.37: Exemplo de modificação representada pelo padrão sintático #816.

```
switch (ast.getType()) {
  case TokenTypes.CTOR_DEF:
  case TokenTypes.METHOD_DEF:
-   startToken = skipAnnotationOnlyLines(ast);
+   startToken = skipModifierAnnotations(ast);
    brace = ast.findFirstToken(TokenTypes.SLIST);
    break;
  ...
```

Fonte: Elaborada pelo autor.

- **Padrão #924**: a modificação representada por esse padrão consiste na renomeação das referências de uma classe que passou pela refatoração de *Rename Class*. Quanto ao impacto na manutenabilidade, a substituição de classes resultou em um aumento de dependência entre classes, resultando no aumento da métrica *CBO*. Apesar da nova classe estar relacionada com o aumento do acoplamento, a variação ocorreu em decorrência das novas chamadas a métodos, tornando a classe atual mais dependente da nova implementação.

Figura 5.38: Exemplo de modificação representada pelo padrão sintático #924.

```
  parser.setFilename(contents.getFileName());
- parser.setASTNodeClass(DetailAST.class.getName());
+ parser.setASTNodeClass(DetailAstImpl.class.getName());
  ...
```

Fonte: Elaborada pelo autor.

- **Padrão #1787**: a modificação representada consiste na substituição da chamada de um método por outro também já existente, somente pela renomeação da chamada. A modificação é classificada como degradação, porém o impacto mais relevante sob a manutenabilidade é resultante da adição de outros métodos e instruções na mesma classe, o que pode ser observado pelo aumento significativo das métricas *WMC* e *LCOM*.

Figura 5.39: Exemplo de modificação representada pelo padrão sintático #1787.

```
  private void checkIdent(DetailAST type) {
-   final FullIdent ident = FullIdent.createFullIdentBelow(type);
+   final FullIdent ident = FullIdent.createFullIdent(type);
    if (isMatchingClassName(ident.getText())) {
      log(ident.getDetailAst(), MSG_KEY, ident.getText());
    }
  }
```

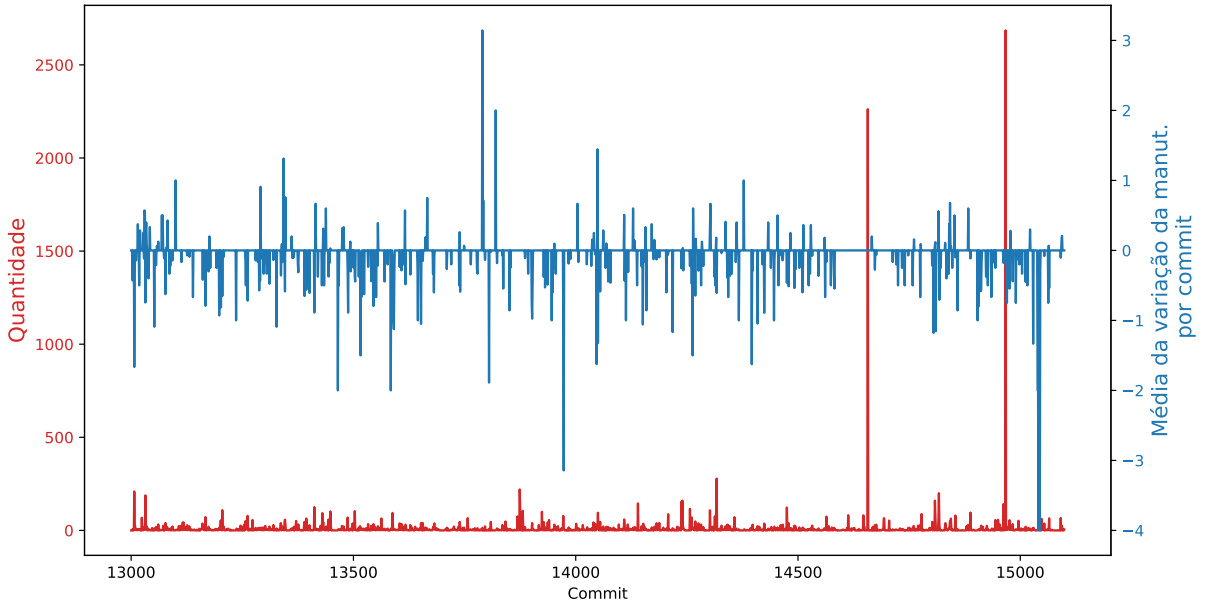
Fonte: Elaborada pelo autor.

5.3.7 FindBugs

O repositório *FindBugs* é um projeto de código-aberto, desenvolvido e licenciado pela *University of Maryland*, que consiste em uma ferramenta de análise estática para a identificação de defeitos no código-fonte de projetos desenvolvidos na linguagem de programação Java. A

seleção do repositório foi realizada em decorrência de sua presença em estudos relacionados presentes na literatura, assim como, pela análise do um software proveniente do âmbito acadêmico. Um fator importante a se destacar consiste que a análise desse repositório considerou as modificações realizada entre os *commits* 13 000 a 15 000, aproximadamente.

Figura 5.40: Quantidade de mapeamentos e variação da manutenabilidade por Commit – FindBugs.



Fonte: Elaborada pelo autor.

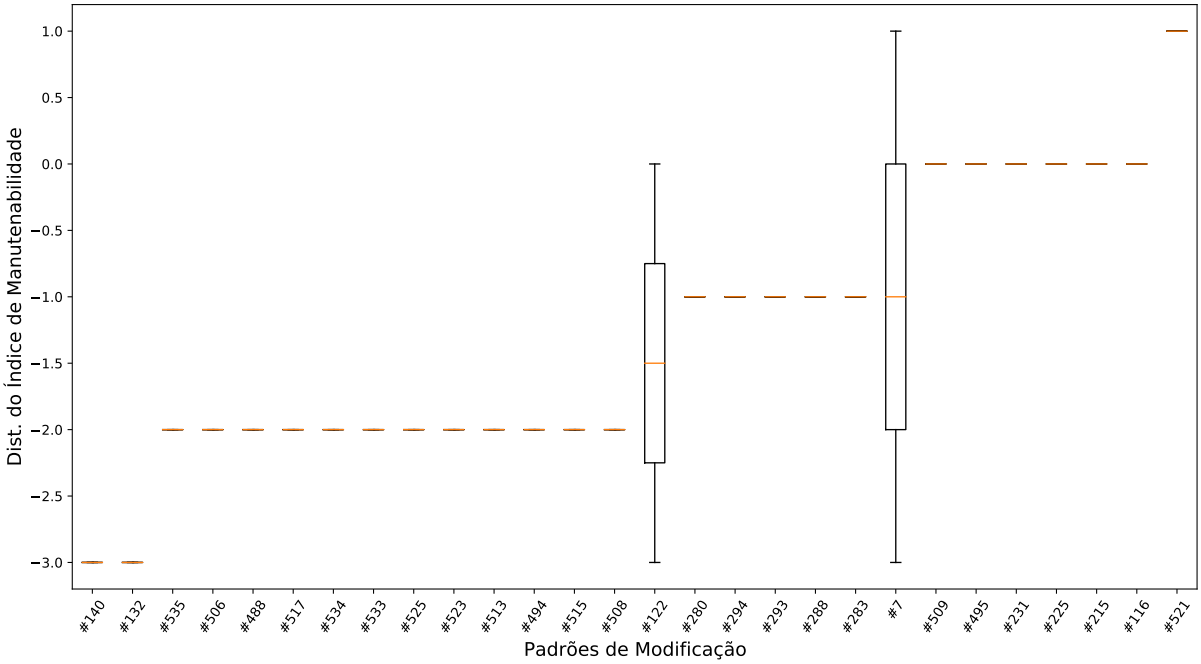
Inicialmente, sobre a variação da manutenabilidade ao longo do período, conforme representado na Figura 5.40, o repositório apresentou um comportamento estável, em que poucos *commits* registraram oscilações acentuadas no sentido de melhorias ou degradações. A única exceção ocorreu ao final do período analisado, em que um conjunto de *commits* registrou consecutivamente modificações com alta tendência de degradação.

Quanto a taxa de modificações por *commit*, o repositório apresentou um número relativamente estável ao longo do período, em que foram registradas, em média, menos que 250 modificações em cada *commit*. Esse comportamento somente não foi registrado em dois instantes entre os *commits* 14 500 e 15 000.

Em relação aos padrões sintáticos, o processo de agrupamento identificou 99 padrões com pelo menos duas ocorrências. Na Figura 5.41 está representado uma amostra contendo as instâncias mais relevantes. Nesse sentido, pode-se observar que, baseado no comportamento médio dos arquivos das ocorrências, a maioria dos padrões apresentou um comportamento estável quanto ao impacto na manutenabilidade, exceto por duas instâncias que registraram maior oscilação. Além disso, pode-se notar uma razoável distribuição dos padrões em diferentes níveis de impacto, em que são registrados padrões com classificação de degradação baixa e forte, e melhorias fracas.

Nesse repositório, com a execução do modelo de regressão, nenhum dos padrões sintáticos identificados no conjunto inicial apresentou impacto relevante em seus respectivos arquivos de

Figura 5.41: Variação da manutenabilidade das ocorrências de cada padrão sintático (Manuten. dos arquivos) – FindBugs.



Fonte: Elaborada pelo autor.

ocorrência.

5.4 Discussão

Nesta seção são analisados alguns pontos relevantes e descobertas ao longo do processo de aprendizado de modificações, os quais podem proporcionar melhorias em sua condução, e consequentemente, melhores resultados em trabalhos futuros.

5.4.1 Extração de modificações do histórico de repositórios

O processo de extração de modificações de código-fonte a partir do histórico permitiu a obtenção de um conjunto variado de modificações.

O primeiro ponto relevante consiste na elevada taxa de modificações por *commit*, assim como, a presença de um elevado número de alterações concentradas em um conjunto restrito de *commits*, na maioria dos repositórios. Em ambos os casos, o elevado número de modificações em diversos arquivos, ou mesmo dentro de um arquivo, dificultou a identificação de cada modificação individual, em decorrência do emaranhamento de trechos do código-fonte, também conhecido como *tangled code changes*.

Uma prática para resolução desse problema consiste na organização da equipe de desenvolvimento para a criação de *commits* com um único propósito, desse modo, somente modificações relacionadas devem ser contempladas. Essa prática permite a redução de escopo de cada *commit*, e consequentemente, facilita a análise manual ou automatizada de cada modificação. Dentre os repositórios analisados, o repositório *CheckStyle* apresentou *commits* com um nível maior de organização e próximos ao modelo descrito acima, como pode ser observado pelo re-

duzido número de alterações em cada *commit*. Essa diferença pode ser notada pelo número de modificações nos arquivos com ocorrência de padrões, conforme representado na Figura 5.10.

Tabela 5.10: Quantidade média de modificações nos arquivos com ocorrência dos padrões sintáticos.

Repositório	**Quantidade**
Refactoring Toy Example	2,67
Gson	15,85
Truth	4,0
JUnit 4	12,61
CheckStyle	4,4

Fonte: Elaborada pelo autor.

Outro ponto relevante quanto a extração das modificações consiste na identificação dos mapeamentos. A utilização do algoritmo *GumTree* permitiu o mapeamento de modificações de modo automático e performático, contudo existem ressalvas a serem destacadas. O primeiro aspecto consiste na falha de identificação de mapeamentos em alterações de atualização/movimentação como deleções seguidas por inserções. Esse comportamento foi observado em um número significante de modificações, seja em alterações entre arquivos ou dentro do mesmo arquivo. De modo a contornar essa limitação, foi empregada uma estratégia de revisões das modificações dentro de cada arquivo, assim como, entre arquivos modificados em um mesmo *commit*, visando a otimização da identificação dos mapeamentos e seus respectivos relacionamentos.

5.4.2 Tipos de modificações dos padrões sintáticos

O processo de aprendizado de padrões sintáticos permitiu a identificação de diferentes modificações entre os padrões nos repositórios analisados. Apesar das diferentes características de codificação, pode-se observar, de modo uniforme, um baixo percentual de padrões sintáticos com impacto relevante, em detrimento da totalidade de padrões sintáticos em cada repositório.

Dentre os tipos de modificações identificados ao longo das instâncias de padrões sintáticos, pode-se observar uma quantidade relevante de modificações de renomeação de classes, métodos e variáveis (identificadores em geral), assim como, a movimentação arquivos entre pacotes de classes. Um segundo tipo recorrente entre os padrões são as modificações relacionadas a implementação de funcionalidades, dentre as quais tem-se a substituição de tipos de variáveis e a modificação da assinatura de métodos para aceitação de uma nova classe/método implementado. Em ambas alternativas, tais modificações resultam em pequenas alterações adicionais em referências ou chamadas aos respectivos elementos.

Um terceiro tipo de modificação observado nos padrões sintáticos relevantes foram as refatorações de software. De modo geral, pode-se observar a utilização de técnicas de refatoração relacionadas com a renomeação e movimentação de entidades de código-fonte, como *move class/method/field* e *rename class/method/field/variable*, assim como as técnicas *push-down method/field* e *pull-up method/field* em modificações com contexto de herança entre as classes. Além disso, também foram identificadas algumas instâncias de refatorações compostas,

sendo combinadas técnicas de extração como *extract method* e *extract class*, e algumas das técnicas previamente listadas.

De modo geral, as modificações representadas pelos padrões sintáticos obtidos consistem em modificações sintaticamente simples e amplamente utilizadas em diversas localidades do projeto, e também se repetem em mais de um projeto. Por outro lado, quanto a modificações mais complexas, não foi possível identificar nenhum padrão sintático relacionado a uma alteração de maior porte, pois tais modificações geralmente são resultados de novas implementações, que resultam de novas especificações e não apresentam o mesmo nível de recorrência que outros tipos.

Nesse sentido, podemos concluir que modificações complexas tendem a não formar padrões sintáticos, devido a dependência do contexto do código-fonte da modificação juntamente com a funcionalidade que deve ser adicionada. Por outro lado, modificações sintaticamente mais simples não apresentam restrição quanto a contexto de execução no código-fonte, permitindo sua replicação ao longo de diferentes pontos dentro do repositório.

5.4.3 Classificação dos padrões sintáticos

Quanto a classificação dos padrões sintáticos, a utilização do presente conjunto de métricas para cálculo do impacto da manutenabilidade foi preciso em determinadas circunstâncias com menor número de modificações em cada arquivo, porém resultou em falhas na classificação entre melhoria ou degradação nas demais situações.

Durante a análise dos repositórios, foi identificada a alta taxa de modificações por arquivo, a qual impacta diretamente sob a identificação das modificações individuais, e também quanto ao impacto das modificações sob a manutenabilidade do código-fonte. Dentre o conjunto de arquivos com ocorrências de padrões sintáticos, pode-se observar que, na maioria dos casos, a variação da manutenabilidade indicada para as ocorrências dos padrões sintáticos não era proveniente da alteração vinculada ao padrão sintático, mas sim a outras modificações de inserção/deleção presentes nos mesmos arquivos.

Um segundo aspecto de falha na classificação dos padrões sintáticos foi o mapeamento incorreto de arquivos entre versões em decorrência de muitas alterações na estrutura do repositório. O incorreto mapeamento dos arquivos resultou na identificação de uma deleção de um arquivo juntamente com a inserção de outro arquivo com mesmo conteúdo, porém com caminhos diferentes, o que, em termos de modificações, resulta em um conjunto de modificações de impacto oposto quanto a variação da manutenabilidade.

Por fim, um aspecto positivo quanto ao modelo adotado, em arquivos com número reduzido de alterações, foi possível identificar o impacto direto de cada modificação de modo mais preciso. Em especial, pode-se salientar que em determinadas instâncias de refatoração, como modificações de movimentação, foi identificado impacto zero sob a manutenabilidade. Por outro lado, em ocorrências de refatorações como *push-down* e *pull-up method*, pode-se observar um comportamento variante quanto ao impacto sob a manutenabilidade, em decorrência do aumento do número de dependência da classes e do aumento/diminuição da sua complexidade. Nesse sentido, pode-se concluir que não necessariamente uma modificação de refatoração pode

ter um impacto positivo sob a manutenabilidade.

5.4.4 Identificação das dependências de cada modificação

O processo de identificação das modificações do processo proposto, de modo semelhante ao algoritmo *GumTree*, utiliza como entrada o par de arquivos de duas versões subsequentes, de modo que possam ser obtidas as modificações conduzidas nesse arquivo. Essa estratégia permite a identificação de cada alteração de modo preciso, contudo não necessariamente da modificação completa, porque, em contextos mais específicos, o conjunto de entidades de código alteradas estão contidas no mesmo arquivo, na maioria dos casos, o arquivo de destino da modificação não é o mesmo que o inicial.

Mediante esse cenário, o processo proposto neste trabalho também adotou a verificação de relacionamento das modificações entre todos os arquivos modificados no *commit*, de modo a estabelecer um mapeamento identificando a origem e o destino de cada modificação. Contudo, por meio da inspeção manual dos padrões sintáticos obtidos, pode-se observar que os padrões sintáticos identificados representam a modificação de modo parcial, em um número significativo de casos, devido a distribuição das dependências da modificação em diferentes arquivos, e não somente limitada a um arquivo de entrada e outro de saída.

Um exemplo de modificação simples que se enquadra nesse cenário consiste na refatoração de renomeação de método. Nesse caso, o processo de agrupamento da modificações categorizava a renomeação do método e a renomeação das chamadas a esse método como dois padrões sintáticos separados, pois apresentam estrutura sintática diferente, porém pertencem a modificação de modo completo. Como consequência, essa limitação impacta tanto a identificação das modificações, quanto a avaliação do impacto sob a manutenabilidade de software.

Como solução deve ser aprimorada a identificação das dependências entre entidades de código-fonte, de modo que seja possível o rastreamento do impacto de uma modificação e quais entidades estão envolvidas, e consequentemente, permitindo uma análise mais completa dos atributos observados, nesse caso, a manutenabilidade de software.

5.5 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos no processo de aprendizado de padrões sintáticos por meio de um visão geral e estudos de casos para cada repositório analisado. Por meio da visão geral dos resultados, foram apresentados alguns números obtidos de mapeamentos e padrões sintáticos extraídos dos repositórios, assim como as taxas de erro do modelo de classificação dos padrões sintáticos.

Em cada estudo de caso, foi apresentada a evolução da variação da manutenabilidade do repositório ao longo do período analisado e os padrões sintáticos obtidos, juntamente com a análise detalhe de uma amostra desses. Por fim, foi elaborada uma seção de discussão quanto ao processo de aprendizado e resultados obtidos indicando aspectos relevantes e melhorias ao processo proposto.

De modo complementar, para cada padrão sintático foi apresentada a variação média de cada métrica analisada nas ocorrências de cada padrão sintático, porém é importante salientar

que os dados apresentados são somente referentes aos padrões sintáticos detalhados. Os demais dados são acessíveis por meio do conjunto de dados extraído neste trabalho, como descrito no Apêndice A.

CAPÍTULO

6

Considerações Finais

Um repositório de software passa por muitas modificações com diferentes propósitos, as quais afetam diretamente a qualidade do códigofonte, e consequentemente, a manutenabilidade do software. As refatorações de software são modificações utilizadas como ferramentas para reestruturação do software, proporcionando melhoria da arquitetura interna do software. Contudo, tais modificações são majoritariamente simples e previamente catalogadas, além disso, as principais abordagens de identificação de oportunidades de refatoração tem limitações quanto a variedades de técnicas suportadas, o que pode não suprir as reais necessidades de modificações do desenvolvedor.

6.1 Contribuições e Limitações

Neste trabalho, foi apresentado um processo de aprendizado de modificações de código-fonte baseado em exemplos extraídos do histórico de versão de repositórios de software, com o intuito de identificar padrões sintáticos de modificações caracterizadas como melhorias, para que possam ser replicadas em repositórios por meio de sugestões, assim como, a identificação de anti-padrões ou modificações caracterizadas como degradações, que não devem ser sugeridas para uso em repositórios de software.

A partir de um dado repositório, o processo proposto extrai as modificações de código-fonte realizadas entre versões consecutivas, representada em cada *commit*, assim como um conjunto de métricas de software dos arquivos antes e depois da aplicação das modificações. Em seguida, as modificações extraídas são agrupadas como base na similaridade estrutural das entidades de código-fonte alteradas, de modo a formar grupos de padrões sintáticos de modificações. Por fim, cada agrupamento é avaliado quanto ao impacto sob a manutenabilidade por meio do variação das métricas nas instâncias de modificação, permitindo a classificação entre melhoria ou degradação.

A condução do processo de aprendizado em cada repositório foi capaz de identificar um conjunto restrito de padrões sintáticos, em sua maioria, modificações simples que envolvem

ações de movimentação e renomeação de entidades de código, assim como a identificação de instâncias de refatoração conduzidas de modo separado e também de forma composta. De modo geral, foi identificada uma maior recorrência de modificações simples para formação de padrões sintáticos, devido ao caráter mais estrutural (não funcional) dessas modificações e a independência de contexto, que permite sejam replicadas em diferentes localidades do repositório.

Perante aos objetivos propostos, almejava-se a obtenção desses tipos de modificações, porém, de modo adicional, esperava-se também a identificação de modificações mais complexas, ou menos comumente utilizadas. Nesse contexto, foi observado que características de codificação de cada repositório, como a quantidade de modificações por *commit*, e principalmente, a quantidade modificações por arquivo dificultam a identificação individual das modificações, e consequentemente, a mensuração e classificação do impacto da modificação sob a manutenabilidade, devido a influência mútua entre as partes.

Além disso, outra limitação está relacionada ao processo de identificação das modificações no histórico de versão, o qual analisa um par de arquivos em versões subsequentes, de modo a estabelecer uma relação de origem/destino da modificação. Contudo, essa estratégia não supre situações de modificações mais complexas, em que uma simples modificação em um único arquivo pode afetar diversos arquivos que são dependentes, requerindo o relacionamento dessas alterações em arquivos diferentes.

A principal contribuição deste trabalho consiste em uma das primeiras iniciativas para o aprendizado modificações que melhorem o código-fonte por meio de uma estratégia baseada em exemplos, juntamente com uma classificação das modificações utilizando métricas de software. Estudos prévios relacionados a utilização de exemplos de modificações tinham como foco modificações corretivas e defeitos, enquanto abordagens voltadas a melhorias de código-fonte permaneciam restritas a regras lógicas e heurísticas.

Como contribuições secundárias, durante a análise dos repositórios, foram extraídas modificações de um conjunto de *commits* em diferentes repositórios, assim como suas respectivas métricas, as quais podem ser utilizadas em estudos de replicação e comparativos, visando a melhoria do processo.

Adicionalmente, foi realizada uma Revisão Sistemática de Literatura (RSL) durante a fase de embasamento deste trabalho. A RSL permitiu a identificação das principais e mais relevantes abordagens e estratégias em cada uma das etapas necessárias para o aprendizado de modificações e, também, para classificação baseada em métricas de software. Como produto da RSL foi elaborado um artigo [1] que está em fase de ajustes.

6.2 Trabalhos Futuros

Como trabalhos futuros, duas tarefas relevantes permanecem para o aperfeiçoamento do processo proposto. A primeira tarefa consiste na adaptação do processo de identificação de modificações para o suporte de modificações que envolvem diversos arquivos, permitindo estabelecer todas as dependências entre as entidades de código-fonte alteradas.

[1] Disponível em: https://bit.ly/3tz7fC5

A segunda tarefa, em consequência da primeira, consiste em ajustes no processo de cálculo e classificação das modificações quanto ao impacto sob a manutenabilidade. Em decorrência do elevado número de modificações, foram identificadas falhas na classificação das modificações, indicando melhorias ou degradações devido ao impacto das demais alterações nos mesmo arquivos, e não relativo à própria modificação analisada.

Adicionalmente, também é sugerida a replicação desse estudo utilizando diferentes repositórios de software, preferencialmente utilizando repositórios com boas práticas de programação e padrões organizados de controle de mudanças ao longo do histórico de versão, assim como adição de suporte a outras linguagens de programação.

Referências

ABREU, E.; BRITO, F. The mood metrics set. In: *proc. ECOOP*, 1995, p. 267.

AL DALLAL, J. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and software Technology*, v. 58, p. 231–249, disponível em: http://dx.doi.org/10.1016/j.infsof.2014.08.002, 2015.

ARNOLD, R. S. *Tutorial on software restructuring*. IEEE Computer Society Press, disponível em: https://doi.org/10.1109/5.24146, 1986.

BAIL, W. G.; ZELKOWITZ, M. V. Program complexity using hierarchical abstract computers. *Computer languages*, v. 13, n. 3-4, p. 109–123, disponível em: https://doi.org/10.1016/0096-0551(88)90019-7, 1988.

BRODER, A. Z. On the resemblance and containment of documents. In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, IEEE, disponível em: http://dx.doi.org/10.1109/SEQUEN.1997.666900, 1997, p. 21–29.

CHACON, S.; STRAUB, B. *Pro git*. Apress, 2014.

CHAWATHE, S. S.; RAJARAMAN, A.; GARCIA-MOLINA, H.; WIDOM, J. Change detection in hierarchically structured information. In: *Acm Sigmod Record*, ACM, disponível em: http://dx.doi.org/10.1145/233269.233366, 1996, p. 493–504.

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, disponível em: https://doi.org/10.1145/2939672.2939785, 2016, p. 785–794.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, v. 20, n. 6, p. 476–493, disponível em: http://dx.doi.org/10.1109/32.295895, 1994.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE software*, v. 7, n. 1, p. 13–17, disponível em: https://doi.org/10.1109/52.43044, 1990.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. Using metrics to evaluate software system maintainability. *Computer*, v. 27, n. 8, p. 44–49, disponível em: http://doi.org/10.1109/2.303623, 1994.

DEL ESPOSTE, A. D. M. *Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via dw e plataforma de monitoramente de código-fonte*. Tese de Doutoramento, Monografia. Universidade de Brasília, 2014.

DIG, D.; COMERTOGLU, C.; MARINOV, D.; JOHNSON, R. Automated detection of refactorings in evolving components. In: *European Conference on Object-Oriented Programming*, Springer, disponível em: http://dx.doi.org/10.1007/11785477_24, 2006, p. 404–428.

DOTZLER, G.; PHILIPPSEN, M. Move-optimized source code tree differencing. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, disponível em: http://dx.doi.org/10.1145/3106237.3106276, 2016, p. 660–671.

DUBEY, S. K.; RANA, A. Assessment of maintainability metrics for object-oriented software system. *ACM SIGSOFT Software Engineering Notes*, v. 36, n. 5, p. 1–7, disponível em: https://doi.org/10.1145/2020976.2020983, 2011.

DURIEUX, T.; CORNU, B.; SEINTURIER, L.; MONPERRUS, M. Dynamic patch generation for null pointer exceptions using metaprogramming. *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017*, disponível em: https://doi.org/10.1109/SANER.2017.7884635, 2017.

FALLERI, J.-R.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and accurate source code differencing. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, ACM, disponível em: http://dx.doi.org/10.1145/2642937.2642982, 2014, p. 313–324.

FLURI, B.; WUERSCH, M.; PINZGER, M.; GALL, H. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, v. 33, n. 11, p. 725–743, disponível em: http:/dx.doi.org/10.1109/TSE.2007.70731, 2007.

FOWLER, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

FOWLER, M. Catalog of refactorings. https://refactoring.com/catalog/, accessed: 2019-10-06, 2019.

FRICK, V.; GRASSAUER, T.; BECK, F.; PINZGER, M. Generating accurate and compact edit scripts using tree differencing. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, disponível em: http://dx.doi.org/10.1109/ICSME.2018.00036, 2018, p. 264–274.

HALSTED, M. Elements of software science (operating and programming systems series). 1977.

HANAM, Q.; BRITO, F. S. D. M.; MESBAH, A. Discovering bug patterns in javascript. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, disponível em: http://dx.doi.org/10.1145/2950290.2950308, 2016, p. 144–156.

HARRISON, R.; COUNSELL, S. J.; NITHI, R. V. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, v. 24, n. 6, p. 491–496, disponível em: http://doi.org/10.1109/32.689404, 1998.

HEAD, A.; GLASSMAN, E.; SOARES, G.; SUZUKI, R.; FIGUEREDO, L.; D'ANTONI, L.; HARTMANN, B. Writing reusable code feedback at scale with mixed-initiative program synthesis. In: *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, ACM, disponível em: http://dx.doi.org/10.1145/3051457.3051467, 2017, p. 89–98.

HEITLAGER, I.; KUIPERS, T.; VISSER, J. A practical model for measuring maintainability. In: *6th international conference on the quality of information and communications technology (QUATIC 2007)*, IEEE, disponível em: https://doi.org/10.1109/QUATIC.2007.8, 2007, p. 30–39.

ISLAM, M. R.; ZIBRAN, M. F. On the characteristics of buggy code clones: A code quality perspective. In: *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, IEEE, disponível em: http://dx.doi.org/10.1109/IWSC.2018.8327315, 2018, p. 23–29.

ISO/IEC-9126, I. Iec 9126-1: Software engineering-product quality-part 1: Quality model. *Geneva, Switzerland: International Organization for Standardization*, v. 21, 2001.

KIM, M.; NOTKIN, D. Discovering and representing systematic code changes. In: *2009 IEEE 31st International Conference on Software Engineering*, IEEE, disponível em: http://dx.doi.org/10.1109/ICSE.2009.5070531, 2009, p. 309–319.

LI, W.; HENRY, S. Maintenance metrics for the object oriented paradigm. In: *[1993] Proceedings First International Software Metrics Symposium*, IEEE, disponível em: https://doi.org/10.1109/METRIC.1993.263801, 1993, p. 52–60.

LIU, Y.; LI, Y.; GUO, J.; ZHOU, Y.; XU, B. Connecting software metrics across versions to predict defects. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, disponível em: http://dx.doi.org/10.1109/SANER.2018.8330212, 2018, p. 232–243.

LORENZ, M.; KIDD, J. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

MALHOTRA, R.; KHANNA, M. Investigation of relationship between object-oriented metrics and change proneness. *International Journal of Machine Learning and Cybernetics*, v. 4, n. 4, p. 273–286, disponível em: http://dx.doi.org/10.1007/s13042-012-0095-7, 2013.

MARTINEZ, M.; DUCHIEN, L.; MONPERRUS, M. Automatically extracting instances of code change patterns with ast analysis. In: *2013 IEEE international conference on software maintenance*, IEEE, disponível em: http://dx.doi.org/10.1109/ICSM.2013.54, 2013, p. 388–391.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, , n. 4, p. 308–320, disponível em: https://doi.org/10.1109/TSE.1976.233837, 1976.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese de Doutoramento, Universidade de São Paulo, 2013.

MENG, N.; KIM, M.; MCKINLEY, K. S. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, v. 46, n. 6, p. 329–342, disponível em: http://dx.doi.org/10.1145/1993316.1993537, 2011.

MENG, N.; KIM, M.; MCKINLEY, K. S. Lase: locating and applying systematic edits by learning from examples. In: *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, disponível em: https://doi.org/10.1109/ICSE.2013.6606596, 2013, p. 502–511.

MENS, T.; TOURWÉ, T. A survey of software refactoring. *IEEE Transactions on software engineering*, v. 30, n. 2, p. 126–139, disponível em: http://dx.doi.org/10.1109/TSE.2004.1265817, 2004.

MICROSOFT, S. Code metrics – maintainability index. [*On-line*]. Disponível em http://www-cs-faculty.stanford.edu/~uno/abcde.html

MOLDEREZ, T.; STEVENS, R.; DE ROOVER, C. Mining change histories for unknown systematic edits. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, disponível em: http://dx.doi.org/10.1109/MSR.2017.12, 2017, p. 248–256.

MOLNAR, A.; MOTOGNA, S. Discovering maintainability changes in large software systems. In: *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, disponível em: https://doi.org/10.1145/3143434.3143447, 2017, p. 88–93.

MONPERRUS, M. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, v. 51, disponível em: https://doi.org/10.1145/3105906, 2018.

MONPERRUS, M. The living review on automated program repair. 2019.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, v. 38, n. 1, p. 5–18, disponível em: http://dx.doi.org/10.1109/TSE.2011.41, 2012.

NEGARA, S.; CODOBAN, M.; DIG, D.; JOHNSON, R. E. Mining fine-grained code changes to detect unknown change patterns. In: *Proceedings of the 36th International Conference on Software Engineering*, ACM, disponível em: http://dx.doi.org/10.1145/2568225.2568317, 2014, p. 803–813.

NGUYEN, H. A.; NGUYEN, A. T.; NGUYEN, T. T.; NGUYEN, T. N.; RAJAN, H. A study of repetitiveness of code changes in software evolution. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, disponível em: http://dx.doi.org/10.1109/ASE.2013.6693078, 2013, p. 180–190.

NISA, I. U.; AHSAN, S. N. Fault prediction model for software using soft computing techniques. In: *2015 International Conference on Open Source Systems & Technologies (ICOSST)*, IEEE, disponível em: http://dx.doi.org/10.1109/ICOSST.2015.7396406, 2015, p. 78–83.

OMAN, P.; HAGEMEISTER, J. Metrics for assessing a software system's maintainability. In: *Proceedings Conference on Software Maintenance 1992*, IEEE Computer Society, disponível em: https://doi.org/10.1109/ICSM.1992.242525, 1992, p. 337–338.

OPDYKE, W. F. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. Tese de Doutoramento, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

OSMAN, H.; LUNGU, M.; NIERSTRASZ, O. Mining frequent bug-fix code changes. In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, IEEE, disponível em: http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747191, 2014, p. 343–347.

PHOTHILIMTHANA, P. M.; SRIDHARA, S. High-coverage hint generation for massive courses: Do automated hints help cs1 students? In: *Proceedings of the 2017 acm conference on innovation and technology in computer science education*, ACM, disponível em: https://doi.org/10.1145/3059009.3059058, 2017, p. 182–187.

POLOZOV, O.; GULWANI, S. Flashmeta: a framework for inductive program synthesis. In: *ACM SIGPLAN Notices*, ACM, disponível em: http://dx.doi.org/10.1145/2858965.2814310, 2015, p. 107–126.

PRETE, K.; RACHATASUMRIT, N.; SUDAN, N.; KIM, M. Template-based reconstruction of complex refactorings. In: *2010 IEEE International Conference on Software Maintenance*, IEEE, disponível em: http://dx.doi.org/10.1109/ICSM.2010.5609577, 2010, p. 1–10.

RATZINGER, J.; SIGMUND, T.; GALL, H. C. On the relation of refactorings and software defect prediction. In: *Proceedings of the 2008 international working conference on Mining software repositories*, ACM, disponível em: http://dx.doi.org/10.1145/1370750.1370759, 2008, p. 35–38.

RAYCHEV, V.; SCHÄFER, M.; SRIDHARAN, M.; VECHEV, M. Refactoring with synthesis. In: *ACM SIGPLAN Notices*, ACM, disponível em: http://dx.doi.org/10.1145/2544173.2509544, 2013, p. 339–354.

ROLIM, R.; SOARES, G.; D'ANTONI, L.; POLOZOV, O.; GULWANI, S.; GHEYI, R.; SUZUKI, R.; HARTMANN, B. Learning syntactic program transformations from examples. In: *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, disponível em: http://dx.doi.org/10.1109/ICSE.2017.44, 2017, p. 404–415.

ROLIM, R.; SOARES, G.; GHEYI, R.; D'ANTONI, L. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018a.

ROLIM, R.; SOARES, G. A.; GHEYI, R. *Learning syntactic program transformations from examples*. Tese de Doutoramento, Universidade Federal de Campina Grande, 2018b.

ROMBACH, H. D. Design measurement: Some lessons learned. *IEEE Software*, v. 7, n. 2, p. 17–25, disponível em: https://doi.org/10.1109/52.50770, 1990.

SEBESTA, R. W. *Conceitos de linguagens de programação*. 9 ed. Porto Alegre: Bookman, 2011.

SEREF, B.; TANRIOVER, O. Software code maintainability: A literature review. *International Journal of Software Engineering & Applications (IJSEA)*, v. 7, p. 69–87, 2016.

SHARMA, M.; KUMARI, M.; SINGH, V. Post release versions based code change quality metrics. In: *Proceedings of the Third International Symposium on Women in Computing and Informatics*, ACM, disponível em: http://dx.doi.org/10.1145/2791405.2791466, 2015, p. 235–243.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, disponível em: http://dx.doi.org/10.1145/2950290.2950305, 2016, p. 858–870.

SILVA, D.; VALENTE, M. T. Refdiff: detecting refactorings in version histories. In: *Proceedings of the 14th International Conference on Mining Software Repositories*, IEEE Press, disponível em: http://dx.doi.org/10.1109/MSR.2017.14, 2017, p. 269–279.

SOARES, G.; GHEYI, R.; SEREY, D.; MASSONI, T. Making program refactoring safer. *IEEE software*, v. 27, n. 4, p. 52–57, disponível em: http://dx.doi.org/10.1109/MS.2010.63, 2010.

TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D.; DIG, D. Accurate and efficient refactoring detection in commit history. In: *Proceedings of the 40th International Conference on Software Engineering*, ACM, disponível em: http://dx.doi.org/10.1145/3180155.3180206, 2018, p. 483–494.

WANG, S.; WEN, M.; CHEN, L.; YI, X.; MAO, X. How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM/IEEE, disponível em: https://doi.org/10.1109/ESEM.2019.8870172, 2019.

WELKER, K. D. The software maintainability index revisited. *CrossTalk*, v. 14, p. 18–21, 2001.

XING, Z.; STROULIA, E. Umldiff: an algorithm for object-oriented design differencing. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ACM, disponível em: http://dx.doi.org/10.1145/1101908.1101919, 2005, p. 54–65.

YUAN, Z.; LIU, C.; YU, L.; ZHANG, L. Changechecker: A tool for defect prediction in source code changes based on incremental learning method. In: *Proceedings of 2013*

3rd International Conference on Computer Science and Network Technology, IEEE, disponível em: http://dx.doi.org/10.1109/ICCSNT.2013.6967127, 2013, p. 349–354.

ZHONG, H.; MENG, N. Towards reusing hints from past fixes. *Empirical Software Engineering*, v. 23, n. 5, p. 2521–2549, disponível em: http://dx.doi.org/10.1007/s10664-017-9584-3, 2018.

APÊNDICE

A

Conjunto de dados

Neste capítulo é apresentada a organização do conjunto de dados extraídos no processo de aprendizado de padrões sintáticos, tanto os arquivos com as estruturas de código-fonte, quanto as métricas de qualidade extraídas para o processo de classificação.

No conjunto de dados [1], estão organizados por repositório no diretório *metadata*. Adicionalmente, também estão disponíveis os *scripts* para construção do modelo de regressão e classificação dos padrões sintáticos.

A.1 Arquivo de métricas de software

O arquivo de métricas consiste em um arquivo compactado, identificado por *metrics.zip*, em cada repositório. Esse arquivo contém as métricas extraídas de todos os arquivos (somente com extensão .java) do repositório organizados em cada *commit*. Os dados referentes a cada *commit* segue o formato *CSV*, conforme representa na Figura A.1.

```
filename,cbo,dit,lcom,loc,nosi,numberOfFields,numberOfMethods,rfc,wmc
src/org/animals/CowRenamed.java,0.0,1.0,1.0,9.0,0.0,0.0,2.0,1.0,2.0
src/org/animals/Dog.java,1.0,1.0,3.0,11.0,0.0,1.0,3.0,2.0,3.0
src/org/animals/Labrador.java,1.0,2.0,0.0,3.0,0.0,0.0,0.0,0.0,0.0
...
```

Figura A.1: Estrutura do arquivo das métricas extraídas dos arquivos de código-fonte.

A.2 Arquivo dos mapeamentos de modificações

O arquivo de mapeamentos, identificado por *mapping.zip*, contém a relação de todas as modificações realizadas em todos os arquivos de um mesmo *commit*, em que cada mapeamento é

[1] https://drive.google.com/drive/folders/1C2MAv9g-V2Ae-ZumxB1ntJqKlqOZrHds?usp=sharing

classificado entre os tipos: inserção, deleção, atualização ou movimentação.

Os dados referentes a cada *commit* são separadas por arquivos, o qual segue a estrutura representada na Figura A.2.

```
{
  "repository": "refactoring-toy-example",
  "branch": "d4bce13a443cf12da40a77c16c1e591f4f985b47",
  "commit": "0a46ed5c56c8b1576dfc92f3ec5bc2f0ea68aafe",
  "mappings": [
    {
      "source": {
        "element": [
          "protected int age;"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
          #containedType[name=AnimalMarilho]#field[name=age]",
        "type": "FieldElement"
      },
      "destination": {
        "element": [
          "protected int age;"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
          #containedType[name=Reptile]#field[name=age]",
        "type": "FieldElement"
      },
      "type": "move",
      "oldFile": "src/org/reptile/AnimalMarilho.java",
      "newFile": "src/org/reptile/Reptile.java"
    },
    ...
  ]
}
```

Figura A.2: Estrutura do arquivo de mapeamentos de cada *commit*.

Em cada arquivo, as informações relativas ao repositório, *branch* e *commit* cujos dados foram extraídos são especificados, juntamente com a relação de mapeamentos obtida. Em cada mapeamento, são identificados os arquivos de origem e destino, o tipo do mapeamento, assim como as estruturas sintáticas antes e depois da alteração, cada qual com o caminho para identificação das estrutura sintáticas.

A.3 Arquivo de mapeamentos em *commits* sequenciais

O arquivo de mapeamentos em *commits* sequenciais, identificado por *multipattern.zip*, consiste em um arquivo compactado. Esse arquivo contém a relação de mapeamentos identificados em cada *commit*, e em casos especiais, quando a modificação respectiva do mapeamento se estende além de um único *commit*, armazena-se a sequência de mapeamentos relacionados.

A estruturação do arquivo é divida em um diretório para cada *commit*, o qual contém um arquivo no formato *JSON* para cada mapeamento.

```
{
  "code": 202,
  "commits": [
    "0a46ed5c56c8b1576dfc92f3ec5bc2f0ea68aafe"
  ],
  "size": 1,
  "items": [
    {
      "source": {
        "element": [
          "protected int age;"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
          #containedType[name=AnimalMarilho]#field[name=age]",
        "type": "FieldElement"
      },
      "destination": {
        "element": [
          "protected int age;"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
          #containedType[name=Reptile]#field[name=age]",
        "type": "FieldElement"
      },
      "type": "move",
      "oldFile": "src/org/reptile/AnimalMarilho.java",
      "newFile": "src/org/reptile/Reptile.java"
    }
  ]
}
```


Figura A.3: Estrutura do arquivo de mapeamentos em *commits* sequenciais.

O arquivo de cada mapeamento contém o código de identificação do mapeamento, o conjunto de *commits* que contém os mapeamento e a sequência de mapeamentos relacionados.

A.4 Arquivo de padrões sintáticos

O arquivo de padrões sintáticos consiste em um arquivo compactado, identificado por *clusters.zip*, que contém o conjunto de todos os padrões sintáticos identificados no repositório. Cada padrão sintático é detalhado em arquivos individuais e com identificadores únicos, conforme representado na Figura A.4.

O arquivo de cada mapeamento contém o código de identificação do padrão, o mapeamento-base que serve de modelo para estrutura sintática da modificação (conforme registrado detalhado na Seção A.3) e, por fim, a conjunto de identificadores dos mapeamentos considerados como ocorrências do padrão sintático.

```
{
  "code": 2,
  "base": {
    "code": 204,
    "commits": [
      "0a46ed5c56c8b1576dfc92f3ec5bc2f0ea68aafe"
    ],
    "size": 1,
    "items": [{
      "source": {
        "element": [
          "public void setName(int name) {\n    this.name = name;\n}"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
        #containedType[name=AnimalMarilho]#method[signature=setNam(int)]",
        "type": "MethodElement"
      },
      "destination": {
        "element": [
          "public void setName(int name) {\n    this.name = name;\n}"
        ],
        "path": "#subPackage[name=org]#subPackage[name=reptile]
        #containedType[name=Reptile]#method[signature=setName(int)]",
        "type": "MethodElement"
      },
      "type": "move",
      "oldFile": "src/org/reptile/AnimalMarilho.java",
      "newFile": "src/org/reptile/Reptile.java"
    }]
  },
  "items": [204,72,73,212,191,192,186,66,67,100,179,173,178,105]
}
```


Figura A.4: Estrutura do arquivo de um padrão sintático.

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 06/02/2021

Leandro Ungari Cayres

Assinatura do autor