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

Renan Pereira Biazini

# Uma Abordagem como suporte a Análise de Impacto de Mudanças em código-fonte orientado a objetos utilizando Visualização de Software e Métricas de Manutenibilidade

São José do Rio Preto
2020

Renan Pereira Biazini

# Uma Abordagem como suporte a Análise de Impacto de Mudanças em código-fonte orientado a objetos utilizando Visualização de Software e Métricas de Manutenibilidade

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, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Orientador: Prof. Dr. Rogério Eduardo Garcia

São José do Rio Preto
2020

B579a

Biazini, Renan Pereira

Uma Abordagem como suporte a Análise de Impacto de Mudanças em código-fonte orientado a objetos utilizando Visualização de Software e Métricas de Manutenibilidade / Renan Pereira Biazini. -- São José do Rio Preto, 2020

79 f. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto

Orientador: Rogério Eduardo Garcia

1. Análise de Impacto de Mudanças. 2. Visualização de Software. 3. Métricas de Manutenibilidade. 4. Engenharia de Software. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Renan Pereira Biazini

# Uma Abordagem como suporte a Análise de Impacto de Mudanças em código-fonte orientado a objetos utilizando Visualização de Software e Métricas de Manutenibilidade

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, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

BANCA EXAMINADORA

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

Prof. Dr. Leonardo Castro Botega
Professor Doutor
Centro Universitário Eurípides de Marília–UNIVEM

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

Presidente Prudente
29 de Maio de 2020

# *Agradecimentos*

Agradeço, em primeiro lugar, a Deus, que me proporcionou este mestrado e me sustentou de pé até o presente momento. A Ele, toda a honra e glória. Obrigado, Senhor.

Agradeço à minha esposa Paola, que sempre me apoiou, principalmente nos momentos difícies, e foi a minha força quando eu não podia caminhar e foi a minha oração quando eu não podia ajoelhar. Essa vitória é nossa, gata.

Agradeço a meus pais, que me deram apoio moral, financeiro durante toda a minha vida, e são os reponsáveis por tudo o que sou. Essa vitória é de vocês também. Obrigado, pai e mãe.

Agradeço a meu sogro e minha sogra, que sempre torceram por mim em cada passo deste mestrado. Em tudo, eu sei que posso contar com vocês. Essa vitória é de vocês também.

Agradeço a meu orientador, Rogério, pelo auxílio e direcionamento dedicados a mim, sem o qual não existiria este trabalho e lhe devo minha gratidão eterna. Obrigado, amigo.

Agradeço a meus companheiros da pós-graduação da Unesp de Presidente Prudente – à turma de Ciência da Computação – em especial ao pessoal do LAPESA. Obrigado, amigos.

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

"[...] Bem-aventurado o homem que sofre a tentação; porque, quando for provado, receberá a coroa da vida, a qual o Senhor tem prometido aos que o amam." - Tiago 1:12 (Bíblia, 1969)

# *Resumo*

A Análise de Impacto de Mudanças visa a identificar partes de um sistema de software afetados pela implementação de uma mudança proposta. E, resoluções de *bugs* ou implementação de novas funcionalidades são atividades habituais em um contexto de desenvolvimento de software. A quantidade de tempo e esforço gastos na manutenção do software se torna muito alto, tornando a análise de impacto ainda mais crucial e importante. No contexto do paradigma orientado a objetos, o esforço é ainda maior devido a complexidade de relacionamentos entre as entidades que compõe o código-fonte de um software analisado. Como auxílio a análise de impacto, técnicas automatizadas foram propostas na literatura com o intuito de diminuir o custo de executá-la, visto que comumente a análise de impacto é feita manualmente. No entanto, as técnicas que auxiliam o processo de identificação de impactados ainda são altamente dependentes da experiência com o software sob análise. Os estudos na literatura não têm foco na análise dos possíveis impactados identificados pelas técnicas, restringem-se a apenas identificar o impacto. Então, este trabalho propõe uma Abordagem visual como suporte a análise de impacto de código-fonte orientado a objetos utilizando técnicas de Visualização de Software e Métricas de Manutenibilidade, visando prover estratégias de priorização baseada na complexidade de manutenção de cada entidade impactada.

Palavras-chave: Análise de Impacto de Mudanças, Visualização de Software, Métricas de Manutenibilidade, Engenharia de Software.

# *Abstract*

A Change Impact Analysis aims to identify parts of a software system affected by the implementation of a proposed change. And, bug fixes or implementation of new features are common activities in a software development context. The amount of time and effort spent on maintaining the software becomes very high, making impact analysis even more crucial and important. In the context of the object-oriented paradigm, the effort is even greater due to the complexity of relationships between the entities that make up the source code of an analyzed software. As an aid to impact analysis, automated techniques have been proposed in the literature to reduce the cost of carrying it out, since impact analysis is usually done manually. However, the techniques that help the impact identification process are still highly dependent on the experience with the software under analysis. Studies in the literature do not focus on the analysis of the possible impacts identified by the techniques, they are restricted to only identifying the impact. So, this work proposes a visual approach to support object-oriented source code impact analysis using Software Visualization techniques and Maintenance Metrics, aiming to provide prioritization strategies based on the maintenance complexity of each impacted entity.

Keywords: Change Impact Analysis, Software Visualization, Maintenance Metrics, Software Engineering.

# Lista de Figuras

# *Lista de Tabelas*

# Sumário

CAPÍTULO
1

# *Introdução*

Durante o ciclo de vida do software, há alterações para atender demandas por atualizações ou correções de defeitos (Pressman e Maxim, 2014). A implementação de uma mudança não é uma tarefa simples, pois depende de maior esforço e custo associados ao produto (Ghotra et al., 2015), quando comparados ao desenvolvimento. As atividades de gerenciamento de mudanças tendem a consumir mais da metade do custo total de um projeto (Pressman e Maxim, 2014). No contexto da Orientação a Objetos, esse custo pode aumentar devido à complexidade da estrutura dinâmica e associações entre entidades. Essa característica causa a propagação dos efeitos das mudanças implementadas em um software (Lee e Youn, 2016).

Uma mudança no código de um sistema compreende (Bohner, 2002b), basicamente, nos seguintes passos:

1. Entender como a mudança afeta o software;
2. Decidir se será implementada a mudança proposta ou não;
3. Implementação da mudança, caso se decida implementar;
4. Testar o sistema alterado.

No entanto, ao implementar uma mudança há a necessidade de uma garantia do funcionamento não somente dos artefatos modificados diretamente, mas outros artefatos relacionados com o impacto direto. Analisar os todos os artefatos do software (força bruta) também é inviável, mesmo com mudanças em software com pouca complexidade. A Análise de Impacto de Mudanças (ou simplesmente Análise de Impacto - AI) é definida como *a identificação das potenciais consequências de uma mudança, fundamentada em estimativas do que precisa ser modificado para realizar uma mudança* (Bohner e Arnold, 1996). Similarmente, Turver e Munro (1994) a definem como *a avaliação das consequências de uma mudança feita em um módulo em outros módulos do sistema*. Já Pfleeger e Bohner afirmam que a AI não só permite

avaliar as consequências de uma mudança, mas também prover estimativas de esforço, tempo e custo da implementação de uma modificação (Pfleeger e Bohner, 1990). em síntese, tanto a definição quanto a afirmação trazem a essência da AI: para estimar de modo mais apropriado, é preciso conhecer os elementos (trechos de código) afetados e os potenciais desdobramentos da mudança de tais trechos (Delfim et al., 2015).

Além disso, um sistema mal projetado ou mesmo a falta de familiaridade dos envolvidos na manutenção com o código tornam difícil a modificação de um software. A utilização de ferramentas, como uma *Integrated Development Environment* (*IDE*), pode facilitar a tarefa de alteração de códigos-fonte (Cito et al., 2019). Entretanto, a possível presença de forte acoplamento entre trechos de código – classes, por exemplo – torna programadores suscetíveis a erros ao realizar alterações. Nesse caso, ao realizar alterações em um software é preciso garantir que novos defeitos não sejam inseridos e que as alterações não tenham como efeito colateral a introdução de defeitos em outras partes do código (gerar inconsistência). Para tanto, a completa compreensão do sistema é necessária e fundamental para que uma mudança possa ser feita adequada e completamente (Delfim et al., 2015).

O resultado produzido a partir de uma AI pode ser utilizado no planejamento das mudanças, rastreando os seus efeitos e permitindo que tais efeitos sejam conhecidos antes de sua realização. Além disso, o resultado pode ser usado para identificar casos de teste que precisam ser executados novamente, a fim de garantir que a mudança tenha sido implementada corretamente e não tenha afetado o funcionamento e a confiabilidade do software como um todo (Maia, 2009). A AI é um processo importante de gerenciamento de mudanças e evolução de software, pois é usado para garantir o funcionamento do sistema após a implementação.

A análise de impacto utilizada na indústria de software tem como foco a garantia de qualidade, e também, é usada como justificativa em avaliações de segurança (Borg et al., 2017). Nesse caso, ela é recomendada no uso de padrões de processo em setores como indústria automotiva, aviação e setor ferroviário. Padrões de processo recomendam que uma AI seja realizada antes de qualquer mudança seja realmente implementada e executada, mas não provêm detalhes de como ela deve ser conduzida, tampouco como usufruir e validar os resultados obtidos para tomadas de decisão (Borg et al., 2017). Assim, a maneira de como uma AI deve ser realizada varia de organização para organização de desenvolvimento de software.

Todavia, a identificação do impacto ocasionado por uma mudança não é um processo trivial. Na maioria das vezes é realizada manualmente e geralmente apresentando um cenário em que os analistas não utilizam ferramentas adequadas aliado a quantidades massivas de informações analisadas, acarretando em um esforço muito grande para realizá-la (de La Vara et al., 2016). Com isso, diversas técnicas para apoiar a análise de impacto foram criadas com o objetivo de diminuir os custos de esforço e tempo desta atividade, axiliando na criação do conjuntos de impactos estimados cada vez mais precisos. Sobretudo, técnicas que são utilizadas como suporte a AI em que os elementos impactados são identificados em partes do código-fonte do sistema analisado. Estas técnicas tem como objetivo em comum mitigar a dependência de experiência do analista com a linguagem de programação do código-fonte, com o software

analisado e com o processo de análise.

## 1.1 Motivação e Formulação do Problema

A análise de impacto de mudanças é utilizada como suporte a muitos processos críticos no desenvolvimento de software, como: propagação de alterações, compreensão do software analisado e seleção de casos de teste para garantias de qualidade. Como no ciclo de vida de uma software, há uma natural evolução do código-fonte (para resoluções de *bugs* ou implementação de novas funcionalidades), a quantidade de tempo e esforço gastos na manutenção do software se torna muito alto, tornando a AI ainda mais crucial e importante (Li et al., 2013).

Apesar da automatização da identificação de possíveis impactados por uma mudança utilizando uma técnica de análise de impacto, o planejamento de execução de mudança proposta ainda é dependente da experiência do gerente de mudança com o projeto analisado, uma vez que a maioria das técnicas de análise tem como retorno apenas quais entidades foram possivelmente impactadas, e não uma análise mais ampla da própria mudança e sua propagação quando se leva em consideração o software como um todo. Com isso, é necessário prover estratégias de priorização baseada na complexidade de manutenção de cada entidade impactada, o facilitando a utilização efetiva da AI na fase de manutenção de software. E apesar de serem encontrados na literatura alguns exemplos de definições de processos de análise de impacto, que estabelecem um fluxo de atividades que compõe uma análise de impacto, as etapas destes processos são descritas de forma abstrata e não provem detalhes em sua definição que auxiliam efetivamente o analista em tomadas de decisão pois definem apenas quais as etapas para uma análise de impacto e não como cada etapa deverá ser executada.

Outra estratégia utilizada para auxiliar a análise de impacto é a utilização de Visualização de Software para que os elementos impactados sejam identificados, auxiliando a AI, utilizando em conjunto com técnicas automatizadas de análise de impacto, mapeando entidades do código-fonte (Follett e Hoeber, 2010) (Mercer et al., 2012) (Mohamad, 2010). No entanto, a análise do conjunto de possíveis impactados gerados é muito superficial, se limitando apenas ao destaque de elementos impactados nas visualizações, o que não é o suficiente para auxiliar em tomadas de decisão de direcionamento da implementação da mudança ou mensurar dificuldade de execução da mudança.

Estes problemas e contextos motivaram o presente trabalho, cujo objetivo é exposto a seguir.

## 1.2 Objetivo e Metodologia

O presente trabalho tem como objetivo geral a elaboração de uma abordagem como suporte à análise de impacto de mudanças analisando aspectos do conjunto de possíveis impactados gerados pela execução de uma técnica automatizada de análise em código-fonte orientado a objetos utilizando Visualização de Software e Métricas de Manutenibilidade. Como objetivos específicos, têm-se: definir um processo de análise de impacto representado por etapas detalhadas e aplícáveis, e analisar não somente o impacto de uma mudança proposta, mas qual elemento propagou um impacto a outro elemento, que tipo de impacto foi propagado e a análise da di-

ficuldade de implementação da mudança proposta, levando em consideração a dificuldade de manutenção de cada entidade possivelmente impactada.

Para atingir o objetivo principal e os objetivos específicos, uma revisão bibliográfica da literatura foi realizada e apresentada neste trabalho, identificando técnicas de análise de impacto de mudanças em código-fonte, técnicas de visualização de software e métricas de manutenibilidade. Foi também identificado na literatura processos de análise de impacto de mudanças e ferramentas de análise de impacto que utilizam visualização de software para identificar os possíveis impactados. Este levantamento bibliográfico foi utilizado como base para a definição da abordagem.

Para entendimento da aplicação da abordagem, uma ferramenta chamada VisImpala foi desenvolvida, implementando todos os aspectos definidos pela abordagem (avaliando a mudança por meio de perspectivas), compreendendo a extração de métricas de manutenção e relacionamentos de um código-fonte orientado a objetos.

Para que a abordagem e a ferramenta sejam validadas, fez-se necessário a execução de testes de execução e análise dos resultados como prova de conceito. Fez-se necessário avaliação da escalabilidade da abordagem através de testes em um software de grande porte de código-aberto, no qual foram necessários ajustes na ferramenta, tanto na extração de dados quanto nas técnicas de visualização, para ser possível a análise de código-fonte extenso.

Após a condução dos testes e avaliação dos resultados, foi possível constatar que a abordagem proveu suporte necessário à análise de impacto de mudanças. Adicionalmente, algumas contribuições e lições aprendidas puderam ser obtidas para a condução de futuros experimentos.

## 1.3 Organização do Trabalho

Para expor o trabalho desenvolvido, esta dissertação encontra-se organizada como segue:

- No Capítulo 2 é apresentada uma fundamentação teórica realizada a partir de um levantamento bibliográfico a respeito dos assuntos envolvidos na realização da pesquisa deste trabalho, e que foram utilizados como base para a definição da abordagem. São eles: Classificação de uma Análise de Impacto de Mudanças, Técnica de Análise de Impacto de Hattori, Métricas de Manutenibilidade, Técnicas de Visualização de Software e Trabalhos Relacionados.

- No Capítulo 3 são apresentadas a abordagem proposta por nosso trabalho e a ferramenta que a implementa, chamada VisImpala.

- No Capítulo 4 são apresentados os testes realizados com a VisImpala a fim de se obter por meio dos resultados, a validação da abordagem proposta.

- No Capítulo 5 são apresentadas as conclusões e considerações finais deste trabalho, assim como propostas de trabalhos futuros.

CAPÍTULO

# 2

# *Fundamentação Teórica*

Neste capítulo serão apresentados os resultados de um levantamento bibliográfico da literatura que foram utilizados como embasamento para a definição da abordagem proposta por este trabalho. Inicialmente, foi definido a classificação de técnicas de análise de impacto. Posteriormente, foi estabelecido um conjunto de técnicas de análise de impacto de mudanças em que os elementos impactados são elementos do código-fonte, utilizando principalmente os resultados de uma revisão bibliográfica encontrada na literatura (Li et al., 2013). Após isto, foi definido o funcionamento da técnica de análise de Hattori, métricas de manitenibilidade e técnicas de visualização, que foram utilizadas como objeto de estudo para este trabalho.

E por fim foram apresentados trabalhos relacionados e suas limitações, em relação a processos de análise de impacto e ferramentas de análise de impacto que utilizam visualização de software para identificação dos possíveis impactados.

## 2.1 Classificação de uma Análise de Impacto de Mudanças

Para aumentar a eficiência da análise de impacto e classificação de técnicas de análise de impacto, Bohner e Arnold (1996) propôs classes de análises de impacto de acordo com dois tipos de perspectivas: Quanto à perspectiva da análise e quanto à abordagem utilizada. Tais abordagens são definidas a seguir.

### 2.1.1 Classificação quanto à perspectiva da análise

Quanto a perspectiva de análise, duas classes são definidas: *Análise de Dependência* e *Análise de Rastreabilidade* (Bohner e Arnold, 1996). Tais classes estão representadas na Figura 2.1 (Hattori, 2008).

Ambas possuem vantagens que potencializam a identificação dos impactos no software.

**Análise de Rastreabilidade:** Este tipo de análise explora as relações entre as dependências entre artefatos de um sistema, podendo relacionar requisitos com diagramas produzidos

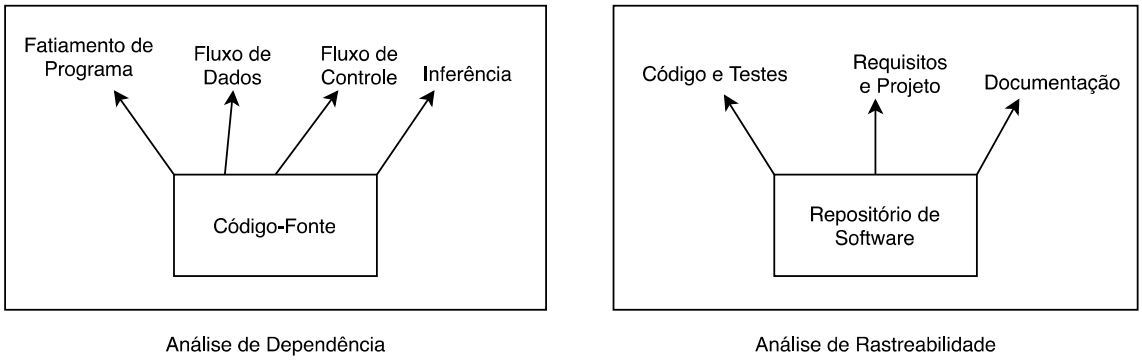


Figura 2.1: Classes de Perspectivas de Análises de Impacto. Adaptado: (Hattori, 2008)

na etapa de Projeto no processo de desenvolvimento do software, por exemplo (Hattori, 2008). A criação de rastreabilidade é a tarefa de associar artefatos por meio de *links* de rastreamento. Rastreamento por captura é um tipo de rastreamento que envolve a criação de *links* de rastreamento para um artefato no momento que este é criado e concomitantemente com os artefatos associados a ele. Já rastreamento por recuperação os *links* são criados após a criação dos artefatos, as associações são buscadas e processadas para criar os *links*. Técnicas de Recuperação de Informação podem ser aplicadas para auxiliar esse tipo de análise, como associar artefatos com conteúdo textual altamente similar (Borg et al., 2017).

**Análise de Dependência:** Este tipo análise explora as relações de dependência entre controle, dados e componentes de um programa, tendo como alvo as relações de dependência entre entidades de baixo nível no código, mas não relacionando outros artefatos do processo de desenvolvimento do software (como diagramas, documento de requisitos, relatório de defeitos, casos de teste, etc). É comumente referenciada como análise de código fonte (Bohner e Arnold, 1996). Em uma revisão da literatura é confirmado que a maioria das pesquisas sobre AI está limitada a essa perspectiva, sendo abordado por 65% dos trabalhos listados (Lehnert, 2011). Mapeando para objetos, os dados podem ser representados por atributos e variáveis, o controle examina todas as possibilidades de fluxos em cada método das classes e os componentes podem ser representados por módulos de funcionalidades do próprio software. Alguns tipos de técnicas de análise de dependência encontradas na literatura são: Análise de fluxo de dados (*data flow analysis*), análise de fluxo de controle (*control flow analysis*), análise baseada em grafos de chamadas, fatiamento de programa (*program slicing*), análise de testes de cobertura e referência cruzada (Hattori, 2008).

### 2.1.2 Classificação quanto à abordagem utilizada

Outra perpectiva para a classificação de uma análise de impacto é quanto à abordagem utilizada. Neste caso, há também duas classes: Estática e Dinâmica (Borg et al., 2017). Tais classes são definidas a seguir.

**Análise Estática:** Neste tipo de análise o código fonte do software é aferido para identificação dos possíveis impactos causados por uma mudança proposta. Ela é utilizada para estimar esforços necessários para implementação, pois é aplicada usualmente antes da implementação

da mudança, e permite ao avaliador determinar qual a melhor opção de mudança dentre as demais opções de acordo com o conjunto possivelmente impactado (Hattori, 2008). Neste tipo de análise, o software afetado pela mudança é mapeado para um conjunto de grafos, em que os nós representam as entidades e as arestas as dependências entre estas entidades. A construção desses grafos pode ser feita em níveis diferentes de granularidade. Por exemplo, em nível de método ou de sentenças, em que se utilizam técnicas de fatiamento de programa(do inglês *program slicing*) para mapear o software para grafos de fluxo de controle e fluxo de dados, para incluir todas as dependências de uma mudança em nível de sentenças. No entanto esse tipo de análise, em nível de sentença, tende a gerar um conjunto de falso-positivos demasiadamente grande, pois são considerados todos os parâmetros de entrada e todos os possíveis comportamentos do sistema, inclusive aqueles impossíveis de acontecer (Rolfsnes et al., 2016).

**Análise Dinâmica:** Neste tipo de análise, rastros de execução de um programa são utilizados para identificar elementos impactados por uma mudança. As análises dinâmicas podem ser categorizadas em *online* e *offline*. Há diversas semelhanças entre as duas, principalmente relacionadas à maneira de análise dos impactos, dos dados coletados e resultados encontrados. No entanto, a principal característica encontrada é a diferença em como os dados dos rastros são analisados, após a execução do programa e durante a execução do programa na *offline* e *online*, respectivamente (Orso et al., 2004). Um elemento é classificado como impactado quando é afetado pela mudança ao menos em uma das execuções do programa analisado. Esse tipo de análise é estritamente dependente do conjunto de execuções, da qualidade dos testes e da entrada de cada execução para capturar rastros de todas as execuções e garantir que o conjunto real de impactados seja identificado. Como não é possível assegurar esta identificação, este tipo de técnica gera falsos-negativo. No entanto, a quantidade de falsos-negativo gerados pela análise dinâmica é bastante reduzida em relação à estática, sendo a primeira considerada mais precisa (Apiwattanapong et al., 2005). As técnicas de análise de impacto dinâmicas mais precisas usam eventos de métodos (entrada e retorno) para calcular a ordem de execução de métodos e, em seguida, o conjunto de impacto dinâmico é produzido incluindo os métodos alterados, bem como todos os métodos que são executados após o(s) método(s) alterado(s) (Huang e Song, 2007).

Como este trabalho de Mestrado teve como foco o uso de técnicas de análise de impacto em código-fonte, a classe que foi considerada quanto à perspectiva da análise é a de dependência pois não considera a identificação de impactos em outros artefatos a não ser o código-fonte. Então, é apresentado a seguir, técnicas de análise de impacto em código-fonte.

## 2.2 Técnicas de Análise de Impacto em Código-fonte

As técnicas da AI baseadas em código-fonte têm como foco a identificação de impactos ocasionados por uma mudança. Por ter o foco restrito a código-fonte, elas podem apresentar melhor precisão dos resultados, pois analisam diretamente os detalhes da implementação (Czibula et al., 2019). Os esforços da AI se concentram no nível do código-fonte e, consequente-

mete, tem havido uma evolução considerável dessas técnicas (SI et al., 2019). Neste trabalho se porpõe utilizar esse tipo de técnica como objeto de estudo, para tentar evoluir e aprimorar sua aplicação.

Consequentemente, uma revisão sistemática da literatura foi conduzida, tendo sido encontradas 23 técnicas de análise de impacto com foco em alterações de código-fonte, todas para a definição do conjunto de impactados. As técnicas encontradas são listadas na Tabela 2.1 (Li et al., 2013), juntamente com uma breve descrição.

Durante o levantamento, foi identificada a classificção das técnicas de AI baseadas em código-fonte. São quatro classes: *análise de dependência tradicional*, *mineração de repositórios de software*, *baseadas em métricas de acoplamento* e *coleta de informações de execução* (Li et al., 2013). Essas classes auxiliaram na escolha da técnica utilizada para a prova de conceito da abordagem proposta, pois definem as informações iniciais necessárias para análise de impacto. Como a abordagem utiliza análise estática do código-fonte, foi escolhida uma técnica estática de análise de dependência tradicional chamada Técnica de Hattori, que é definida a seguir.

## 2.3 Técnica de Análise de Impacto de Hattori

Essa técnica, proposta por Hattori (2008), consiste na análise estática do código-fonte de um sistema orientado a objeto, extraindo relações de dependência entre os elementos e a navegação por estas relações, a partir de uma mudança proposta, identificando possíveis impactados. Os dados iniciais para a execução da técnica de Harttori consitem em: uma especificação das mudanças e o código-fonte do sistema analisado. Essa técnica tem como retorno um conjunto de elementos impactadas. Os elementos impactados considerados pela técnicas são: classe, método e atributo. A técnica seleciona o algoritmo de busca de impactados de acordo com o tipo de mudança proposto (Hattori, 2008).

Os tipos de mudanças utilizados pela técnica de Hattori têm como referência a classificação das possíveis dependências entre as entidades de um sistema orientado a objetos proposta por Lee et al. (1998). As dependências foram classificadas em cinco categorias, listadas a seguir.

1. Dependência classe para classe:
   (a) C1 é superclasse direta de C2 (C2 herda de C1);
   (b) C1 é subclasse direta de C2 (C1 herda de C2);
   (c) C1 é classe ancestral de C2 (C2 herda indiretamente de C1);
   (d) C1 usa C2 (C1 referencia C2, inclui referência direta e indireta);
   (e) C1 contém C2.
2. Dependência classe para método:
   (a) método M retorna objeto da classe C;
   (b) C implementa método M.

| **Técnica** | **Definição** |
|---|---|
| T1 | Utiliza métrica de acoplamento lógico para sistemas orientado a objetos para identificar o conjunto de impacto. |
| T2 | Use as informações de dados coletados de usuários como suporte a AI. |
| T3 | Fornece uma técnica para AI com base em um perfil de execução completo. |
| T4 | Aplica a mineração de dados a repositórios de históricos de versão para extrair o acoplamento de co-mudança entre os arquivos. |
| T5 | Usa o conjunto execute-after de entidades como suporte a AI. |
| T6 | Usa o grafo de chamadas de controle para executar a AI. |
| T7 | Usa programação dinâmica em traços instrumentados de sistemas diferentes para calcular o conjunto de impacto. |
| T8 | Analisa os mecanismos de influência de escopo, assinaturas de funções e acessos à variáveis globais como suporte a AI. |
| T9 | Usa a similaridade textual para recuperar a solicitação de mudança anterior contidas no repositório do software analisado. |
| T10 | Executar análise de dependência na execução de sistemas orientado a objetos como suporte a AI. |
| T11 | Usa a métrica de acoplamento de função dinâmica entre duas funções como suporte a AI. |
| T12 | Cria agrupamentos de arquivos de sistemas de software intimamente associados contidos no repositório de software de AI. |
| T13 | Aplica dois algoritmos de mineração de dados diferentes Apriori e DAR no repositório de software como suporte a AI. |
| T14 | Analisa registros de mudança através da decomposição de valores singulares para produzir arquivos de co-mudança como suporte a AI. |
| T15 | Usa o gráfico de chamadas para calcular o conjunto de impacto. |
| T16 | Usa a medição de acoplamento conceitual como suporte a AI. |
| T17 | Usa um modelo hierárquico para calcular interativamente o conjunto de impacto. |
| T18 | Mistura acoplamentos conceituais e evolutivos para apoiar a AI. |
| T19 | Use comentários de código-fonte e informações de mudanças contidas no repositório de software como suporte a AI. |
| T20 | Usa análise multivariada de séries temporais e regras de associação para realizar a AI. |
| T21 | Analisa mecanismos de impacto de diferentes tipos de mudança como suporte a AI. |
| T22 | Usa o acoplamento baseado em tópicos relacionais para capturar tópicos em classes e relacionamentos entre eles para realizar AI. |
| T23 | Usa classificadores baseados em aprendizado de máquina como suporte a AI. |

Tabela 2.1: Técnicas de AI em código-fonte - Adaptado de (Li et al., 2013).

3. Dependência classe para variável:
   (a) variável V é uma instância da classe C;
   (b) V é uma variável da classe C;
   (c) V é definida pela classe C.
4. Dependência método para variável:
   (a) V é parâmetro do método M;
   (b) V é variável local de M;
   (c) V é importada por M (por ex., é variável não-local usada por M);
   (d) V é definida por M.
5. Dependência método para método:
   (a) método M1 invoca método M2;
   (b) M1 sobrescreve M2.

O tipos gerais de mudanças abordados são: adição, remoção e alteração de um elemento. Para cada tipo de elemento foram classificados tipos de mudanças mais específicos no intuito de mapear todos os possíveis tipos de dependência entre eles, conforme pode ser observado na Figura 2.2.

Utilizando estas definições, é apresentado a seguir como a técnica de Hattori determina as relações existentes entre os elementos de um software, a partir de uma análise e extração de informação de um código-fonte.

### 2.3.1 Representação do Grafo de Relações entre Elementos

A técnica de Hattori (2008) propõe que a partir do código-fonte de uma aplicação um modelo de representação do software, composto por conjuntos de elementos e relações, pode ser gerado. Os elementos considerados são: classe, método e atributo. Já uma relação conecta dois elementos e representa uma conexão de dependência que o elemento chamador possui com o chamado. As relações são classificadas de acordo com as dependências vistas anteriormente. Na Figura 2.3 podem ser vistos todos os tipos de relações abordadas, combinando os elementos em pares. As relações inversas estão representadas na cor vermelha na Figura 2.3. Essa redundância está presente a fim de facilitar a navegabilidade do *design* no decorrer do processo de análise de impacto.

O modelo criado pode ser representado por uma estrutura de grafo, em que os vértices representam os elementos e as arestas representam as relações entre os elementos. Assim, a técnica de Hattori propõe algoritmos para busca por elementos impactados, percorrendo o grafo pelas relações de interesse (Hattori, 2008).

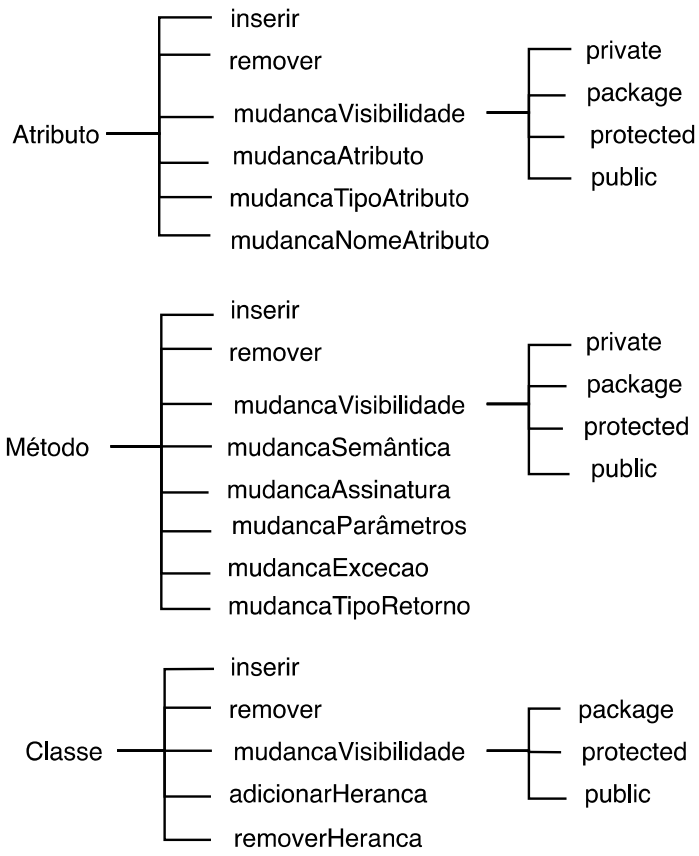


Figura 2.2: Classificação dos tipos de mudanças de acordo com cada tipo de elemento. Adaptado: (Hattori, 2008)

### 2.3.2 Busca de Impactos de Mudança

Ao receber o conjunto de mudanças a serem analisadas e o modelo de elementos e relações que representam o sistema, a próxima etapa da técnica de Hattori é a busca dos impactos para cada mudança, individualmente.

Para cada mudança, um algoritmo diferente é executado e cria uma árvore de elementos impactados, na qual a raiz é a própria mudança e as folhas e os nós intermediários são elementos chamadores. O princípio de busca dos algoritmos de análise de impacto é percorrer o grafo do sistema buscando por elementos chamadores. O algoritmo escolhido é baseado no tipo de mudança proposta, de acordo com os tipos de mudanças vistas na Figura 2.2.

No entanto, alguns pontos devem ser observados com relação ao elemento e suas respctivas relações em um programa de paradigma orientado a objeto, como polimorfismo e herança, apresentados e discutidos a seguir.

### 2.3.3 Polimorfismo e Herança

O polimorfismo é uma das características de linguagens orientada a objetos que dificulta a aplicação da técnica de análise estática do código, pois permite que uma chamada de método seja executada de várias formas diferentes. Esse é o principal causador de falso-positivos na

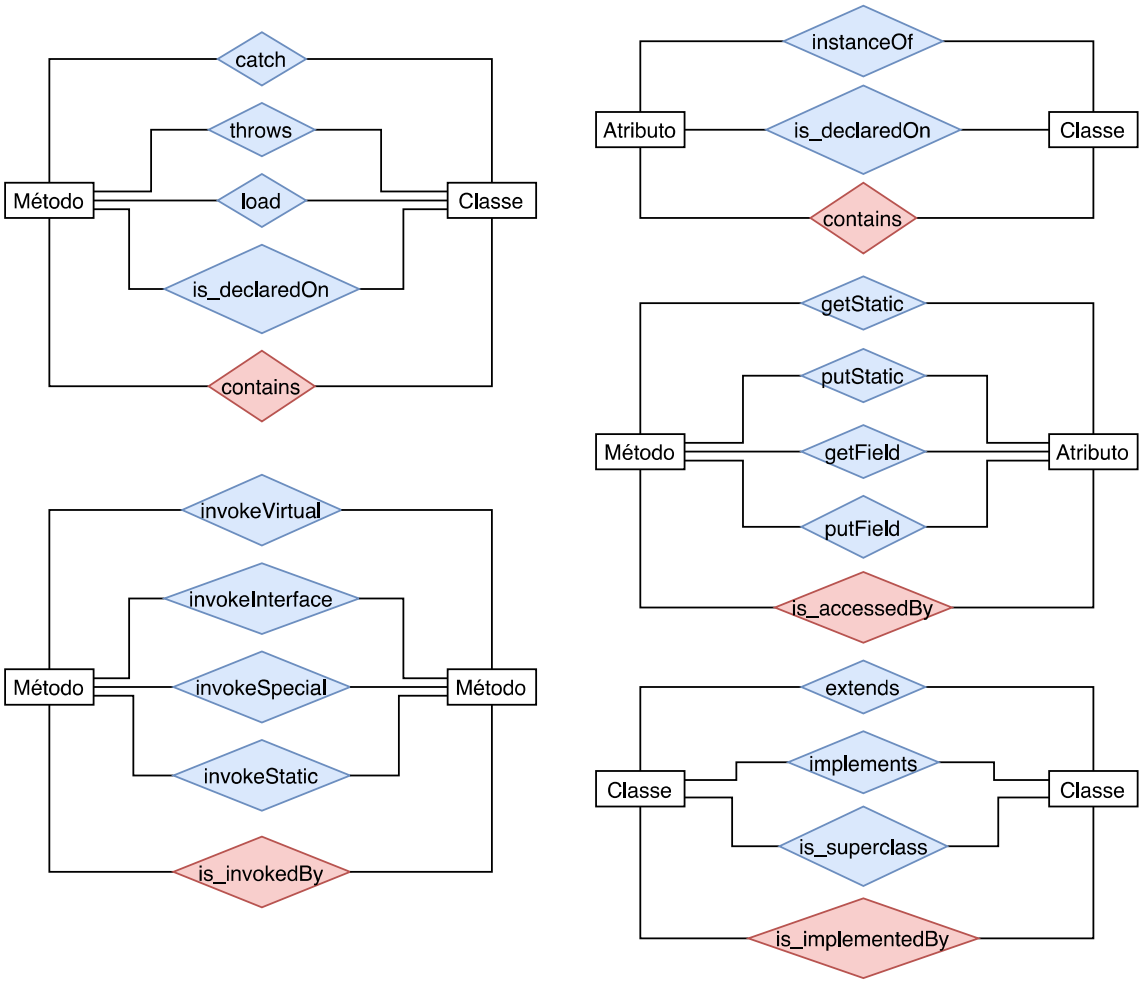


Figura 2.3: Diagrama de elementos e suas possíveis relações. Fonte: (Hattori, 2008)

análise de impacto estática de Hattori. Seu uso torna impossível de determinar qual subtipo usado para executar um método de um super-tipo apenas com o *bytecode* (Hattori, 2008). Essa informação de chamada de método só pode ser obtida em tempo de execução. Portanto, todos os subtipos serão incluídos no conjunto de possíveis impactados da análise de impacto.

Já a herança é também uma característica que dificulta a aplicação da técnica de Hattori. As relações hierárquicas, que podem ser vistas na Tabela 2.2, são analisadas sempre que um subtipo ou um supertipo for impactado.

| **Subtipo** | **Relação** | **Supertipo** |
|---|---|---|
| interface | estende | interface |
| classe abstrata | implementa | interface |
| classe | implementa | interface |
| classe abstrata | estende | classe abstrata |
| classe | estende | classe abstrata |
| classe | estende | classe |

Tabela 2.2: Tipos de relações de herança presentes na linguagem Java Fonte: (Hattori, 2008).

É considerado que uma mudança em qualquer nível da hierarquia de herança impacta todos

os outros elementos contidos nessa hierarquia. Portanto, se o método de um super-tipo é modificado, os métodos equivalentes (herdados ou re-implementados) dos seus subtipos devem ser incluídos no conjunto possíveis impactados. E se o método de um subtipo é modificado, o método de seu super-tipo equivalente (re-implementado no subtipo) deve ser incluido no conjunto de possíveis impactados.

### 2.3.4 Algoritmos

Para a seleção dos elementos impactados a partir da busca no grafo de relações entre elementos (visto na Seção 2.3.1), a técnica de Hattori estabelece quatro algoritmos: *RemoveInheritance*, *AddInheritance*, *ChangeVisibility* e *Modification* (Hattori, 2008). As principais características de cada algoritmo estão descritas a seguir.

- *RemoveInheritance*: Este algoritmo trata mudanças de elementos que envolvam a remoção da herança de uma classe. Este tipo de remoção a uma subclasse, impacta diretamente todos os métodos da superclasse que foram herdados pela subclasse. Se existe alguma chamada do tipo super, em que um método da subclasse chama, explicitamente, uma entidade da superclasse, esse método chamador entrará no conjunto de impactos. Se em um método da subclasse, houver uma chamada a um método da superclasse que não foi redefinido na subclasse, esse método chamador também será adicionado ao conjunto de impactos.

- *AddInheritance*: Este algoritmo trata impactos gerados pela adição de herança a uma classe. Se a subclasse já estender de outra superclasse, esta deverá ser removida. Caso a herança a ser adicionada seja a um supertipo ou a uma superclasse abstrata, então os métodos do supertipo ou os métodos abstratos da superclasse devem ser implementados na subclasse. Portanto, neste caso, todas as subclasses que estendem da superclasse serão inseridas no conjunto de impactados.

- *ChangeVisibility*: Este algoritmo trata as mudanças em elementos que envolvam aumento ou redução de visibilidade de um elemento. A mudança pode envolver aumento ou redução de visibilidade. Em Java, as palavras reservadas *Private*, *Package*, *Protected* e *Public* são modificadores de acesso utilizados para mudar a visibilidade de um elemento. Um elemento *Public* possui a maior visibilidade, sendo visível a qualquer outro elemento. Já um elemento *Private* possui menor visibilidade possível. Quando há uma redução de visibilidade, vários impactos podem ser causados, pois os elementos chamadores podem perder o acesso ao elemento modificado. Para este tipo de visibilidade é preciso verificar as relações de dependência de cada membro da hierarquia de herança que for superclasse ou supertipo do elemento modificado. Um *array* é criado contendo o elemento em questão e todas as suas superclasses, no qual será percorrido. Para cada classe analisada, há uma verificação dos métodos que a chamam de acordo com as características específicas para cada visibilidade restante. Se houver mudança para o tipo *Public*, então não há impactos. Se houver mudança para o tipo *Package* e se o método chamador não está no

mesmo pacote da classe então ela é uma entidade impactada. Se houver mudança para o tipo *Protected*, além ser feita a verificação do tipo anterior, se a classe à qual o método chamador pertence não for uma subclasse da classe iterada, então ela é impactada. Se houver mudança para tipo *Private*, se a classe à qual o método chamador pertence não for a própria classe iterada, então ela é impactada.

- *Modification*: Este algoritmo trata as demais mudanças em elementos. Para operações de adição (*add*) não há verificação pois foi considerado todas as mudanças como atômicas. Ele recebe como parâmetro o elemento modificado e busca todos os métodos que a acessam, de acordo com seu tipo. Se o elemento modificado for um método, ele busca no conjunto de relações os do tipo *is_invokedBy* e retorna os métodos chamadores. No entanto, se o elemento modificado for um atributo, então ele busca por relações do tipo *is_accessedBy* e retorna os métodos chamadores. E por fim, se o elemento for uma classe, ele busca por relações do tipo *is_invokedBy* e retorna os métodos chamadores. Estes métodos retornados são adicionados ao conjunto de impactos.

A Técnica de Hattori continua a execução dos algoritmos de forma recursiva, até que não sejam mais identificados impactados e toda a pilha de execução seja encerrada. Após o fim da execução, um conjunto de elementos do código-fonte possívelmente impactados é retornado.

## 2.4 Métricas de Manutenibilidade

Dentre os objetivos de uma análise de impacto de mudanças, um deles é a estimativa da quantidade de esforço envolvido na implementação de uma mudança. Para auxiliar nesta estimativa várias métricas de manutenção de software foram desenvolvidas, sendo a maioria delas, envolvendo análise automatizada de código-fonte (Chen et al., 2017). A seguir veremos as mais utilizadas na indústria de software como estimativa de esforço de manutenção.

### 2.4.1 Linhas de Código (LOC - *Lines of Code*)

Métrica utilizada para medir complexidade e esforço de manutenção que consiste na contagem de linhas de código-fonte de um software analisado (Velarde et al., 2016). Esta métrica pode ser aplicada também em funções, módulos, métodos e ou classes individuais contidos em um programa. Esta métrica não reflete sozinha a complexidade com exatidão, pois é dependente da linguagem de programação. Ou seja, linguagens altamente verbosas possuem desvantagem em relação a não-verbosas se analisarmos dois trechos de código semânticamente idênticos (Ochodek et al., 2020). Esta métrica possui a característica de possuir simplicidade de implementação e entendimento.

Além disso, outras métricas podem ser derivadas da contagem de linhas, que podem ser vistas a seguir:

- Linhas de código funcional (*LOCpro*): Métrica que representa linhas de código que possuem trechos de programação;

- Linhas de comentários (*LOCcom*): Métrica que representa linhas de código que possuem apenas comentários;
- Linhas em branco (*LOCbl*): Métrica que representa linhas em branco.

### 2.4.2 Complexidade Ciclomática

Métrica proposta por McCabe (1976), que consite na medida do número de caminhos linearmente independentes (caminhos únicos) extraídos do código-fonte. A complexidade ciclomática utiliza um grafo de fluxo de controle do software analisado para determinar a quantidade de casos de teste unitário mínimo necessário.

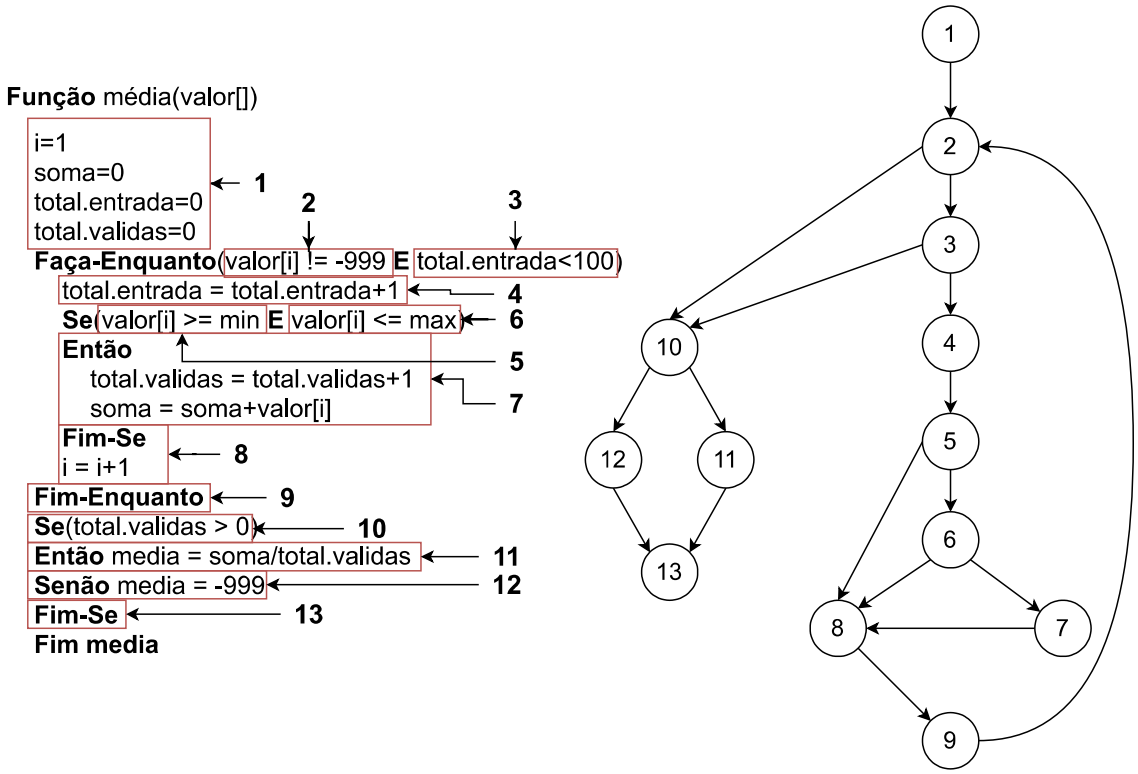


Figura 2.4: Exemplo de função e seu grafo de fluxo de controle. Fonte: Própria

Os nós do grafo correspondem a grupos indivisíveis de comandos de um código-fonte de um programa e as arestas direcionadas conectam dois grupos se um segundo grupo pode ser executado após um primeiro grupo. A presença de estuturas condicionais e estruturas de repetição no código-fonte analisado aumentam a complexidade, porque propicia caminhos extras para cada tomada de decisão em cada estutura definida. Além de aplicada ao sistema como um todo, a complexidade ciclomática pode ser aplicada em funções, módulos, métodos ou classes individuais contidos em um programa (Junior et al., 2016).

A complexidade ciclomática (M) pode ser calculada com a fórmula:

$$M = E - N + 2 \tag{2.1}$$

sendo $E$ = número de arestas do grafo e $N$ = número de nós do grafo.

Na Figura 2.4, podemos observar um exemplo de uma função de calculo de média e o grafo de fluxo de controle que a representa. Utilizando a fórmula de cálculo da complexidade apresentada temos:

$$M = 17 - 13 + 2 = 6 \tag{2.2}$$

A complexidade pode ser avaliada seguindo os seguintes parâmetros:

1. *M* entre 1 e 10: Complexidade baixa e programa fácil de entender e manter;
2. *M* entre 11 e 20: Complexidade moderada e programa razoávelmente complexo de entender e manter;
3. *M* maior que 20: Complexidade alta e programa impossível de entender e manter (altamente recomendado reescrever o trecho analisado)

Utilizando a avaliação da métrica com o exemplo da Figura 2.4, a função de média é de baixa complexidade e de fácil manutenção.

### 2.4.3 Métricas de Halstead

Métricas propostas por Halstead et al. (1977) que consitem no mapeamento do código-fonte como uma sequência de tokens e na classificação de cada token como operador ou operando. Estas métricas foram extensivamente testadas como medida de complexidade (Coimbra et al., 2018), sendo consideradas como forte indicadores de complexidade de código-fonte e frequentimente utilizadas para mensurar a dificuldade de manutenção do trecho analisado (Hariprasad et al., 2017).

As métricas de Halstead são baseadas em métricas extraídas inicialmente do código-fonte:

- $n_1$ = número de operadores únicos (distintos).
- $n_2$ = número de operandos únicos (distintos).
- $N_1$ = número total de operadores.
- $N_2$ = número total de operandos.

Utilizando essas métricas iniciais, as métricas de Halstead podem ser vistas a seguir.

O Tamanho (*Program length* ou *N*) é a soma do número total de operadores e operandos no programa. Ele é representado por:

$$N = N_1 + N_2 \tag{2.3}$$

O Tamanho do vocabulário (*Vocabulary size* ou *n*) é a soma do número de operadores e operandos únicos. Ele pode ser representado por:

$$n = n_1 + n_2 \tag{2.4}$$

O Volume (*Program volume* ou *V*) é a relação do tamanho de implementação do trecho analisado. Ele pode ser representado por:

$$V = N * \log_2(n) \tag{2.5}$$

A Dificuldade de implementação (*Difficulty level* ou *D*) é a propensão a erros do código analisado, ou seja, se os mesmos operandos forem usados várias vezes no programa, é mais provável que haja erros. Ela está relacionada a quantidade de operadores e operandos únicos e pode ser representada por:

$$D = (n_1/2) * (N_2/n_2) \tag{2.6}$$

O Nível de Implementação (*Program level* ou *L*) é o inverso da propensão a erros do programa, ou seja, quanto maior a dificuldade menor é o nível de implementação. Ele pode ser representador por:

$$L = 1/D \tag{2.7}$$

O Esforço de implementação (*Effort to implement* ou *E*) é o esforço para implementar ou entender um programa. Ele pode ser representado por:

$$E = V * D \tag{2.8}$$

O Tempo de Implementação (*Time to implement* ou *T*) é o tempo necessário por um programador implementar o trecho analisado. Baseado em um estudo de psicologia do pensamento, esta métrica se utiliza do esforço de implementação do código analisado para retornar uma medida de tempo em segundos> Ele pode ser presentado por:

$$T = E/18 \tag{2.9}$$

Número de possíveis erros (*Number of delivered bugs* ou *B*) é a estimativa do número de erros na implementação no trecho analisado. Ele pode ser representado por:

$$B = E^{\frac{2}{3}}/3000 \tag{2.10}$$

### 2.4.4 Índice de Manutenibilidade

Também chamada de *Maintainability Index* ou simplesmente MI, esta métrica polinomial proposta por Welker et al. (1997) utiliza complexidade ciclomática, a métrica de volume de Halstead, linhas de código e porcentagem de comentários para determinar se o código-fonte tem um alto, médio e baixo grau de dificuldade de manutenção. Essa categorização é importante para o nosso trabalho, pois é utilizado como base para priorização em uma das perspectivas geradas em nossa abordagem.

Esta métrica, que foi validada em módulos de sistemas da *Hewlett-Packard* (Welker et al., 1997), tem a vantagem em relação as demais métricas vistas anteriormente de ser indepen-

dente da linguagem do módulo analisado, podendo ser obtido uma análise mais próxima da realidade (Strečanskỳ et al., 2020). O MI pode ser definido por:

$$MI = 171 - 5.2 * \log_2 V - 0.23 * G - 16.2 * \log_2 LOC + 50 * \sin\sqrt{2.4 * CM} \quad (2.11)$$

sendo,

- $V$ o volume de Halstead;
- $G$ a complexidade ciclomática;
- $LOC$ número de linhas que possuem implementação (ou LOCpro);
- $CM$ o percentual de linhas de comentário em relação ao total de linhas do trecho analisado.

O MI pode ser avaliado para um trecho analisado seguindo os seguintes parâmetros (Seref e Tanriover, 2016):

- Caso $MI < 65$ o trecho é de difícil manutenção, com trechos de código muito ruins (grandes, não comentados, não estruturados), sendo preferível reescrever o módulo. Os valores de MI podem ser negativos;
- Caso $65 \leq MI < 85$ o trecho é de manutenibilidade moderada;
- Caso $MI \geq 85$ o trecho é de boa manutenibilidade, ou seja, o analista não terá dificuldade de compreendê-lo e alterá-lo

## 2.5 Técnicas de Visualização de Software

Para auxiliar a compreensão de sistemas complexos de software e melhorar a produtividade do processo de desenvolvimento, a Visualização de Software se torna necessária (Merino et al., 2018). Ela traz benefícios como: economia de tempo e dinheiro e melhoria na análise de código-fonte, sendo estes fatores muito importantes para a análise de impacto de mudanças (Martins, 2012). A seguir, definimos propriedades de duas técnicas de visualização que podem ser utilizadas para visualização de software e que foram utilizadas em nossa abordagem.

### 2.5.1 *Treemap*

A *Treemap* é uma visualização na qual a estrutura hierárquica é visualmente mapeada em retângulos, sendo o nó da hierarquia mais alta representado por um retângulo mais externo e de maior área que é recursivamente dividido em retângulos internos (menores), cada um representando um nó filho dessa hierarquia (Shneiderman e Wattenberg, 2001).

Nesta visualização o tamanho dos retângulos são proporcionais ao valor dos dados que os representa, levando em consideração também os elementos que estão contidos entre si, respeitando a estrutura hierárquica representada. Vários outros atributos podem ser mapeados: cor

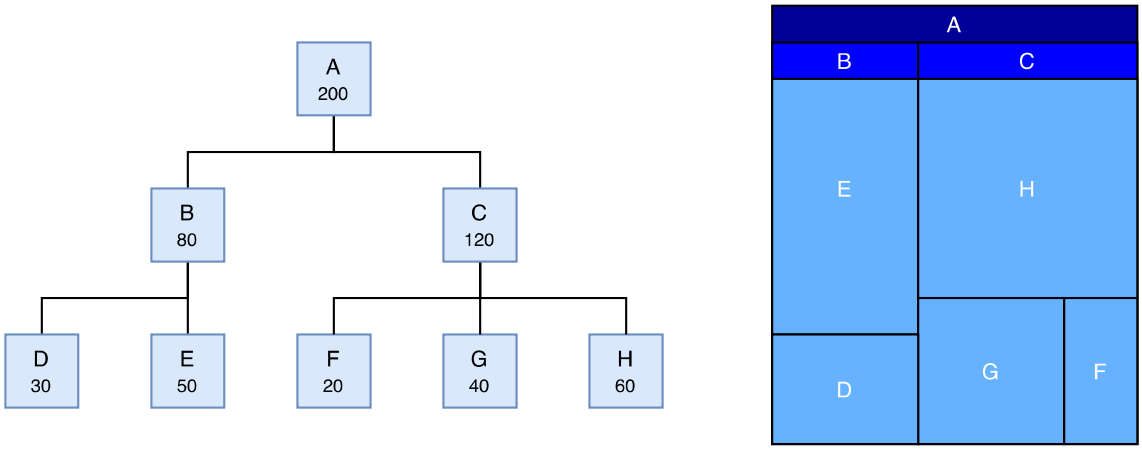


Figura 2.5: Exemplo de estrutura hieráquica e a *treemap* que a representa. Fonte: Própria

representando nível hieráquico, ou cor representando uma característica específica ou variações de opacidade de cores e cores das bordas dos retângulos (Scheibel et al., 2020).

Podemos observar na Figura 2.5 uma representação de estrutura hieráquica e a *Treemap* correspondente. Neste exemplo, os atributos visuais que foram utilizados para a representação são:

- O tamanho dos retângulos é proporcional ao peso de cada nó da estrutura hieráquica;
- Diferentes opacidades da mesma cor foram utilizadas para diferenciar os níveis da estrutura, ou seja, nós do mesmo nível hieráquico possuem a mesma opacidade.

### 2.5.2 *Force-directed graph*

O *Force-directed graph* é uma visualização utilizada para mostrar relacionamentos entre elementos, onde são mapeados em um grafo, no qual os vértices representam os elementos e as arestas representam os relacionamentos (Gove, 2019). Os vértices de um grafo são posicionados em um espaço bidimensional ou tridimensional de maneira que todas as suas arestas tenham comprimento semelhante (Walshaw, 2000).

Utilizando atribuições de forças nos posicionamentos relativos das arestas e vértices, é possível posicioná-los de maneiras de acordo com o tipo e intensidade de força aplicada – por exemplo, ao aplicar uma força de repulsão, os vértices se reposicionam afastando-se uns dos outros. Há um agrupamento natural dos objetos que estão bem conectados, podendo ser visualmente interessantes para ajudar a descobrir relacionamentos entre grupos que, de outra forma, não seriam óbvio (Haleem et al., 2019). Comulmente, os nós se organizam em função de aplicação de uma força de repulsão entre eles, alterando as coordenadas de localização de cada nó seguindo uma função definida. Outro atributos visuais podem ser explorados, por exemplo cores e diferentes opacidades, formas e tamanhos dos vértices, largura da aresta.

Podemos observar na Figura 2.6 a representação de um *Force-directed graph* que representa as chamadas de métodos contidos em dois pacotes de um software. Os vértices representam os métodos das classes contidas nos pacotes e as arestas representam as chamadas entre eles.

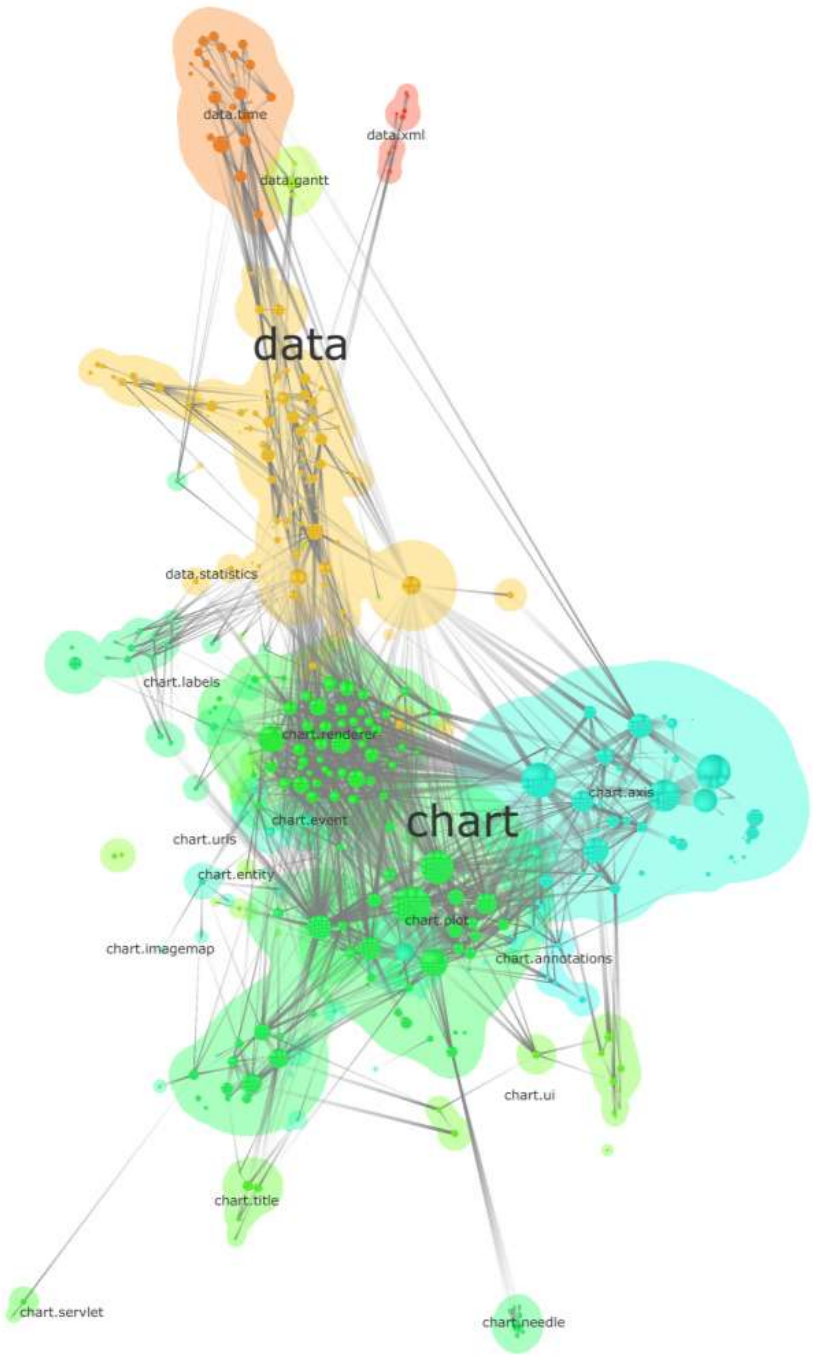


Figura 2.6: Exemplo de *Force-directed graph*. Fonte: (Balzer e Deussen, 2005)

Para a abordagem proposta, foi utilizada uma variante do *Force-directed graph*, chamada de *Force-directed tree*, em que uma estrutura hieráquica com pacotes, classes, métodos e relacionamentos hierárquicos entre eles foram mapeados em vértices e arestas respectivamente (Arleo et al., 2018). Nesta variação, os elementos que estão próximos hieraquicamente na estrutura, tendem a se posicionar mais próximos na visualização, como por exemplo, métodos pertencentes a uma classe se posicionarão mais próximo dessa classe na visualização em relação a distância com as demais classes. Em resumo, ainda é um *Force-directed graph*, mas com regras

de posicionamento dos elementos específicas para representação de estruturas hieráquicas.

É possível observar na Figura 2.7 uma representação de um pacote de um software com suas classes e métodos contidos nas classes. Neste exemplo, os vértices que representam as classes possuem o formato de círculo e os vértices que representam os métodos possuem o formato de quadrado. Ao observar a Figura 2.7, constatamos que métodos que pertencem hieráquicamente a uma classe orbitam em volta do vétice que a representa e estão próximos dela que em relação a outras classes.

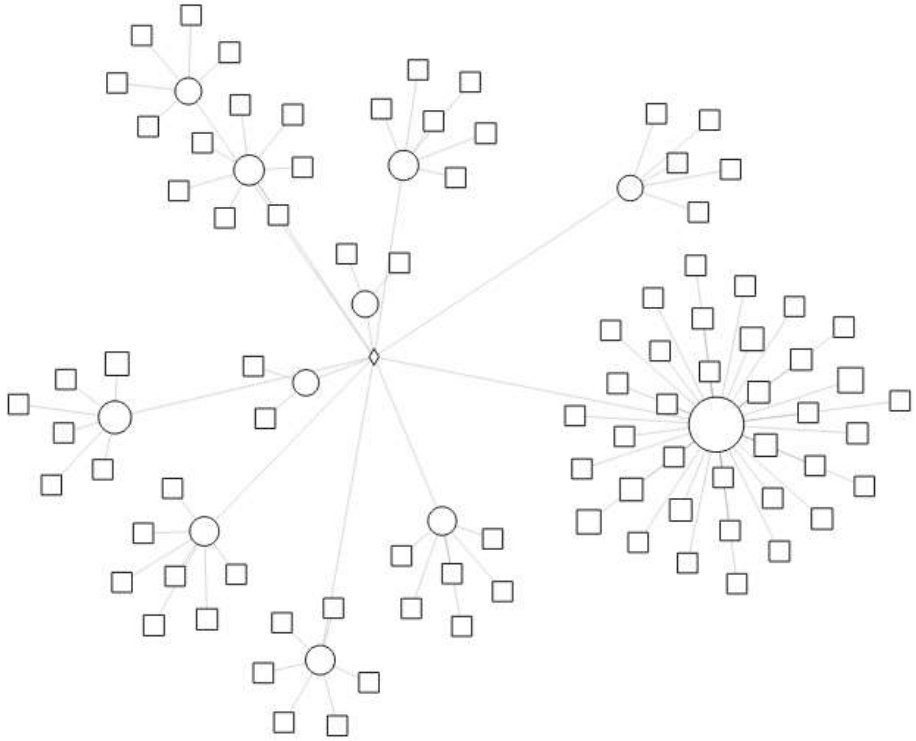

Figura 2.7: Exemplo de *Force-directed tree* que representa um pacote de um software. Fonte: Própria

## 2.6 Trabalhos Relacionados

Como o trabalho propôs a elaboração de uma abordagem para auxiliar o processo de análise de impacto, fez-se necessário a análise de trabalhos relacionados a processos de uma AI e definição de regras e/ou estratégias que auxiliem a execução de uma mudança utilizando os resultado de uma AI.

Na Figura 2.8 é descrito um processo elementar de AI, proposto por Bohner (2002a). Sua organização é evolutiva: a primeira etapa consiste em examinar a especificação da mudança e do software para identificar os artefatos afetados pela mudança; o *Conjunto de Impacto Inicial* (CII) representa o conjunto de artefatos possivelmente afetados pela mudança proposta.

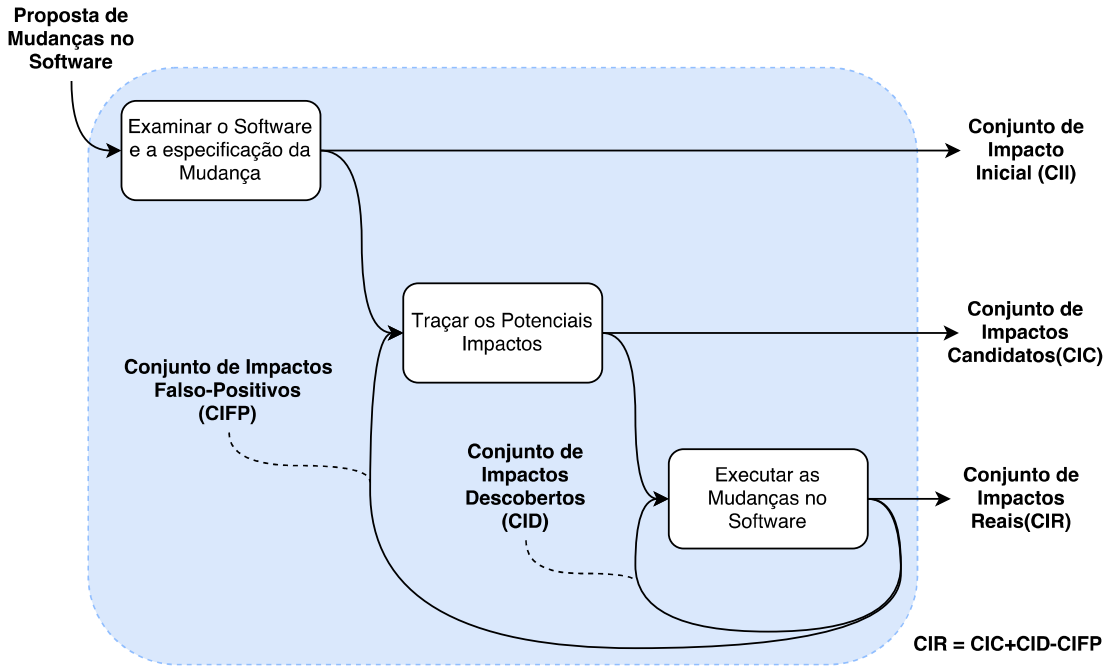


Figura 2.8: Modelo de Análise de Impacto de Mudanças em Software. Adaptado: (Bohner, 2002a)

Em seguida, é realizada a análise propriamente dita e é produzido o *Conjunto de Impactos Candidatos* (CIC), contendo os artefatos afetados pela mudança. Após a implementação da mudança, é produzido o *Conjunto de Impactos Reais* (CIR), contendo os artefatos realmente afetados e modificados. Este conjunto depende da maneira como a mudança é implementada.

Considerando que o processo de análise de impacto é interativo, é possível que outros impactos, não contidos no CIC, sejam descobertos enquanto uma mudança está sendo implementada – vide Figura 2.8. O *Conjunto de Impactos Descobertos* (CID) contêm os artefatos identificados como impactados durante a implementação da mudança. O *Conjunto de Impactos Falso-Positivos* (CIFP) contêm os artefatos estimados a serem alterados que durante ou após a implementação da mudança, não foram de fato afetados. O CIR é encontrado a partir da soma de CIC e CID e a subtração do CIFP. No entanto, este modelo de processos definido por (Bohner, 2002a) define as etapas da análise de maneira abstrata e não indica como cada etapa será executada.

Sun et al. (2010) propuseram uma abordagem para AI que consiste em uma análise estática de código-fonte orientado a objetos que utiliza tipos de mudanças e dependências entre entidades para identificação dos possíveis impactados. Para a melhoria de precisão dos resultados encontrados por esta abordagem de Sun et al. (2010), foi proposta uma estratégia de melhoria da precisão da identificação do conjunto de impacto inicial utilizando uma estrutura chamada Grafo de Classes Orientada a Objetos e de Dependência entre entidades (chamado de *Object Oriented Class and Member Dependence Graph*), em que os vértices representam as entidades do código-fonte analisado e as arestas representam as dependências entre si. Para identificação de impactados regras são definidas para cada tipo de mudança relacionada a cada tipo de im-

pacto sobre como cada entidade identificada como impactada propaga um certo tipo de impacto, e as entidades impactadas são obtidas percorrendo o grafo gerado, até que a entidade impactada não possua mais dependências. Todo tipo de mudança tem seu próprio mecanismo de impacto. Eles acreditavam que se o CII for calculado com precisão suficiente, o CIR será mais preciso. No entanto, Sun et al. (2010) define uma abordagem em que possui uma técnica de AI própria, ou seja, não é possível selecionar a técnica que possua a melhor precisão para o projeto analisado, além de não trazer um resultado ou estratégia ao analista que possa ser utilizada para direcionamento da execução da mudança em si.

Já Wilkerson (2012) propôs uma taxonomia para classificar as causas do impacto da mudança de software. A taxonomia fornece um vocabulário que classifica impactos como: diretos e indiretos. Impactos diretos ocorrem quando elementos do código-fonte são editados de alguma forma, levando em relação versões anteriores do código. Impactos indiretos ocorrem quando elementos do código-fonte são impactados resultantes de uma dependência de outros elementos que são direta ou indiretamente impactados. Entretanto, por a taxonomia poder ser utilizada para identificar impactos tanto em código-fonte de paradigma procedural e orientado a objetos, foi necessário generalizar os tipos de impactos identificados, o que pode prejudicar a sua utilização em análise de impacto em código-fonte orientado a objetos, pois aspectos inerentes a este paradigma não são considerados (como polimorfismo e herança).

Devido a complexidade na execução de análise de impacto de mudança e a dificuldade do rastreamento do impacto gerado por uma técnica automatizada de análise, diversas abordagens visuais foram propostas na literatura como apoio ao analista, em uma tentativa de extração de informação do resultado obtido pela execução de uma técnica automatizada de análise. Follett e Hoeber (2010) propuseram a *ImpactViz*, uma ferramenta que utiliza uma abordagem visual para ilustrar os efeitos em cascata que alterações em classes podem afetar todo o software analisado. Eles utilizaram a extração das entidades e relacionamentos de chamadas de métodos por meio da análise do código-fonte de um repositório de software. A ferramenta se utiliza de grafo direcionado de força para representar o conjunto de chamadas de métodos, onde as arestas representam as chamadas de métodos entre as classes e os vértices representam as classes do software analisado. As arestas possuem direção que mostram a chamada de um método, ou seja, se uma aresta apontando de uma classe A para uma classe B indica que A chama um método da classe B.

A *ImpactViz* usa cores de fundo e da fonte nos nós para fornecer distinção visual entre classes. Vértices com fundo branco e escrita preta representam classes que não possuem alterações no histórico de versões do repositório de software, enquanto que os vértices com fundo preto e escrita branca representam classes que possuem alterações. O impacto de uma mudança é determinado por cada mudança em um método específico em cada alteração de versão do software. Para cada mudança é gerada uma região, que inclui a classe na qual o método impactado é definido e todas as outras classes que possuem chamadas a este método. Estas regiões de impacto são mostradas visualmente com cor de fundo em volta das classes impactadas, conforme podemos observar na Figura 2.9.

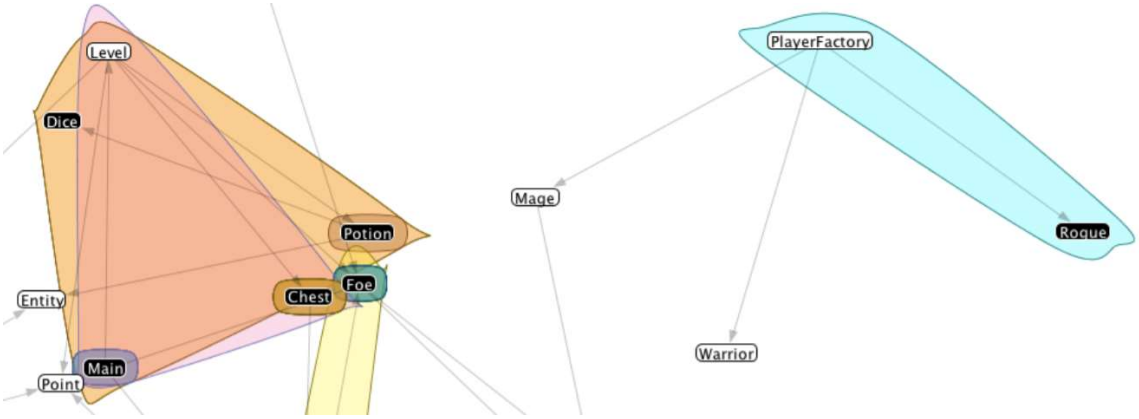


Figura 2.9: Análises de Impacto de Mudanças com a *ImpactViz*. Fonte: (Follett e Hoeber, 2010)

Já Mercer et al. (2012) propuseram uma abordagem visual de análise de impacto de mundaças utilizando uma técnica estática chamada *DiSE*. Esta técnica consiste na utilização de fatiamento de programa para a geração de um grafo de depêndencia. As mudanças são identificadas utilizando uma análise de diferença de código-fonte(*Diff*) e o impacto é identificado analisando os possíveis caminhos do grafo gerado (Rungta et al., 2012). Uma visualização do grafo de dependencia é gerado e as entidades impactadas são destacadas pela cor da borda dos nós, bem como as partes impactadas são destacadas no código-fonte, conforme podemos ver na Interface da ferramenta criada (vide Figura 2.10).

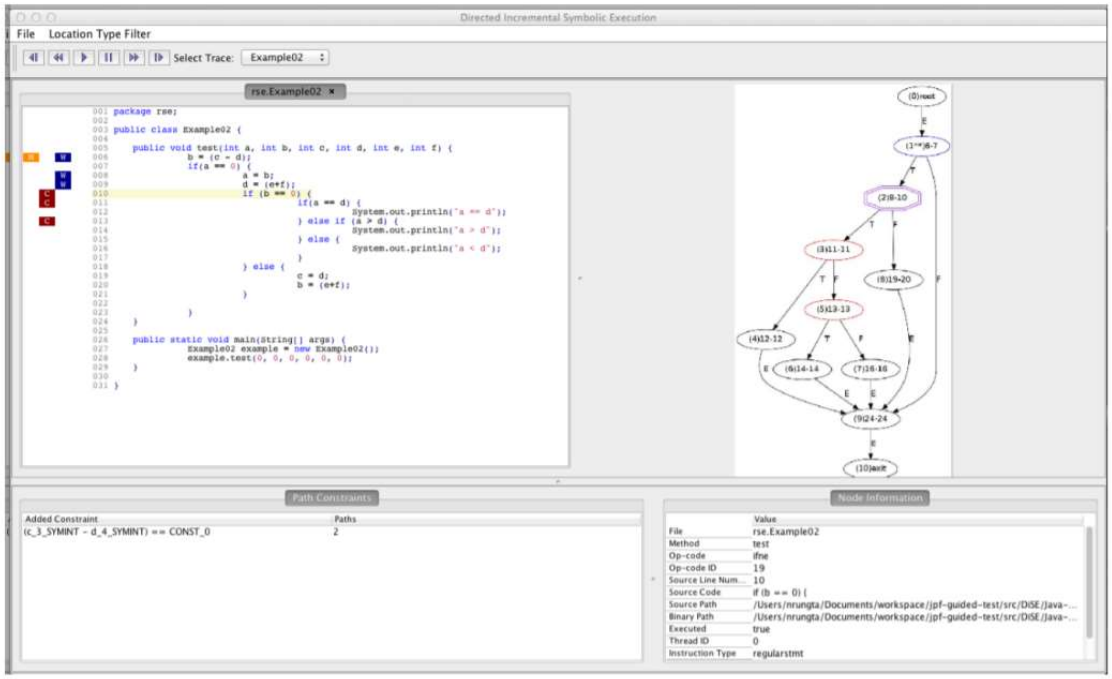


Figura 2.10: Análises de Impacto de Mudanças com a *DiSE*. Fonte: (Mercer et al., 2012)

No entanto, estas ferramentas propostas por (Follett e Hoeber, 2010), (Mercer et al., 2012) e (Di Rocco et al., 2013) têm como retorno apenas uma lista de possíveis impactados, sem

qualquer análise do conjunto gerado, e apenas mostra o resultado da execução de uma técnica de análise específica.

Outro trabalho proposto por Mohamad (2010), propôs a implementação de uma ferramenta de análise de impacto de mudanças, chamada *CIA-V*, que utiliza extração de entidades e relacionamentos de código-fonte orientado a objetos, escrito na linguagem C++, para identificar a propagação gerada por uma mudança proposta. A *CIA-V* utiliza a visualização mostrando apenas os elementos impactados e os relacionamentos que foram selecionados como justificativa do impacto. Podemos observar na Figura 2.11, a interface da *CIA-V* e o resultado da execução de uma análise de impacto.

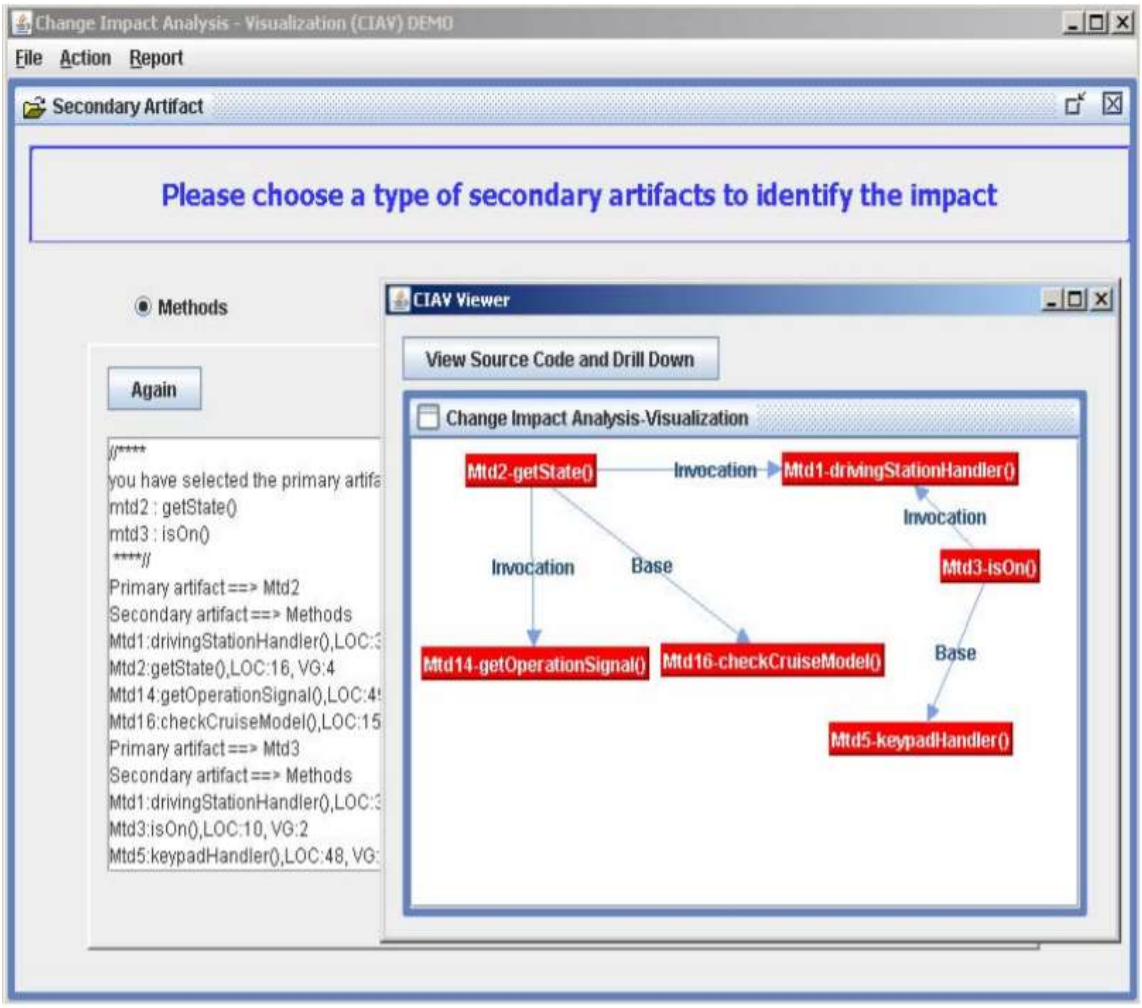


Figura 2.11: Análises de Impacto de Mudanças com a *CIA-V*. Fonte: (Mohamad, 2010)

## 2.7 Considerações Finais

Esta revisão bibliográfica foi realizada como base e apoio para a definição da abordagem deste trabalho. A classificação das técnicas de análise de impacto foram definidas com o intuito de restringir o escopo do trabalho, para análises de impacto utilizando técnicas de código-fonte. A técnica de Hatori foi escolhida para um teste de conceito da abordagem que será proposta

por ser de fácil entendimento e fácil execução (considerando as técnicas de análise de impacto de código-fonte apresentadas), pois utiliza apenas o próprio código-fonte para identificação dos possíveis impactados.

Quanto as abordagens encontradas na literatura e apresentadas, destaca-se a de Bohner (2002a), que define um modelo de processo de execução de uma análise de impacto. No entanto, este modelo traz conceitos mais abstratos da etapas da análise, pois especifica as atividades a serem executadas e não como devem ser executadas, não trazendo tantos benefícios reais ao um analista na execução de uma mudança proposta. Além disso, as abordagens (Follett e Hoeber, 2010), (Mercer et al., 2012) e (Di Rocco et al., 2013) têm como retorno apenas uma lista de possíveis impactados, sem qualquer análise do conjunto gerado, não explorando outras perspectivas ou *insights* que direcionem as etapas da análise.

Dentre as ferramentas relacionadas ao trabalho, é possível destacar a *CIA-V*, ferramenta que utiliza-se de uma técnica de análise de impacto e extração de entidades/relacionamentos para identificação de elementos do códig-fonte impactados por uma mudança (Mohamad, 2010). A *CIA-V* gera visualizações do impacto de uma mudança, gerando em suas representações, entidades impactadas e a propagação do impacto, em forma de uma estrutura muito parecida com a árvore de propagação definida em uma das perspectivas de nossa abordagem. Entretanto o objetivo deste trabalho não se limitará a apenas esta analise, mas trazer outras perspectivas que são importantes como suporte ao analista que gerencia a execução de uma mudança, como visualizar o impacto em relação a todo o código-fonte.

No próximo capítulo é apresentado a abordagem proposta por este trabalho, visando explorar as deficiências encontradas nos modelos e ferramentas apresentados.

CAPÍTULO

# 3

# *Abordagem Visual como suporte à Análise de Impacto*

Neste capítulo é apresentada a abordagem proposta, seu funcionamento, suas etapas e as perspectivas geradas com visualização de software. Também é apresentada a ferramenta $VisImpala$, desenvolvida como parte deste projeto para implementar a abordagem proposta e realizar as avaliações apresentadas.

## 3.1 Abordagem Proposta

O objetivo deste trabalho foi propor uma abordagem que auxilia e facilita a utilização dos dados gerados pela execução de uma técnica automatizada de AI no código-fonte no direcionamento das atividades de implementação de uma mudança proposta para um software. A fim de atingir este objetivo, nossa abordagem se utilizou de:

- Extração de características do código-fonte do software analisado;
- Extração de métricas de manutenibilidade dos artefatos possívelmente impactados;
- Identificação de possíveis impactados no código-fonte, gerados pela execução de uma técnica automatizada de análise de impacto;
- Utilização de Visualização de Software, para melhoria da compreensão da extensão da influência da mudança no código-fonte.

Podemos observar na Figura 3.1 o esquema que representa a abordagem proposta, bem como suas etapas.

Inicialmente, são extraídos diretamente do código-fonte orientado a objetos(do software a ser analisado) entidades que compõe o código-fonte (pacotes, classes, métodos e variáveis) e os relacionamentos hierárquicos que representam relações de pertencimento entre entidades do

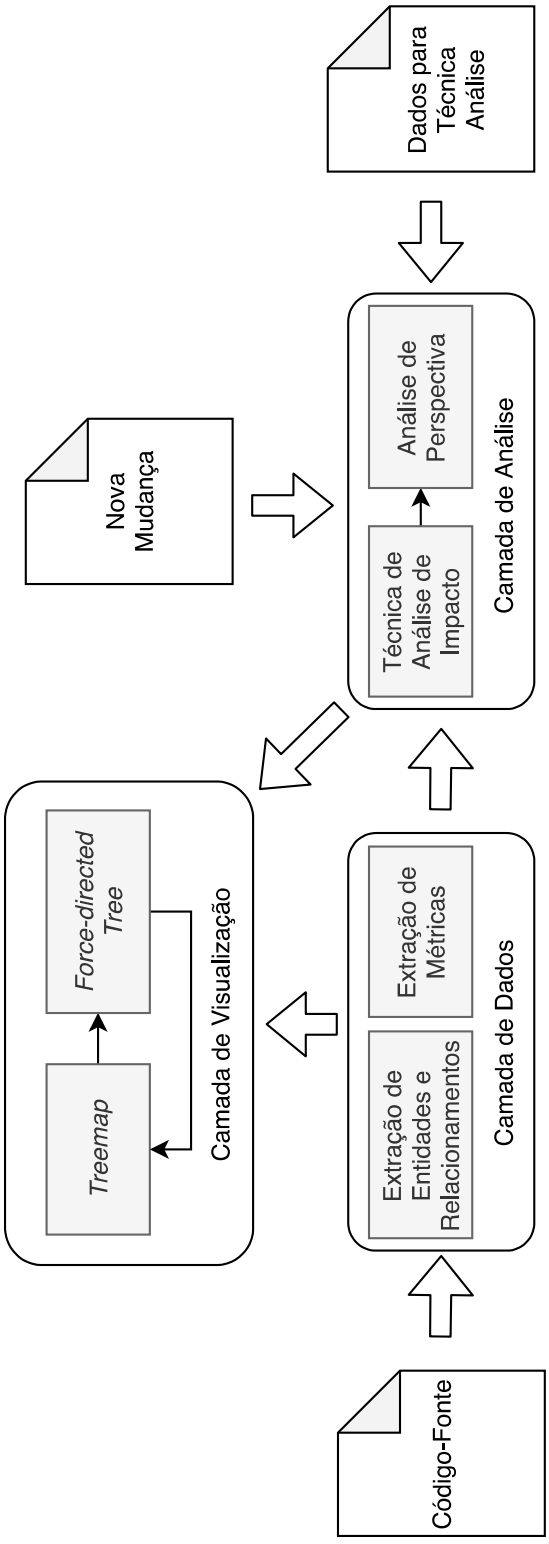


Figura 3.1: Abordagem Visual como suporte a AI. Fonte: Própria

código-fonte (por exemplo, classes que estão contidas em um pacote, ou métodos que estão contidos em uma classe). Simultaneamente, as métricas de manutenibilidade dos artefatos também são extraídas diretamente do código-fonte (vide 2.4). Esses conjuntos iniciais de dados são utilizados para a criação das visualizações que representaram o software analisado. Esta

etapa inicial, representada pela Camada de Dados (vide Figura 3.1), é executada somente se o código-fonte inicial analisado for alterado, necessitando sincronizar as entidades/relacionamentos, as métricas e as visualizações.

A partir de nova mudança proposta para análise, uma técnica automatizada de análise de impacto de mudanças é executada e um conjunto de possíveis impactados é gerado como resultado. Neste conjunto estão contidas entidades que podem ser identificadas como artefatos do código-fonte que foram extraídos na primeira etapa da abordagem (Camada de Dados). Utilizando-se do conjunto gerado, há um processamento de cada entidade impactada com base em três perspectivas: Análise de nível de impacto, análise de esforço de manutenibilidade e análise de como cada entidade foi impactada. A perspectiva é escolhida pelo analista que está utilizando a abordagem, sendo possível a utilização de apenas uma perspectiva por vez.

Após a etapa de análise da perpectiva concluída e o conjunto de possíveis impactados identificado, os dados gerados são refletidos nas visualizações, conforme a perspectiva selecionada. Estas etapas, que são representadas pela Camada de Análise (vide Figura 3.1), são executadas cada vez que uma mudança nova é proposta para análise. Para cada nova mudança proposta ou mudança no código-fonte analisado, se houver mudanças anteriores propostas, tanto a execução da técnica de AI quanto a análise de perspectiva serão executadas novamente utilizando as mudanças anteriores e a nova mudança, pois as perspectivas levam em consideração todo o conjunto de mudanças propostas. Isto é feito para garantir que as representações dos impactos estejam atualizados e sincronizados nas visualizações.

Em nossa abordagem, o formato de uma nova mudança proposta consiste na definição da entidade impactada diretamente pela mudança, o tipo de mudança (vide Figura 2.2) e um complemento da mudança (por exemplo, se o tipo da mudança proposta for mudança de assinatura do método, o complemento será a nova assinatura). Essas informações são suficientes para a execução da maioria das técnicas de análise de impacto de código-fonte orientado a objetos (Li et al., 2013).

A nossa abordagem necessita da execução de uma técnica automatizada de análise de impacto em código-fonte orientado a objetos, ou seja, que o conjunto de possíveis impactados cotenham apenas elementos de código-fonte orientado a objetos (como classes, métodos e variáveis). Mas, para algumas técnicas, são necessários dados adicionais para suas execuções (por exemplo: conjunto de rastros de execução, histórico de um repositório de software, conjunto de casos de teste, etc.). Esses dados devem servir a Camada de Análise sempre que a técnica de AI tiver que ser executada.

Quanto a Camada de Visualização, escolhemos para a abordagem a visualização *Treemap* para representação dos pacotes presentes no código-fonte analisado. Em uma proposta inicial, todas as entidades seriam representadas utilizando somente esta visualização, mas foi descartada esta proposta, após testes iniciais com análise de projetos de grande porte, pois tornavam a visualização do software inviável devido a grande quantidade de entidades e relacionamentos. Em nossa abordagem, a variável visual do tamanho dos retângulos da *Treemap* representam a média das intensidades de uma métrica de manutenibilidade escolhida pelo analista, das classes

contidas em cada pacote.

Como ainda precisaríamos representar outras entidades do código-fonte (classes e métodos), como alternativa para contornar a limitação da *Treemap*, nós propusemos para a abordagem a utilização de uma *Force-directed Tree* para representação da estrutura hierárquica interna de cada pacote, representando suas classes e métodos inclusos. Nesta visualização, o vértice que representa o pacote analisado é posicionado ao centro, com os vértices que representam as classes contidas no pacote orbitando em sua volta e os métodos contidos nas classes orbitando em volta das classes em que estão contidas. A variável visual do tamanho dos vértices foi aplicada às classes e métodos de acordo com a intensidade de uma métrica de manutenibilidade escolhida pelo analista. Já em relação à variável visual do formato dos vértices do grafo, foram escolhidos os seguintes formatos para cada tipo de entidade analisada pela nossa abordagem:

- Losânglo ou *diamond* para representar o pacote;
- Círculo para representar as classes;
- Quadrado para representar os métodos;

Como foi apresentado anteriormente, na Camada de Análise nós mencionamos na etapa de análise de perspectivas, três perspectivas diferentes que nossa abordagem traz como resultado: Análise de nível de impacto, análise de esforço de manutenibilidade e análise de como cada entidade foi impactada. Essas perspectivas serão definidas a seguir.

### 3.1.1 Análise de nível de impacto

Ao executar uma técnica de análise de impacto, o resultado obtido é um conjunto de entidades possivelmente impactadas. Os elementos desse conjunto podem ser classificados em:

- Entidades diretamente impactadas pela mudança;
- Entidades identificadas como impactadas a partir da análise de propagação gerada por uma entidade diretamente impactada;
- Entidades identificadas como impactadas a partir da análise de propagação gerada por uma entidade identificada ao analisar uma propagação.

A perspectiva de análise de nível de impacto, consiste no agrupamento de entidades impactadas em níveis de propagação, gerando uma estrutura hierárquica que representa a propagação da análise de uma mudança proposta, chamada de árvore de propagação. O primeiro nível da árvore (raiz) é representado pelas entidades diretamente impactadas pela mudança. Já o segundo nível, é representado pelas entidades impactadas pela análise da propagação e os demais níves, a entidades impactadas pela análise da propagação de uma entidade identificada ao analisar a propagação. Podemos observar na Figura 3.2 o exemplo de uma árvore de propagação de uma mudança proposta.

Essa perspectiva foi proposta, por exemplo, para auxiliar o analista na elaboração do cronograma de implementação da mundança, visto que a perspectiva traz uma idéia de ordenação

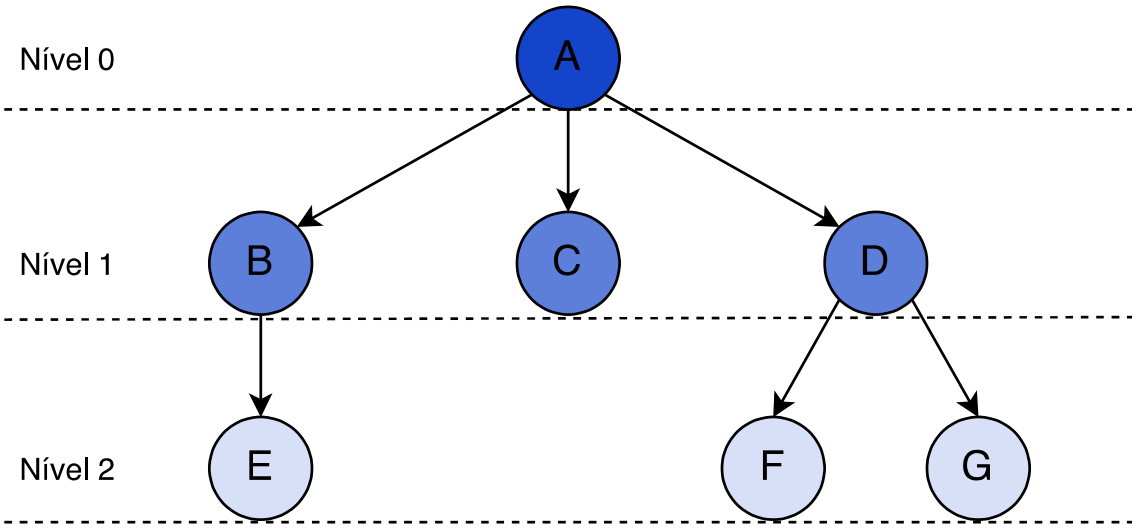


Figura 3.2: Exemplo de árvore de propagação. Fonte: Própria

baseada na ordem em que os elementos fora impactados. Observando o exemplo de propagação da Figura 3.2, se não houver a utilização de nossa abordagem, ou seja, apenas a execução da técnica de AI, pode ocorrer que o analista selecione elementos mais próximos das folhas da árvore de propagação antes de elementos mais próximos da raiz da propagação. Utilizando os elementos impactados do exemplo (Figura 3.2), há a possíbilidade que o elemento F seja analisado antes do elemento D, o que pode acarretar em uma falsa análise de não haver impacto ou mudanças no elemento F, porque esse impacto pode ocorrer somente depois de implementadas as mudanças no elemento D.

Neste exemplo, há uma diminuição das opacidades das cores de preenchimento dos vértices conforme os níves de propagação se distanciam do nível inicial (nível 0), que representa a entidade que foi impactada diretamente. Da mesma forma, em nossa abordagem, na *Force-directed tree* o preenchimento interno e variação de opacidade de uma cor específica, escolhida pelo analista, são utilzadas para destacar os vértices que representam entidades impactadas, levando em consideração o nível de propagação (vide Figura 3.2). Para a *Treemap*, a cor de destaque é utilizada para o prenchimento dos retângulos que representam pacotes que contêm alguma entidade impactada, mas neste caso, a opacidade da cor escolhida é a mesma do elemento que possui o nível de propagação mais próximo do nível inicial dentre os elementos impactados que estão contidos no pacote.

Como esta perspectiva utiliza-se de diferenciação de opacidade de uma cor, o cálculo da opacidade de cada nível leva em consideração todas entidades de todas as mudanças propostas, variando a opacidade em valores entre 0,1 e 1,0. Assim, é garantido que entidades que estejam no mesmo nível de propagação, mas que foram identificadas em pela análise de impacto de mudanças diferentes, tenham a mesma opacidade de cor.

### 3.1.2 Análise de esforço de manutenibilidade

Na Camada de Dados de nossa abordagem (Figura 3.1), métricas de esforço de manutenção são capturadas. Dentre estas métricas, é capturada o Índice de Manutenibilidade, que é utilizada para classificar entidades do código-fonte quanto a dificuldade de manutenção. Para esta segunda perspectiva, entidades contidas no conjunto de possíveis impactados, que foram identificados pela análise de uma mudança proposta, são classificados quanto ao esforço de manutebilidade, analisando a intensidade da métrica Índice de Manutenibilidade de cada um.

Utilizando os parâmetros padrões de classificação desta métrica, as entidades impactadas podem ser classificadas em:

- Baixo esforço de manutenção ($MI \geq 85$)
- Moderado esforço de manutenção ($65 \leq MI < 85$)
- Alto esforço de manutenção ($MI < 65$)

Essa perspectiva foi proposta, por exemplo, para auxiliar o analista na distribuição de tarefas de verificação/implementação da mudanças, permitindo-o distribuir a implementação da mudança em entidades com um alto esforço de manutenção para programadores mais experientes. Com esta perspectiva, há a possibilidade do analista obter uma visão geral do esforço da mudança como um todo, auxiliando na predição do esforço, tempo e custos necessários para a implementação de uma ou mais mudanças propostas.

Nesta segunda perspectiva, os vértices que representam os elementos impactados são destacados na *Force-directed tree* utilizando uma cor no preenchimento interno do vértice. Para cada categoria de esforço (baixo, moderado ou alto) uma cor é escolhida para identificá-la. A opacidade de cores varia de 0,1 à 1,0, sendo que:

- Para elementos das categorias de baixo e moderado esforço, quanto maior a intensidade da métrica de MI, maior o nível de opacidade;
- Para elementos da categoria de alto esforço, quanto menor a intensidade da métrica de MI, maior o nível de opacidade. Essa configuração foi escolhida diferente das demais categorias com o intuito de destacar os elementos com maior dificuldade de manutenção.

Para a *Treemap*, o pacote que possuir algum elemento impactado, será identificado com o seu preencimento utilizando a cor e opacidade do elemento impactado com maior esforço de manutenção.

### 3.1.3 Análise de tipo de impacto

A terceira perspectiva utiliza a análise das propagações e como uma entidade é afetada, quer seja por um impacto direto como um impacto propagado, levando em consideração a entidade que a impactou. Para isto, foi gerada uma lista de 91 possibilidades de tipos de impacto levando em consideração aspectos da orientação a objetos. Nesta lista estão contidas os tipos de mudanças que são compreendidas pela abordagem (vide Figura 2.2), tipos de propagações geradas por

um cada tipo de mudança e os tipos de propagações geradas por um tipo de propagação. Na Tabela 3.2 podemos observar as possibilidades de tipos de impacto, contendo: um identificador único para cada mudança/propagação, uma breve descrição da mudança/propagação, o conjunto identificadores de propagações que cada mudança/propagação pode gerar, o tipo de impacto (M para as mudanças e P para propagações), e o identificador do grupo de tipos de impacto a que a mudanças/propagação pertence.

Assim como nas demais perspectivas, as entidades impactadas por uma mudança serão identificadas pela utilização de uma cor de preenchimento. A estratégia adota inicialmente era de cada tipo de mudança/propagação ser associada a uma cor diferente. No entanto, isto se tornou inviável por causa do grande tamanho de paleta de cores necessária para este mapeamento. Então, os tipos de mudanças/propagações foram agrupados, e para cada grupo gerado, uma cor é associada. Portanto, ao identificar o tipo de mudança/propagação de uma entidade, a cor de preenchimento utilizada passou a ser a mapeada para cor do grupo que este tipo pertence. Podemos observar na Tabela 3.1 os grupos de tipos de mudanças/propagações, contendo: um identificador único do grupo, e uma breve descrição.

| ID | Descrição |
|---|---|
| G1 | Mudanças em Variáveis |
| G2 | Mudanças em Métodos |
| G3 | Mudanças em Classes |
| G4 | Outras Mudanças e Propagações |
| G5 | Propagações em Métodos de Controle de Acesso (get) ou métodos que modifiquem diretamente (set) variáveis |
| G6 | Propagações em Classes relacionadas com Métodos de Controle de Acesso (get) ou métodos que modifiquem diretamente (set) variáveis |
| G7 | Propagações em Métodos relacionadas a chamadas de métodos |
| G8 | Propagações relacionadas com Herança de Classes |

Tabela 3.1: Grupos de tipos de mudanças/propagações. Fonte: Própria

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 1 | changeVisibility variável private to friendly | 46;48;44; | M | G1 |
| 2 | changeVisibility variável private to protected | 46;48;44; | M | G1 |
| 3 | changeVisibility variável private to public | 50;52;44; | M | G1 |
| 4 | changeVisibility variável friendly to private | 54;55;44; | M | G1 |
| 5 | changeVisibility variável friendly to protected | 56;58;44; | M | G1 |
| 6 | changeVisibility variável friendly to public | 50;52;44; | M | G1 |
| 7 | changeVisibility variável protected to private | 54;55;60;61;44; | M | G1 |
| 8 | changeVisibility variável protected to friendly | 60;61;44; | M | G1 |
| 9 | changeVisibility variável protected to public | 50;52;44; | M | G1 |
| 10 | changeVisibility variável public to private | 62;63;44; | M | G1 |
| 11 | changeVisibility variável public to friendly | 64;65;44; | M | G1 |
| 12 | changeVisibility variável public to protected | 66;67;44; | M | G1 |
| 13 | changeAttributeType variável to type | 68;44 | M | G1 |
| 14 | changeAttributeName variável to variável2 | 44; | M | G1 |
| 15 | add variável | 69; | M | G1 |
| 16 | remove variável | 70;71;44; | M | G1 |
| 17 | changeAttribute variável | 44; | M | G1 |
| 18 | changeVisibility método friendly to private | 72;44; | M | G2 |
| 19 | changeVisibility método friendly to protected | 44; | M | G2 |
| 20 | changeVisibility método friendly to public | 44; | M | G2 |
| 21 | changeVisibility método protected to private | 73;44; | M | G2 |
| 22 | changeVisibility método protected to friendly | 74;44; | M | G2 |
| 23 | changeVisibility método protected to public | - | M | G2 |
| 24 | changeVisibility método public to private | 73;44; | M | G2 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 25 | changeVisibility método public to friendly | 75;44; | M | G2 |
| 26 | changeVisibility método public to protected | 76;44; | M | G2 |
| 27 | add método | 77;78 | M | G2 |
| 28 | remove método | 44; | M | G2 |
| 29 | changeSemantics método | 44; | M | G2 |
| 30 | changeSignature método | 79;44; | M | G2 |
| 31 | changeSignatureParameter método | 80;44; | M | G2 |
| 32 | changeSignatureException método | 81;44; | M | G2 |
| 33 | changeReturnType método | 82;44; | M | G2 |
| 34 | changeVisibility classe friendly to public | - | M | G3 |
| 35 | changeVisibility classe public to friendly | 83;84; | M | G3 |
| 36 | add classe | - | | G3 |
| 37 | remove classe | 85;86;87;88; | M | G3 |
| 38 | addInheritance classe | - | M | G3 |
| 39 | removeInheritance classe | 89;90;91 | M | G3 |
| 40 | changeVisibility método private to friendly | - | M | G2 |
| 41 | changeVisibility método private to public | - | M | G2 |
| 42 | changeVisibility método private to protected | - | M | G2 |
| 43 | changeVisibility método protected to public | - | M | G2 |
| 44 | Métodos chamadores deste método/classe/variável serão afetados quanto a sua lógica/semântica | 44; | P | G4 |
| 45 | Outras Consequências Não Identificadas | 45; | P | G4 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 46 | Se esta variável possuir método de controle de acesso (get), poderá gerar código morto se não houver método de outra classe pertencente a outro pacote utilizando este controle de acesso. | 47;44; | P | G5 |
| 47 | Métodos de classes do mesmo pacote não precisarão utilizar o método de controle de acesso (get) desta variável para utilizar seu valor, podendo utilizar a variável diretamente. | 44; | P | G5 |
| 48 | Se esta variável possuir método que a modifique diretamente (set), poderá gerar código morto se não houver método de outra classe pertencente a outro pacote utilizando este controle de acesso. | 49;44; | P | G5 |
| 49 | Métodos de classes do mesmo pacote não precisarão utilizar o método que a modifique diretamente (set) desta variável para modificar o valor, podendo utilizar a variável diretamente. | 44; | P | G5 |
| 50 | Se este variável possuir método de controle de acesso (get), gerará código morto (o próprio controle de acesso); | 51;44; | P | G5 |
| 51 | Métodos de qualquer classe do projeto não precisarão utilizar o método de controle de acesso (get) desta variável para utilizar seu valor, podendo utilizar a variável diretamente; | 44; | P | G5 |
| 52 | Se este variável possuir método que a modifique diretamente (set), gerará código morto (o próprio modificador direto); | 53;44; | P | G5 |
| 53 | Métodos de qualquer classe do projeto não precisarão utilizar o método que a modifique diretamente (set) desta variável para modificar seu valor, podendo utilizar a variável diretamente; | 44; | P | G5 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 54 | Necessidade de implementação de método de controle de acesso (get) e método que a modifique diretamente (set) na classe em que está contido este variável, se o mesmo não existir e houver acessos diretos ou modificações diretas a variável por métodos de classes do mesmo pacote; | - | P | G6 |
| 55 | Métodos de classes pertencentes ao mesmo pacote da classe que contém essa variável deverão utilizar método de controle de acesso (get) ou o método que a modifique diretamente (set) se estiverem acessando ou modificando diretamente esta variável; | 44; | P | G5 |
| 56 | Se esta variável possuir método de controle de acesso (get), poderá gerar código morto se não houver método de outra classe pertencente a outro pacote utilizando este controle de acesso, exceto subclasses da classe que está contido o variável; | 57; | P | G5 |
| 57 | Métodos de classes do mesmo pacote e métodos de subclasses da classe que contêm essa variável não precisarão utilizar o método de controle de acesso (get) para utilizar seu valor, podendo utilizar a variável diretamente; | 44; | P | G5 |
| 58 | Se esta variável possuir o método que modifica o variável diretamente (set), poderá gerar código morto se não houver método de outra classe pertencente a outro pacote utilizando para atribuição de valor, exceto subclasses da classe que está contido o variável; | 59; | P | G5 |
| 59 | Métodos de classes do mesmo pacote e métodos de subclasses da classe que contêm esse variável não precisarão utilizar o método que modifica o variável diretamente (set) para atribuição de valor, podendo atribuir diretamente; | 44; | P | G5 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 60 | Necessidade de implementação de método de controle de acesso (get) e método que a modifique diretamente (set) na classe em que está contido este variável, se o mesmo não existir e houver acessos diretos ou modificações diretas a variável por métodos de subclasses da classe que contém esse variável; | - | P | G6 |
| 61 | Métodos de classes pertencentes a subclasses da classe que contém esse variável deverão utilizar método de controle de acesso (get) ou o método que a modifique diretamente (set) se estiverem acessando ou modificando diretamente esta variável; | 44; | P | G5 |
| 62 | Necessidade de implementação de método de controle de acesso (get) e método que a modifique diretamente (set) na classe em que está contido este variável, se o mesmo não existir e houver acessos diretos ou modificações diretas a variável, exceto métodos da classe em que está contido a variável; | - | P | G6 |
| 63 | Métodos de qualquer classe do projeto deverão utilizar método de controle de acesso (get) ou o método que a modifique diretamente (set) se estiverem acessando ou modificando diretamente esta variável, exceto métodos da classe em que está contido a variável; | 44; | P | G5 |
| 64 | Necessidade de implementação de método de controle de acesso (get) e método que a modifique diretamente (set) na classe em que está contido este variável, se o mesmo não existir e houver acessos diretos ou modificações diretas a variável, exceto métodos da classe em que está contido a variável e métodos de classes do mesmo pacote; | - | P | G6 |
| 65 | Métodos de classes pertencentes a classes do mesmo pacote da classe que contém esse variável deverão utilizar método de controle de acesso (get) ou o método que a modifique diretamente (set) se estiverem acessando ou modificando diretamente esta variável; | 44; | P | G5 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 66 | Necessidade de implementação de método de controle de acesso (get) e método que a modifique diretamente (set) na classe em que está contido esta variável, se o mesmo não existir e houver acessos diretos ou modificações diretas a variável, exceto métodos da classe em que está contido a variável, métodos de classes do mesmo pacote e métodos de subclasses da classe que contém essa variável; | - | P | G6 |
| 67 | Métodos de classes pertencentes a classes do mesmo pacote e métodos de subclasses da classe que contém esse variável deverão utilizar método de controle de acesso (get) ou o método que a modifique diretamente (set) se estiverem acessando ou modificando diretamente esta variável; | 44; | P | G5 |
| 68 | Mudança de retorno do método de controle de acesso (get) | 44; | P | G5 |
| 69 | Possível necessidade de implementações do método de controle de acesso (get) e do método que a modifique diretamente (set) na classe em que está contido esta variável. | - | P | G6 |
| 70 | Remoção do método de controle de acesso da variável (get). | 44; | P | G5 |
| 71 | Remoção do método que modifique a variável diretamente (set) | 44; | P | G5 |
| 72 | Métodos chamadores deste método pertencentes ao mesmo pacote que o contém serão afetados quanto a sua lógica/semântica, por não poderem chamar mais este método diretamente. | 44; | P | G7 |
| 73 | Métodos chamadores deste método que não pertencerem a mesma classe deste método serão afetados quanto a sua lógica/semântica por não poderem chamar mais este método diretamente. | 44; | P | G7 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 74 | Métodos chamadores deste método pertencentes a subclasses da classe que o contém serão afetados quanto a sua lógica/semântica por não poderem chamar mais este método diretamente. | 44; | P | G7 |
| 75 | Métodos chamadores deste método que não pertencerem ao mesmo pacote da classe que o contém serão afetados quanto a sua lógica/semântica por não poderem chamar mais este método diretamente. | 44; | P | G7 |
| 76 | Métodos chamadores deste método que não pertencerem ao mesmo pacote da classe que o contém ou que não estão contidos em subclasses da classe que o contém serão afetados quanto a sua lógica/semântica por não poderem chamar mais este método diretamente. | 44; | P | G7 |
| 77 | Se o método for adicionado em uma interface, classes que a implementarem deverão implementar esse método. | - | P | G8 |
| 78 | Se o método adicionado for abstrato, classes concretas que estendem da classe abstrata em que o método foi adicionado deverão obrigatoriamente o implementar. | - | P | G8 |
| 79 | Métodos das subclasses da classe que contém esse método que reescrevam esse método precisarão mudar assinatura também. | 44; | P | G8 |
| 80 | Métodos das subclasses da classe que contém esse método que o sobrescrevem precisarão modificar assinatura dos parâmetros para o novo informado. | 44; | P | G8 |
| 81 | Métodos chamadores deste método serão afetados quanto a sua lógica/semântica, especificamente no bloco catch que trata esta exceção. | 44; | P | G7 |
| 82 | Métodos das subclasses da classe que contém esse método que o sobrescrevem precisarão modificar o tipo de retorno para o novo informado. | 44; | P | G8 |

**Tabela 3.2 continuação da página anterior**

| ID | Descrição | Propagação | Tipo | Grupo |
|---|---|---|---|---|
| 83 | Subclasses que não pertencerem ao mesmo pacote desta classe serão afetados quanto a herança existente, pois não conseguiram acessar a classe pai; | - | P | G8 |
| 84 | Métodos que não pertencerem a classes do mesmo pacote desta classe que carregam essa classe serão afetados quanto a sua lógica/semântica, não podendo carregá-la por falta de acesso a classe diretamente. | 44; | P | G8 |
| 85 | Métodos que carregam essa classe serão afetados quanto a sua lógica/semântica. | 44; | P | G7 |
| 86 | Subclasses dessa classe serão afetadas na definição de herança (extends); | - | P | G8 |
| 87 | Métodos pertencentes a subclasses dessa classe que sobrescrevem métodos da classe a ser removida serão afetados quanto a sua lógica/semântica. | 44; | P | G8 |
| 88 | Os construtores das subclasses dessa classe serão afetados se tiverem alguma chamada a construtor da classe a ser removida. | 44; | P | G8 |
| 89 | Métodos da classe que foram herdados da classe pai e foram re-implementados serão afetados quanto a sua lógica/semântica, pois podem explicitamente/implicitamente possuírem a chamada ao método da classe pai. | 44; | P | G8 |
| 90 | Métodos da classe que chamarem algum método da classe pai serão afetados quanto a sua lógica/semântica. | 44; | P | G8 |
| 91 | Construtores da classe serão afetados quanto a sua lógica/semântica, pois métodos de outras classes podem instanciar objetos desta classe a partir da declaração da classe pai. | 44; | P | G8 |

Tabela 3.2: Tipos de Mundanças e Propagações. Fonte: Própria

Esta perspectiva foi proposta, por exemplo, para auxiliar o analista na implementação de casos de testes, direcionando-o na identificação dos trechos a serem testados e como serão testados, a partir da identificação da maneira como a entidade foi impactada.

Cada vértice que representa um elemento impactado é identificado na *Force-directed Tree* utilizando a cor de preenchimento mapeada para o grupo a que pertence o tipo de mudança/-propagação identificada. Em relação a representação dos pacotes na *Treemap*, a cor de prenchimento de um pacote que contém uma entidade impactada representa o grupo de maior frequência.

## 3.2 Ferramenta VisImpala

Para podermos testar a nossa abordagem e comprová-la, foi necessário o desenvolvimento de uma ferramenta que implementa a nossa abordagem. A VisImpala é a ferramenta que foi desenvolvida para análise de código-fonte de um software escrito na linguagem de programação JAVA, que auxilia o processo de análise de impacto de mudanças, produzindo visualizações de software e destacando os possíveis impactados nas perspectivas de nossa abordagem. Na Figura 3.3 podemos observar a organização das funcionalidades da ferramenta VisImpala.

A ferramenta foi divida em módulos para que as funcionalidades e cada etapa da abordagem se tornassem independentes. A VisImpala possui duas camadas principais:

- A API, que engloba os módulos de análise do código-fonte, execução da técnica de análise de impacto de Hatori, analisador de perspectiva e atualização dos dados da visualização. A API, também é responsável pela persistência dos dados gerados pela extração de informação do código-fonte. Esta camada, foi implementada utilizando a linguagem JAVA;

- A View, que engloba as visualizações propostas pela abordagem e a configuração de parâmetros que refletem na visualização, como por exemplo a escolha da perspectiva para análise da mudança ou a proposta de uma nova mudança para análise. A View, que apresenta uma interface na WEB, foi implementada utilizando as linguagens HTML, CSS e Javascript.

Podemos observar na Figura 3.4, uma visão geral da interface da ferramenta VisImpala.

Os dados de entrada da ferramenta, que são os arquivos de código-fonte a serem analisados (com extensão .java), são processados utilizando uma biblioteca de *parser* chamada srcML, em sua versão 0.9.5 (Collard et al., 2013). Esta é uma biblioteca de exploração e manipulação de código-fonte, que identifica componentes do código em JAVA e se utiliza de marcação de tags para identificar elementos da sintaxe da linguagem analisada (Alnabhan et al., 2018). É possível observar na Figura 3.5 o exemplo de um trecho de código-fonte escrito em JAVA e o arquivo XML gerado após a execução do srcML.

Utilizando-se desse arquivo XML, a ferramenta VisImpala captura as entidades e relacionamentos que compõe o código-fonte que será utilizada para análise, bem como a extração das métricas de manutenibilidade utilizadas pela abordagem percorrendo e identificando as tags

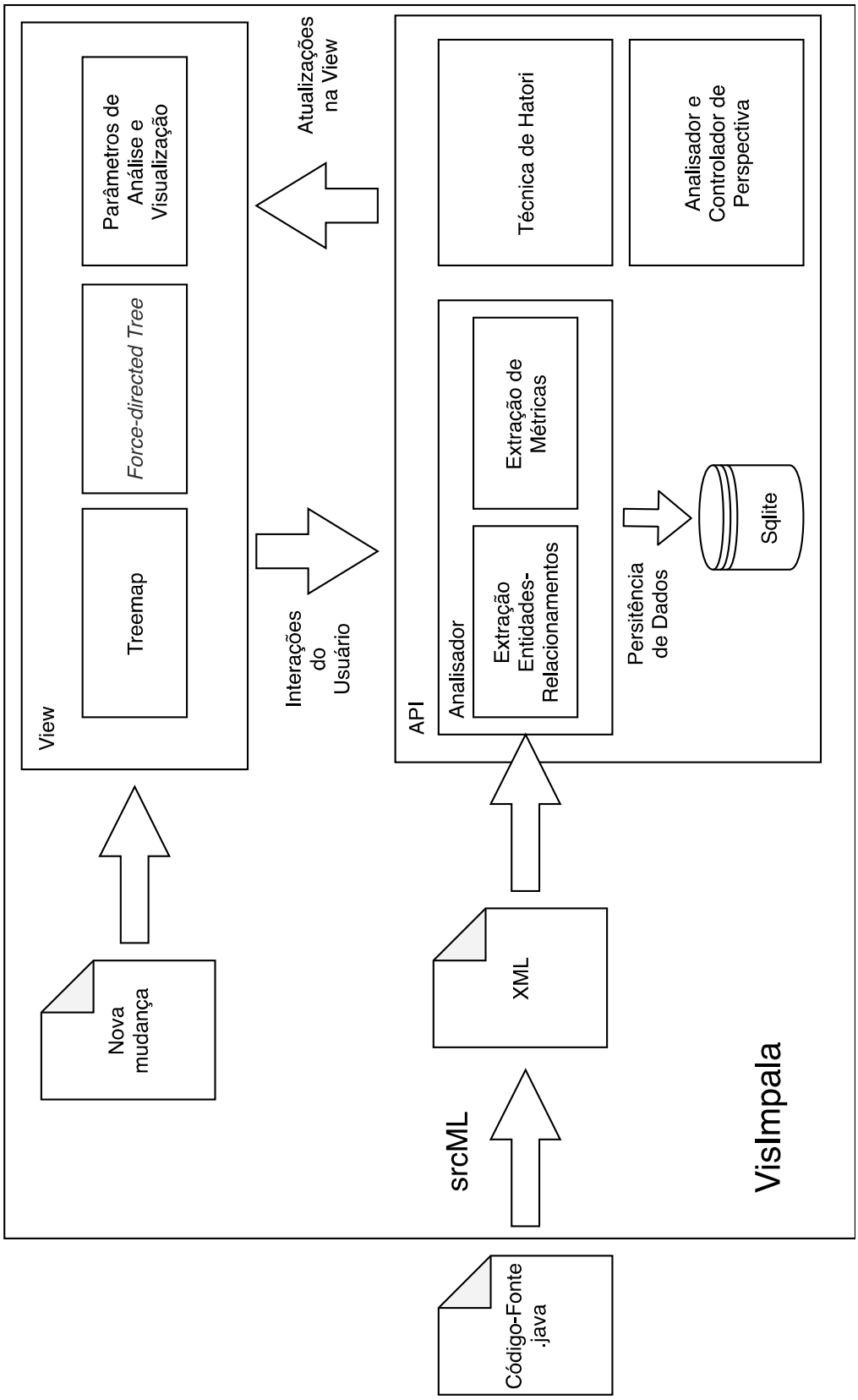


Figura 3.3: Esquema de organização VisImpala. Fonte: Própria

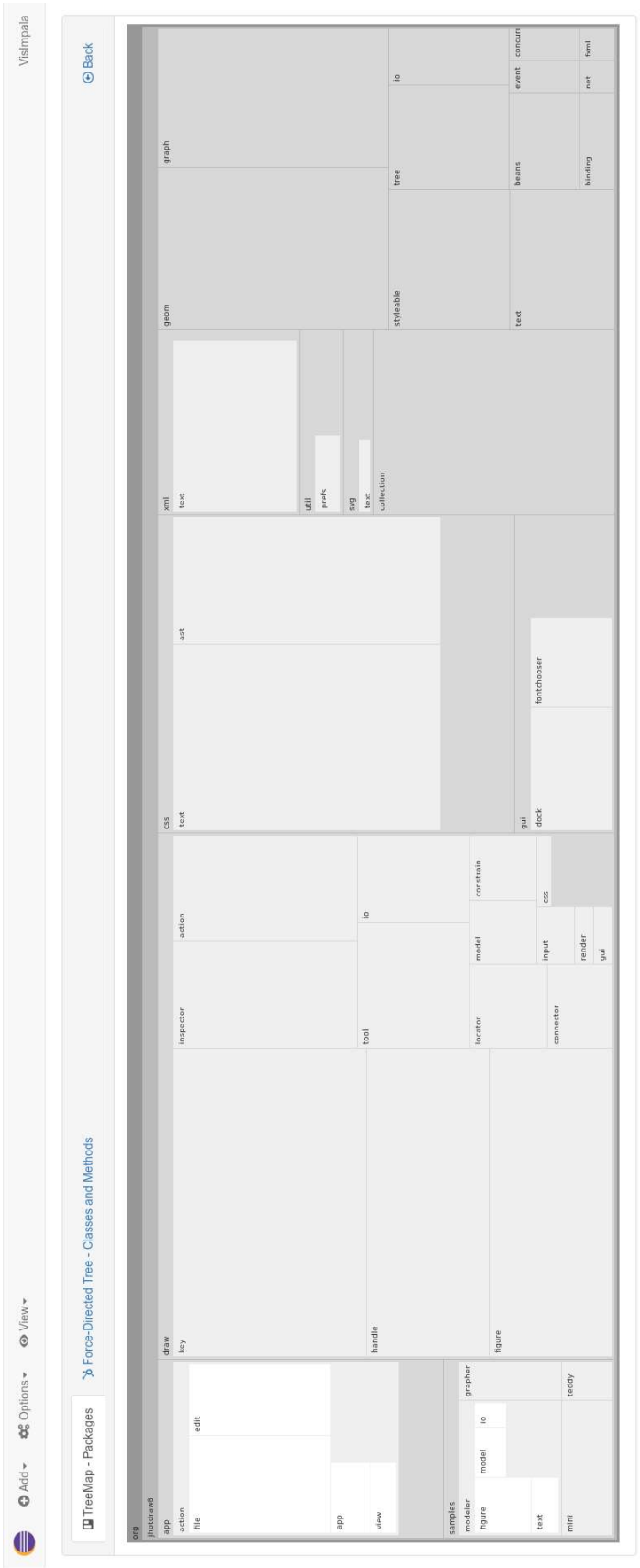


Figura 3.4: Interface inicial da VisImpala. Fonte: Própria

```java
import java.util.Date;

public class HelloWorld {
    public static void main(String[] args) {
        Date now = new Date();
        System.out.println("Hello World!");
        System.out.println("now: " + now);
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="0.9.5" language="Java"
    filename="Documentos/teste.java"><import>import <name><name>java</name
    ><operator>.</operator><name>util</name><operator>.</operator><name>Date
    </name></name>;</import>

<class><specifier>public</specifier> class <name>HelloWorld</name> <block>{
    <function><specifier>public</specifier> <specifier>static</specifier>
        <type><name>void</name></type> <name>main</name><parameter_list
        >(<parameter><decl><type><name><name>String</name><index>[]</index
        ></name></type> <name>args</name></decl></parameter>)</parameter_list
        > <block>{
        <decl_stmt><decl><type><name>Date</name></type> <name>now</name>
            <init>= <expr><operator>new</operator> <call><name>Date</name
            ><argument_list>()</argument_list></call></expr></init></decl
            >;</decl_stmt>
        <expr_stmt><expr><call><name><name>System</name><operator>.</operator
            ><name>out</name><operator>.</operator><name>println</name></name
            ><argument_list>(<argument><expr><literal type="string">"Hello
            World!"</literal></expr></argument>)</argument_list></call></expr
            >;</expr_stmt>
        <expr_stmt><expr><call><name><name>System</name><operator>.</operator
            ><name>out</name><operator>.</operator><name>println</name></name
            ><argument_list>(<argument><expr><literal type="string">"now: "
            </literal> <operator>+</operator> <name>now</name></expr
            ></argument>)</argument_list></call></expr>;</expr_stmt>
    }</block></function>
}</block></class></unit>
```

Figura 3.5: Exemplo de trecho de código escrito em JAVA e correpondente com marcação do srcML. Fonte: Própria

que repesentam cada entidade. Este módulo, chamado de Analisador, percorre cada arquivo do código-fonte em busca de tags que caracterizam por exemplo, a definição de uma classe, ou a chamada de um método.

Em nossos testes, verificamos que as atividades de extração de entidades/relacionamentos e extração de métricas são muito despendiosas em relação a tempo e esforço de processamento. Este fato, impossibilitava a interação em tempo real do usuário com as visualizações, visto que seria necessário essa análise todas as vezes que houvesse alguma mudança de perspectiva ou uma nova proposta de mudança. Portanto, tivemos a necessidade de implementação de uma persistência dos dados da extração (entidades/relacionamentos e métricas). Para isto, utilizamos o banco de dados embarcado SQLite em sua versão 3.3.1, o qual obteve uma ótima performance, mesmo com a ferramenta manipulando simultaneamente grande quantidade de dados

pela ferramenta (Owens, 2006).

Após a extração das entidades/relacionamentos e das métricas, a API envia para a View os dados iniciais para que os pacotes capturados sejam inseridos na *Treemap* (vide Figura 3.4). Com estes dados a View constrói a *Treemap* com a representação hierárquica dos pacotes do software analisado. Foi escolhida uma escala de cinza para a cor de cada nó da Treemap, aumentando a luminosidade conforme as camadas se aprofundam na hieraquia. O atributo tamanho varia de acordo com a média das métricas das classes contidas no pacote, podendo o usuário selecionar a métrica para o projeto analisado. O usuário ao selecionar um pacote, faz com que a ferramenta carregue sob demanda as entidades (classes, métodos) contidas no pacote e as representa na *Force-directed tree*, sendo nó central da visualização o pacote selecionado e os ramos as classes e métodos, seguindo as especificações de representação da abordagem proposta. O tamanho dos vértices que representam classes e métodos é definido pelo valor relativo à métrica selecionada pelo usuário. Esse carregamento por demanda foi necessário pois em nossos testes identificamos problemas de performance de renderização de elementos na tela por parte da implementação das técnicas de visualização, em pacotes com um grande números de classes e métodos. A biblioteca D3.js foi utilizada para gerar as visualizações (Zhu, 2013).

Podemos observar na Figura 3.6 a representação da estrutura hierárquica interna de um pacote gerada pela VisImpala. A ferramenta possui a funcionalidade de *zoom* para a visualização da estrutura interna do pacote, e a implementação da força aplicada nos elementos da *Force-directed tree* foi feita de maneira que os vértices irão reorganizar seus posicionamentos para evitar sobreposição. Ao passar o mouse sobre algum vértice, a ferramenta apresenta informações do elemento que o vértice representa como: Assinatura da classe/método, identificador da métrica visualizada e intensidade da métrica.

Para análise de impacto, a VisImpala utiliza a técnica de Hatori, técnica estática de código-fonte orientado a objetos (2.3). A técnica de Hatori foi escolhida devido a facilidade de sua aplicação em conjunto com a nossa abordagem considerando as técnicas apresentadas (Li et al., 2013), pois para sua implantação e execução, ela necessita como entrada um conjunto de entidades e relacionamentos extraídos do código-fonte, conjunto este que já é capturado para execução de nossa abordagem. Para sua execução, o usuário cadastra na View uma nova mudança (vide Figura 3.7), que dispara ua requisição para a API, contendo a mudança e a perspectiva selecionada. A técnica de Hatori é executada utilizando as entidades/relacionamentos persistidos e a mudanças proposta. Com o conjunto de possíveis impactados criado, a VisImpala executa o módulo Analisador e Controlador de Perspectiva utilizando o conjunto gerado e a perspectiva selecionada para obter as atualizações que serão necessárias às visualizações. Por fim, a API retorna para a View os dados da perspectiva para serem refletidas nas visualizações. Outra funcionalidade presente na VisImpala é a de gerenciamento de multiplas propostas de mudança. As mudanças propostas são listadas e o usuário pode selecionar se o impacto de uma mudança estará ou não visível nas visualizações. Ao selecionar esta opção, a View dispara uma requisição para a API, para que a análise da perspectiva ocorra novamente, mas levando em consideração apenas o impacto das mudanças visíveis. Ao término da análise, a API retorna a

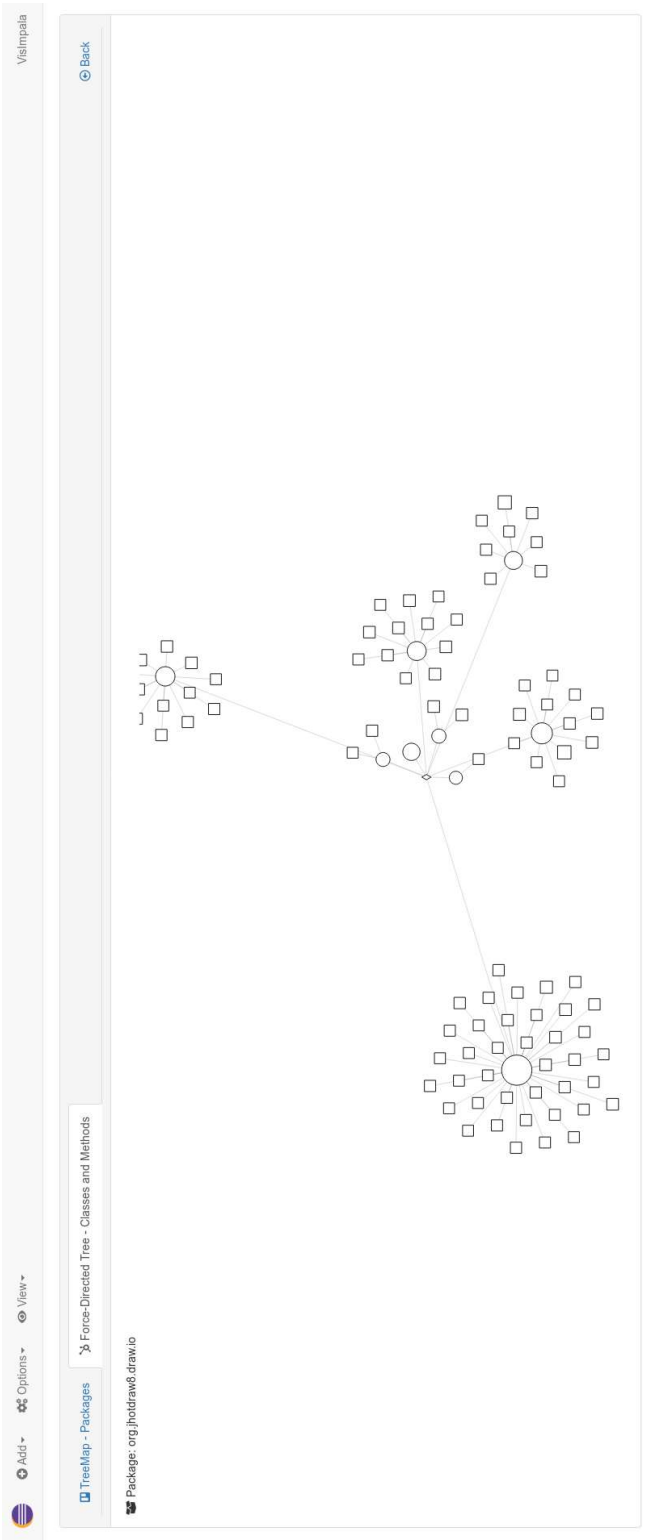


Figura 3.6: *Force-directed tree* que representa a estrutura hieráquica interna de um pacote. Fonte: Própria

View, que atualiza as visualizações de acordo com o resultado da perspectiva selecionada. Há também nesta tabela de mudanças a opção de exclusão de uma mudança proposta. De mesmo modo da opção anterior, o impacto e o resultado da perspectiva á atualizado, desconsiderando a


Figura 3.7: Formulário de cadastro de uma nova mudança. Fonte: Própria

mudança excluída. É possível observar na Figura 3.8 a tabela de mudanças propostas.


Figura 3.8: Tabela de mudanças propostas. Fonte: Própria

## 3.3 Considerações Finais

Neste capítulo foi definida uma abordagem que especifica etapas de análise de impacto em código-fonte orientado a objetos e resultados que podem servir como direcionamento de como a mudança deve ser executada baseada na análise do impacto encontrado, objetivo principal deste trabalho. Esta abordagem se diferencia das demais apresentadas (vide 2.6) em relação a como a análise deve ser conduzida, com etapas bem definidas, e resultados além de somente quais elementos foram impactados. Para que a abordagem fosse avaliada e validada, e suas etapas automatizadas, a ferramenta VisImpala foi produzida. No próximo capítulo, são apresentados os testes de prova de conceito que foram realizados, bem como os resultados encontrados e avaliações se a ferramenta e a abordagem serviram para o propósito proposto.

CAPÍTULO

# 4

# *Avaliação e resultados*

Neste capítulo serão definidos os teste realizados com a ferramenta VisImpala a fim de comprovar nossa abordagem por meio dos resultados obtidos. Para isto, foram feitos testes que comprovassem a eficácia da abordagem na identificação dos impactados em cada uma das perspectivas e um teste para demonstrar a análise de impacto levando em consideração múltiplas mudanças propostas.

## 4.1 Definições iniciais

Os testes inicias foram definidos da seguinte forma: Uma mudança é proposta e adicionada a VisImpala. Utilizando o conjunto de possíveis impactados, manualmente se obtém o resultado. O resultado é comparado ao resultado obtido pela ferramenta. Para todos os testes, a métrica de linhas de código foi utilizada para tamanho dos elementos nas visualizações. Para o teste da perspectiva de nível de impacto, a cor de preenchimento utilizada para destacar entidades impactadas foi configurada para azul. No caso da análise pela perspectiva de de esforço de manutenibilidade para cada categoria são: verde para baixa dificuldade, amarelo para dificuldade moderada e vermelho para alta dificuldade. Os índices de categorização utilizados são os de valor padrão definido pela métrica de Índice de Manutenibilidade. Quanto a análise pela perspectiva de como cada entidade foi impactada, foram configuradas na VisImpala cores para cada grupo de tipos de mudança/propagação, visto na Tabela 4.1.

## 4.2 Teste das Perspectivas

Os primeiros testes foram executados inicialmente analisando um sistema orientado a objetos escrito na linguagem JAVA chamado de Biblioteca. Este sistema foi implementado especialmente para o teste inicial da VisImpala. Deste sistema de controle de blibliotecas foram extraidos pela ferramenta: 5 pacotes, 21 classes, 250 métodos, 377 variáveis e 2189 relacionamentos. Para análise das perspectivas, é definido na Tabela 4.2 a mudança proposta.

| Grupo | Cor |
|---|---|
| G1 | Rosa |
| G2 | Violeta |
| G3 | Laranja |
| G4 | Ciano |
| G5 | Verde |
| G6 | Vermelho |
| G7 | Amarelo |
| G8 | Azul |

Tabela 4.1: Mapeamento de cores e grupos para análise de impacto configurado na VisImpala. Fonte: Própria

| Entidade | Tipo de Mudança |
|---|---|
| biblioteca.controladores.Controlador.adicionaLivroEmprestimo | mudancaAssinatura |

Tabela 4.2: Mudança utilizada para teste inicial. Fonte: Própria

Para validação da pespectiva de nível de impacto foi necessário utilizar o conjunto de impactados para a elaboração manual da árvore de propagação, que pode ser observada na Figura 4.1. Para que possamos identificar cada método nas visualizações, definimos um identificador para cada método impactado (vide Figura 4.1).

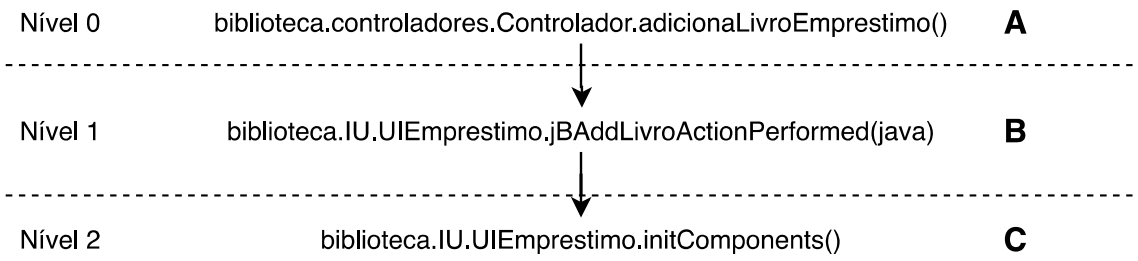


Figura 4.1: Árvore de propagação da mudança proposta. Fonte: Própria

Após a execução da análise da prespectiva pela ferramenta, os elementos impactados são identificados com uma cor apenas e há variação de opacidade conforme se distancia da raiz da árvore de propagação. Podemos observar na Figura 4.2 as representações internas dos dois pacotes com entidades impactadas. O resultado gerado pela ferramenta é o esperado, pois o elemento C é menos opaco que o elemento A, sendo justificado pelo fato do elemento A ser o elemento da raiz, enquanto o nível de C é 2.

Já na visualização dos pacotes, utilizando a *Treemap*, a opacidade da cor de preenchimento deve ser a do elemento contido que possuir o nível mais próximo da raíz da árvore de propagação. Então, as opacidades de A e B foram utilizadas pela ferramenta, conforme podemos observar na Figura 4.3, seguindo assim as regras definidas pela abordagem.

Para a validação da perspectiva de esforço de manutenibilidade, foi utilizada a mesma mudança proposta. É possível observarmos na Figura 4.4 que a entidade A representa o próprio método relacionado diretamente com a mudança e a cor do preenchimento condiz com o nível de dificuldade baixa, categorizado pela abordagem proposta, uma vez que possui índice de ma-

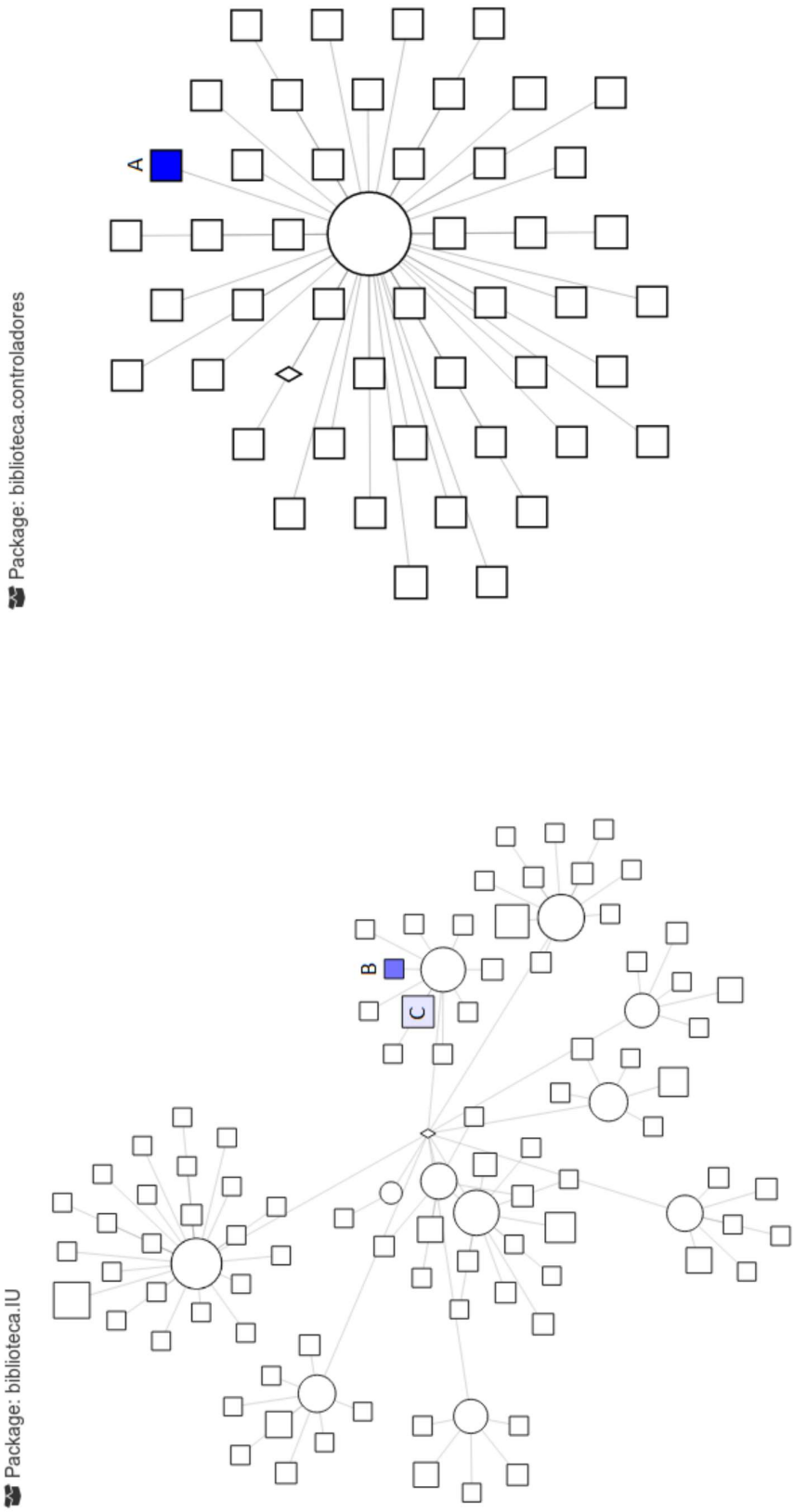


Figura 4.2: Estrutura interna dos pacotes analisados quanto a perspectiva de nível de impacto. Fonte: Própria

nutenibilidade igual 136,88. Pode-se concluir também que o método B foi categorizado como de alta dificuldade, pois possui índice de manutenibilidade igual a 35,64 e o método C como de moderada dificuldade pois possui índice igual a 77,10.

Quanto a visualização dos pacotes, utilizando a *Treemap*, a opacidade e a cor de preenchimento devem ser a do elemento contido que possuir o menor valor para índice de manutenibilidade, sendo o de maior dificuldade de manutenção. Então, as cores e opacidades de A e C foram utilizadas pela ferramenta, conforme podemos observar na Figura 4.5, seguindo assim as regras definidas pela abordagem.

Para a validação da perspectiva de como cada entidade foi impactada, foi utilizada a mesma mudança proposta. Para identificar que tipo de impacto foi propagado para cada entidade possivelmente impactada, foi feita uma análise de cada impactado com base no conjunto de possíves tipos de mudanças/propagações. Como os possíveis impactados são gerados a partir do resultado da execução da técnica de análise, os mesmos métodos foram impactados e os mesmos pacotes foram relacionados. O tipo de impacto da entidade A pode ser facilmente verificado pelo fato de essa entidade ser impactada diretamente pela mudança proposta. Portanto, o tipo de impacto é identificado como a própria mudança. Utilizando o mapeamento de cores, o tipo de impacto da entidade A é classificada no grupo G2. A cor mapeada para este grupo é a cor violeta, mesma resultado que a visualização gerada pela VisImpala produziu, conforme podemos observar na Figura 4.6.

Para analisarmos os próximo elemento impactado gerado pela propagação do impacto de A, nesse caso a entidade B, é preciso analisar a mudança associada ao impacto e verificar as possíveis propagações a ela associada. Na lista proposta pela abordagem de possíveis mudanças/propagações, as propagações possíveis para a categoria da mudança de assinatura são observadas na Tabela 4.3.

| ID | Descrição | Grupo |
|---|---|---|
| 79 | Métodos das subclasses da classe que contém esse método que reescrevam esse método precisarão mudar assinatura também | G8 |
| 44 | Métodos chamadores deste método/classe/variável afetados quanto à sua lógica/semântica | G4 |

Tabela 4.3: Opções de tipo de impacto. Fonte: Própria

Ao analisar os relacionamentos carpturados pela VisImpala da classe em que está contida a entidade B (classe *biblioteca.UI*), não há um relacionamento de herança com a classe que contêm a entidade diretamente impactada (classe *Controlador*). Portanto, a entidade B é classificada quanto ao tipo de impacto no grupo G4. No mapeamento proposto, este grupo é representado pela cor ciano, mesma cor mapeada pela VisImpala para a entidade B, como podemos observar na Figura 4.6.

Ao verificarmos a tabela de possíveis mudanças/propagações, a única propagação disponível para elementos do grupo G4 é novamente imapctos do grupo G4. Portanto, a entidade C é classificada quanto ao tipo de impacto no grupo G4. No mapeamento proposto, este grupo é representado pela cor ciano, mesma cor mapeada pela VisImpala para a entidade C, como

podemos observar na Figura 4.6.

## 4.3 Teste de Conjunto de Mudanças e Escalabilidade

Apesar de os testes iniciais da nossa ferramenta utilizando o software Biblioteca como aplicação analisada validarem a nossa abordagem, ela não é suficiente para avaliar a escalabilidade da VisImapala e verificar sua performance na análise de um sistema. Para isto, o software JhotDraw (Gamma e Eggenschwiler, 2004), um framework Java de código aberto que auxilia no desenvolvimento de aplicativos gráficos, foi utilizado em uma segunda avaliação. A ferramenta VisImpala capturou do código-fonte da versão 7.0.6 do JhotDraw: 59 pacotes, 606 classes, 6447 métodos, 8775 variáveis e 49.117 relacionamentos. A extração dos dados no primeiro teste foi efetuado em todo o código-fonte de uma só vez, porque a biblioteca srcML possibilita a geração de um único arquivo para análise/extração de uma só vez. No entanto, esta estratégia se tornou inviável devido à quantidade de entidades. Para isso, foi feita uma mudança no Analisador da ferramenta, de modo que apenas uma classe seja lida por vez, e seus dados e métrica sejam extraídos, o que possibilitou paralelismo, diminuindo drasticamente o tempo de extração.

Com isto, um segundo teste foi produzido, agora analisando o código-fonte do JhotDraw, para avaliar a análise de uma perspectiva ao propor múltiplas mudanças. A perspectiva escolhida foi a de análide de esforço de manutenção. Os parâmetros de cores e limites das categorias do Índice de Manutenibilidade permanecem os mesmos do primeiro teste. As mudanças propostas para análise podem ser vistas na Tabela 4.4.

| ID | Entidade | Tipo de Mudança |
|---|---|---|
| 1 | org.jhotdraw8.css.CssColor.getColor() | Mudança de assinatura |
| 2 | org.jhotdraw8.css.text.<br>CssNumberConverter.<br>produceTokensNonnull(TT,org.jhotdraw8.io.<br>IdFactory,Consumer) | Mudança de assinatura |
| 3 | org.jhotdraw8.css.text.<br>CssPaintConverter.<br>produceTokensNonnull(TT,org.jhotdraw8.io.<br>IdFactory,Consumer) | Mudança de assinatura |

Tabela 4.4: Proposta de múltiplas mudanças. Fonte: Própria

Para este teste, vamos observar no pacote *org.jhotdraw8.css.text* o efeito da propagação da mudança, pois este pacote possui entidades impactadas diretamente e entidades impactadas pela propagação, o que facilitará a análise do resultado do teste. A visualização da estrutura deste pacote e análise gerada pela perspectiva de dificuldade de manutenção gerada pela VisImpala, pode ser observado na Figura 4.7

Ao selecionar apenas a visualização da mudança 2 na ferramenta, apenas as entidades impactadas por essa mudança são destacadas (vide Figura 4.8). Com isso, é possível analisar os impactos de forma seletiva ou em conjunto de 2 ou mais mudanças de forma simultânea.

## 4.4 Considerações Finais

Neste capítulo, foi apresentado a avaliação da abordagem porposta por este trabalho por meio de testes com a ferramenta VisImpala. Primeiramente, foi avaliado o resultado de uma análise de impacto de uma mudança proposta considerando e avaliando cada perspectiva da abordagem, utilizando o resultado produzido pela ferramenta. Para testar a escalabilidade da ferramenta, foi proposto testes em um projeto grande de código aberto, e além disso, foi avaliado e validade que a abordagem suporta uma ou múltiplas mudanças propostas simultaneamente ou cada uma por vez.

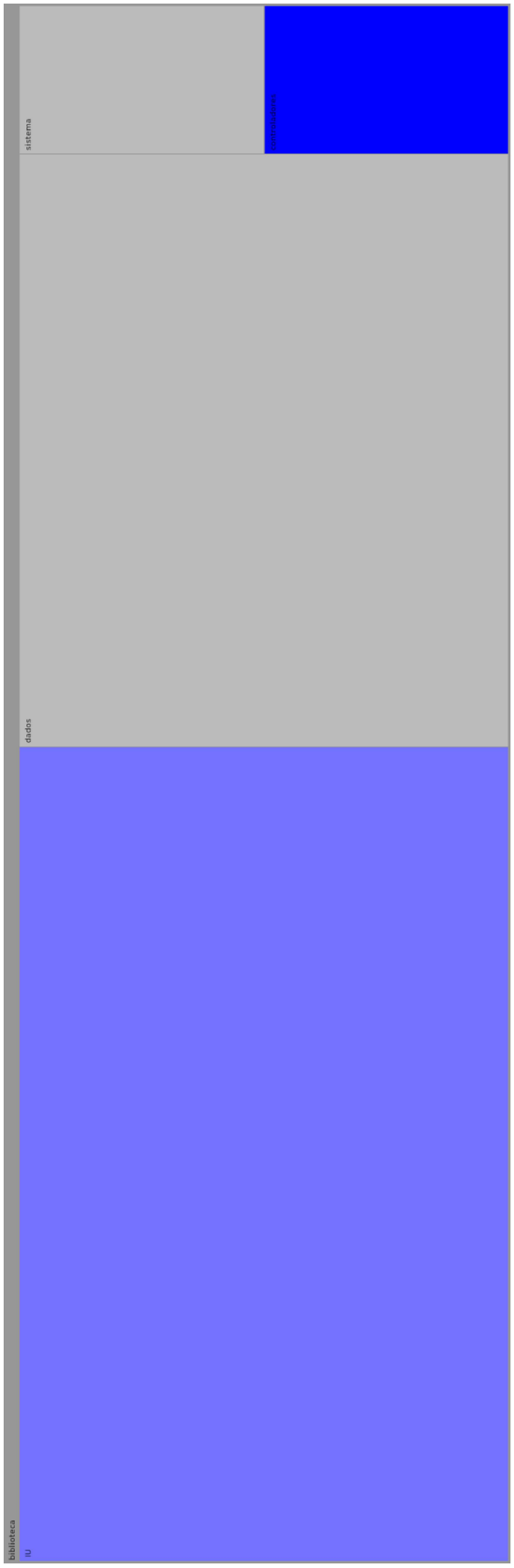


Figura 4.3: *Treemap* com pacotes impactados destacados pespectiva de nível de impacto. Fonte: Própria

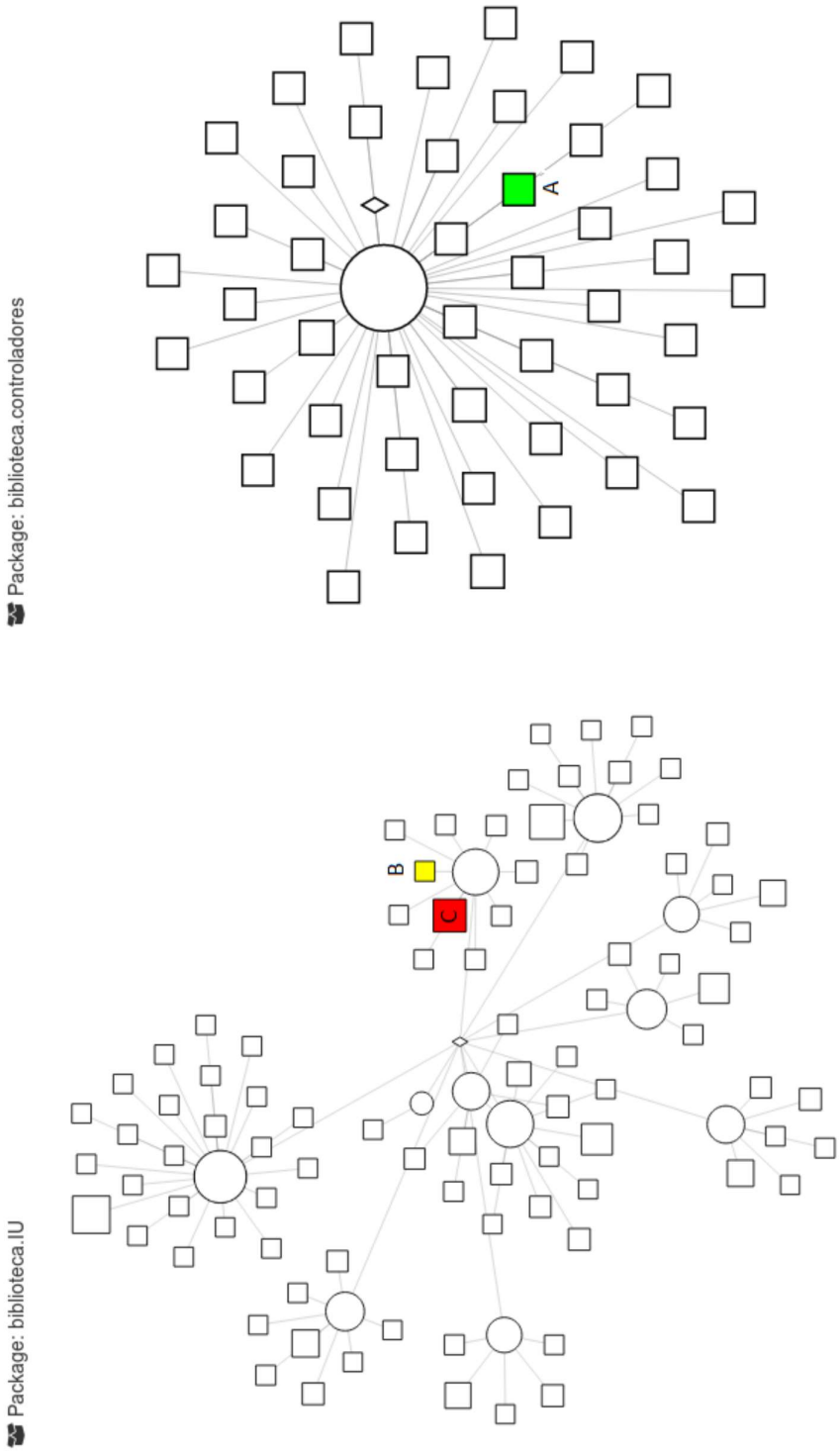


Figura 4.4: Estrutura interna dos pacotes analisados quanto a perspectiva de esforço de manutenibilidade. Fonte: Própria

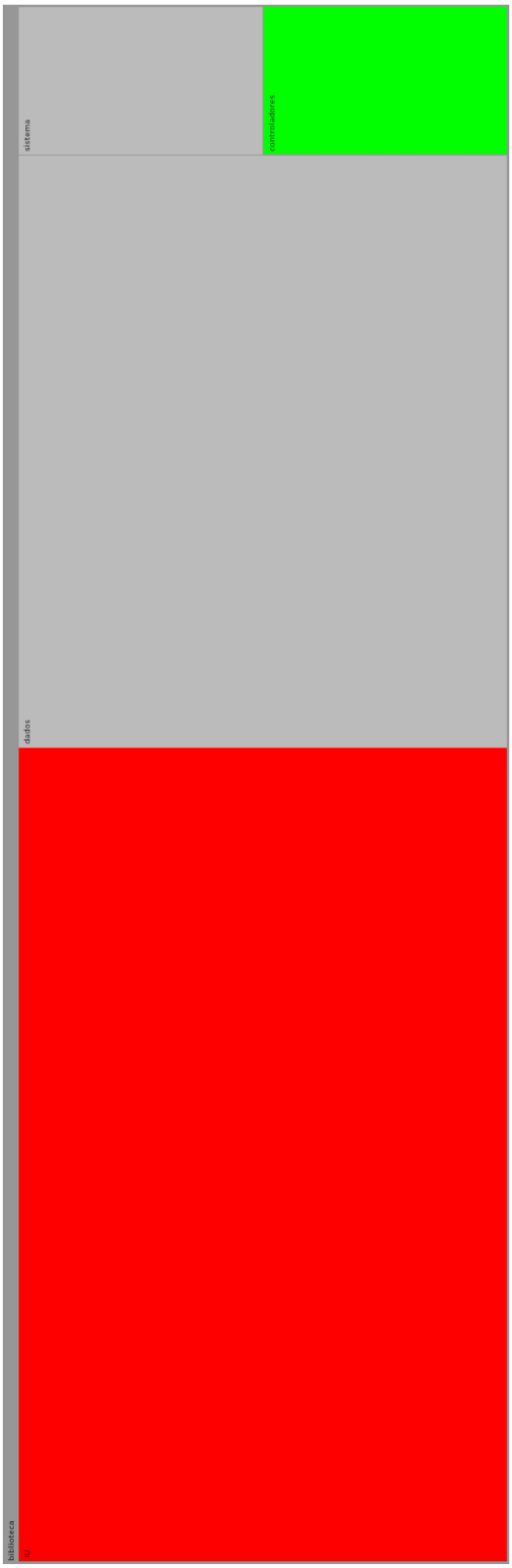


Figura 4.5: *Treemap* com pacotes impactados destacados quanto a perspectiva de esforço de manutenibilidade. Fonte: Própria

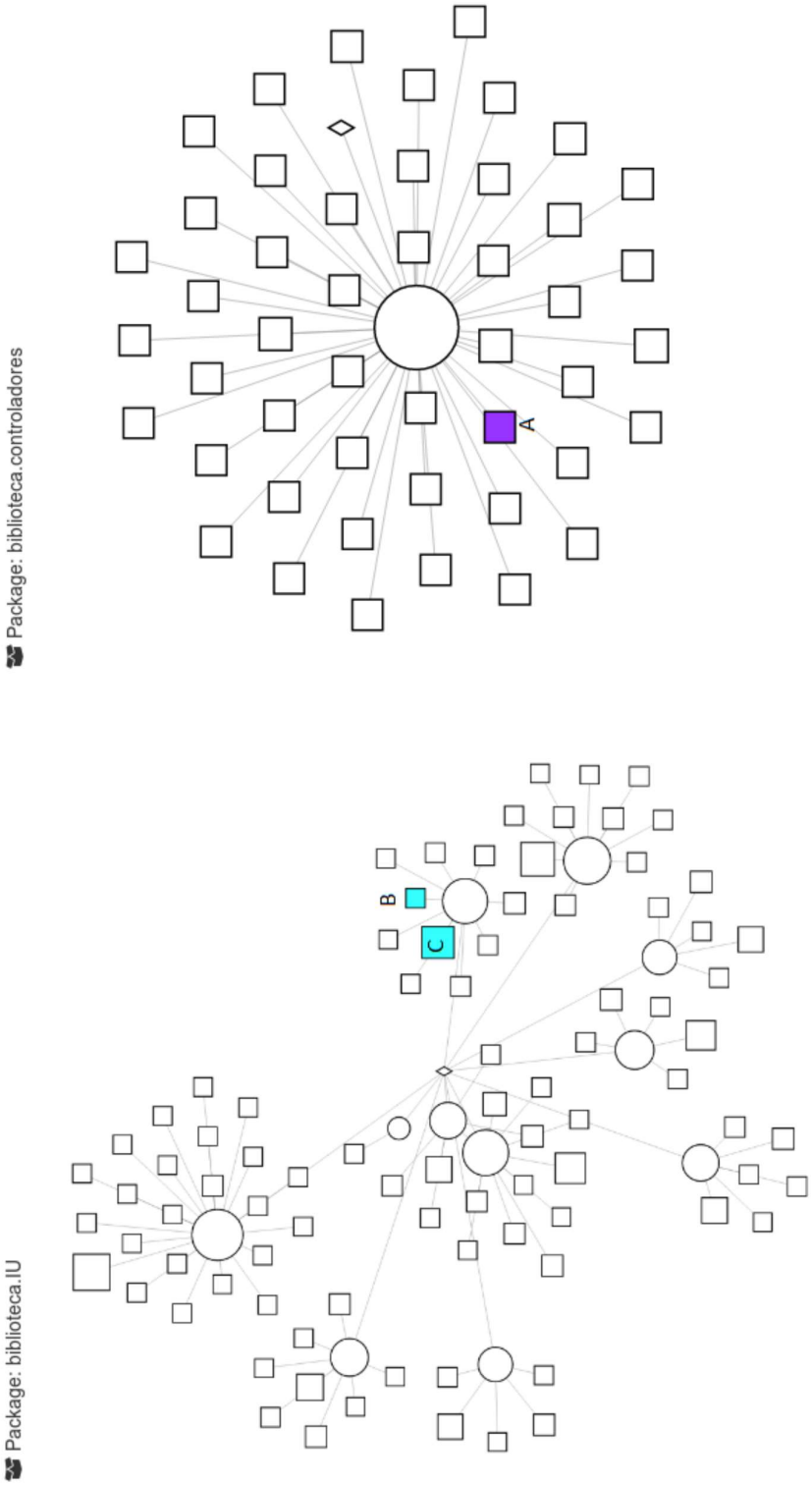


Figura 4.6: Estrutura interna dos pacotes analisados quanto a perspectiva do tipo de impacto. Fonte: Própria

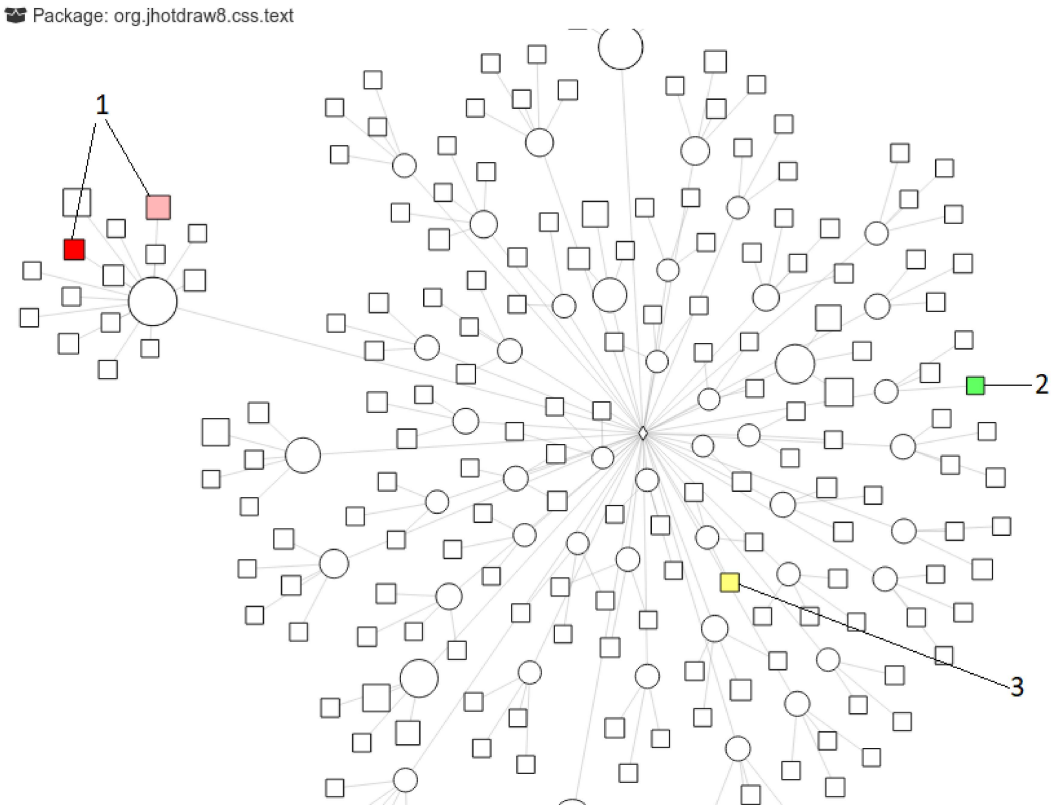


Figura 4.7: Estrutura interna do pacote analisados *org.jhotdraw8.css.text* quanto a perspectiva dificuldade de manutenção. Fonte: Própria

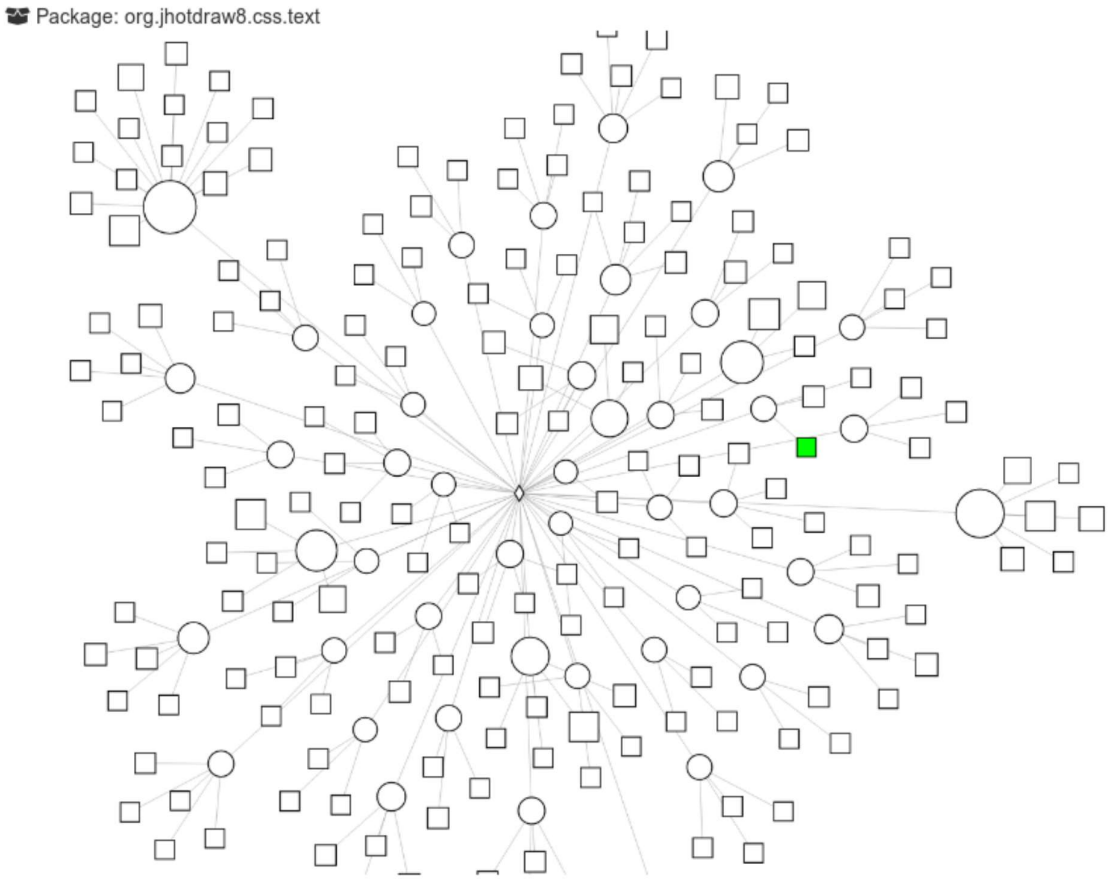


Figura 4.8: Impacto gerado ao selecionar apenas a mudança 2. Fonte: Própria

CAPÍTULO

# 5

# *Considerações Finais e Trabalhos Futuros*

Neste trabalho, foi investigada a hipótese de melhorar a compreensão do resultado produzido por uma técnica de análise de impacto de mudanças em código-fonte utilizando Visualização de Software e Métricas de Manutenibilidade. E através da análise de relacionamentos e intensidade das métricas, foram possíveis a extração de resultados que podem servir como base para estratégias e tomadas de decisão por parte de um analista em relação a implementação da técnica. O resultado encontrado pela aplicação da abordagem em uma análise se mostra muito promissor para auxiliar não somente na condução da análise de impacto, mas direcionar como a mudança será implementada. Outro aspecto importante, é que a abordagem proposta pode servir como base comparativa de técnicas de análise de impacto, visto que os resultados obtidos são independentes da técnica escolhida, dando ao analista, uma flexibilidade que as demais ferramentas não permitiam.

A ferramenta VisImpala, desenvolvida como suporte à nossa abordagem proposta, é utilizada para demonstrar os resultados obtidos e as avaliações. Como resultado, a estratégia proposta ajuda na identificação, priorização e ranqueamento de entidades impactadas por uma mudança e permite apontar os resultados visuais e observações apresentados em cada uma das perspectiva da abordagem. Em recente revisão sistemática da literatura, foram identificadas ferramentas de análise de impacto em código-fonte (Galbo, 2017). Todas as ferramentas relacionadas implementavam uma técnica de análise de impacto de mudanças e traziam como resultado um conjunto de possíveis impactados. Diferentemente, a VisImpala não representa a implementação de uma nova técnica de análise de impacto, mas sim um conjunto de perspectivas geradas a partir do processamento desse conjunto gerado. Isto faz com que o analista possa escolher qual técnica de análise de impacto de adequa melhor ao software analisado. O resultado disto é que mesmo o analista utilize uma técnica de análise de impacto que possui um

precisão ruim, a abordagem irá permitir que as perspectivas sejam geradas, visto que a análise e processamento são realizados em cada entidade possivelmente impactada.

Utilizando o resultado da nossa abordagem, é permitido: 1) Exploração visual de um software em diferentes níveis de detalhes; 2) Análise do nível de impacto, gerando uma árvore de propagação; 3) Análise da dificuldade de implementação de uma mudança; 4) Análise do tipo de impacto em uma entidade que será gerada por uma mudança; 5) Visualização de entidades do código-fonte impactados pela execução de uma técnica automática. O fato de que métricas de manutenibilidade e técnicas de análise de impacto serem muito utilizadas para mensurar dificuldade de manutenção a indústria de software, reforçam a importância da abordagem como suporte a análise de impacto.

Um artigo científico foi produzido com parte deste trabalho, sendo submetido e aprovado na CISTI'2020 - 15ª Conferência Ibérica de Sistemas e Tecnologias de Informação intitulado "*A visual approach to support Change Impact Analysis in object-oriented source code*", onde foram apresentados os resultados dos testes preliminares da ferramenta (Biazini et al., 2020).

O srcML, que foi utilizado na VisImpala para extração dos dados do código-fonte escrito na linguagem JAVA, tem suporte a diversas linguagens de programação (C++, C#). Como elementos do código-fonte que possuem o mesmo sentido de representação (uma classe, por exemplo, pode ser representada em todas as linguagens relaciondas anteriormente) são representados pela mesma tag (do exemplo, pela tag <class>), isto facilita a aplicação da nossa abordagem em código-fonte de diversas linguagens de programação, sendo preciso apenas alguns ajustes quanto a tags específicas de cada linguagem.

Apesar de os testes realizados validarem a abordagem, um experimento controlado projetado com o objetivo de identificar a eficácia da implementação da abordagem no processo de análise de impacto em uma empresa de software. A ferramenta VisImpala também deve ser avaliada por um grupo híbrido de analistas com foco na experiência do usuário com a interface, realizando ajustes com o feedback por partes dos usuários em relação a dificuldades na utilização da ferramenta. Um teste na indústria se faz necessário também, visto que assim, a eficácia total da abordagem e da ferramenta seriam atestadas.

Adicionalmente, pretende-se atualizar a abordagem para levar em consideração alguns aspectos de linguagem orientada a objetos não foram tratados, como herança múltipla e funções lambda. Com adição de novos relacionamentos e novas entidades capturadas e analisadas, as perspectiva precisarão ser atualizadas, como por exemplo, quanto a perspectiva de tipo de impacto, a tabela de opções de mudanças/propagações será atualizada.

E por fim, algumas perguntas podem ser feitas em relação a abordagem e a ferramenta que podem direcionar trabalhos futuros:

- É preciso verificar se as Técnicas de Visualizão escolhidas, foram as mais adequadas para se destacar os resultados das perpectivas da abordagem, ou a adição de mais vizualizações pode melhorar o que está sendo visualizado ou se outras técnicas de visualização são mais adequadas para a que facilite a utilização dos dados das perspectivas geradas;

- Outra melhoria se refere a tentiva de aplicações de técnicas híbridas, aplicando várias

técnicas de análise de tipos diferentes citados (vide 2);

- Conforme o tamanho do software analisado aumenta, o tempo execução de extração de elementos e relações do código-fonte aumenta, pois há mais código para analisar. Uma possibilidade é verificar se um processo de paralelização pode diminuir este tempo de execução e melhorar a performance da ferramenta;
- A ferramenta deve ser integrada a uma IDE para facilitação de sua utilização e integração com os demais programas utilizados pelo analista.

# Referências


ALNABHAN, M.; HAMMOURI, A.; HAMMAD, M.; ATOUM, M.; AL-THNEBAT, O. 2d visualization for object-oriented software systems. In: *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, IEEE, 2018, p. 1–6.

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. Efficient and precise dynamic impact analysis using execute-after sequences. In: *Proceedings of the 27th international conference on Software engineering*, ACM, 2005, p. 432–441.

ARLEO, A.; DIDIMO, W.; LIOTTA, G.; MONTECCHIANI, F. A distributed multilevel force-directed algorithm. *IEEE Transactions on Parallel and Distributed Systems*, v. 30, n. 4, p. 754–765, 2018.

BALZER, M.; DEUSSEN, O. Exploring relations within software systems using tree-map enhanced hierarchical graphs. In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 2005, p. 1–6.

BIAZINI, R. P.; CORREIA, R. C. M.; ELER, D. M.; GARCIA, R. E. A visual approach to support change impact analysis in object-oriented source code. *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, p. 1–6, 2020.

BÍBLIA tradução de joão ferreira de almeida. *São Paulo: Sociedade Bíblica do Brasil*, v. 2, 1969.

BOHNER, S. A. Extending software change impact analysis into cots components. In: *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, IEEE, 2002a, p. 175–182.

BOHNER, S. A. Software change impacts-an evolving perspective. In: *Software Maintenance, 2002. Proceedings. International Conference on*, IEEE, 2002b, p. 263–272.

BOHNER, S. A.; ARNOLD, R. S. An introduction to software change impact analysis. *Software change impact analysis*, p. 1–26, 1996.

BORG, M.; WNUK, K.; REGNELL, B.; RUNESON, P. Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context. *IEEE Transactions on Software Engineering*, v. 43, n. 7, p. 675–700, 2017.

CHEN, C.; ALFAYEZ, R.; SRISOPHA, K.; BOEHM, B.; SHI, L. Why is it important to measure maintainability and what are the best ways to do it? In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, p. 377–378.

CITO, J.; LEITNER, P.; RINARD, M.; GALL, H. C. Interactive production performance feedback in the ide. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, p. 971–981.

COIMBRA, R. T.; RESENDE, A.; TERRA, R. A correlation analysis between halstead complexity measures and other software measures. In: *2018 XLIV Latin American Computer Conference (CLEI)*, IEEE, 2018, p. 31–39.

COLLARD, M. L.; DECKER, M. J.; MALETIC, J. I. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, p. 516–519.

CZIBULA, I. G.; CZIBULA, G.; MIHOLCA, D.-L.; ONET-MARIAN, Z. An aggregated coupling measure for the analysis of object-oriented software systems. *Journal of Systems and Software*, v. 148, p. 1–20, 2019.

DELFIM, F. M.; SCATALON, L. P.; PRATES, J. M.; GARCIA, R. E. Visual approach for change impact analysis: A controlled experiment. In: *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, IEEE, 2015, p. 391–396.

DI ROCCO, J.; DI RUSCIO, D.; IOVINO, L.; PIERANTONIO, A. Traceability visualization in metamodel change impact detection. In: *Proceedings of the Second Workshop on Graphical Modeling Language Development*, 2013, p. 51–62.

FOLLETT, M.; HOEBER, O. Impactviz: visualizing class dependencies and the impact of changes in software revisions. In: *Proceedings of the 5th international symposium on Software visualization*, 2010, p. 209–210.

GALBO, S. P. D. *A survey of impact analysis tools for effective code evolution*. Tese de Doutoramento, 2017.

GAMMA, E.; EGGENSCHWILER, T. Jhotdraw. 2004.

GHOTRA, B.; MCINTOSH, S.; HASSAN, A. E. Revisiting the impact of classification techniques on the performance of defect prediction models. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, 2015, p. 789–800.

GOVE, R. Force-directed graph layouts by edge sampling. In: *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, IEEE, 2019, p. 1–5.

HALEEM, H.; WANG, Y.; PURI, A.; WADHWA, S.; QU, H. Evaluating the readability of force directed graph layouts: A deep learning approach. *IEEE computer graphics and applications*, v. 39, n. 4, p. 40–53, 2019.

HALSTEAD, M. H.; ET AL. *Elements of software science*, v. 7. Elsevier New York, 1977.

HARIPRASAD, T.; VIDHYAGARAN, G.; SEENU, K.; THIRUMALAI, C. Software complexity analysis using halstead metrics. In: *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, IEEE, 2017, p. 1109–1113.

HATTORI, L. P. Análise probabilística de impacto de mudanças baseada em históricos de mudanças de software. *Master's thesis, Universidade Federal de Campina Grande, Campina Grande-Brazil*, 2008.

HUANG, L.; SONG, Y.-T. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In: *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, IEEE, 2007, p. 374–384.

JUNIOR, H. D. S. C.; DO PRADO, A. F.; ARAÚJO, M. A. P. Complexity tool: Uma ferramenta para medir complexidade ciclomática de métodos java-complexity tool: a tool for measuring cyclomatic complexity in java methods. *Multiverso: Revista Eletrônica do Campus Juiz de Fora-IF Sudeste MG*, v. 1, n. 1, p. 66–76, 2016.

DE LA VARA, J. L.; BORG, M.; WNUK, K.; MOONEN, L. An industrial survey of safety evidence change impact analysis practice. *IEEE Transactions on Software Engineering*, v. 42, n. 12, p. 1095–1117, 2016.

LEE, C.; YOUN, C. Dynamic impact analysis method using use-case and uml models on object-oriented analysis. *Journal of KIISE*, v. 43, n. 10, p. 1104–1114, 2016.

LEE, M. L.; ET AL. *Change impact analysis of object-oriented software*. George Mason University Virginia, 1998.

LEHNERT, S. A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep*, 2011.

LI, B.; SUN, X.; LEUNG, H.; ZHANG, S. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, v. 23, n. 8, p. 613–646, 2013.

MAIA, M. C. O. Técnica híbrica de análise de impacto para sistemas orientados a objetos. *Campina Grande*, 2009.

MARTINS, R. M. *Técnicas de visualização para avaliação e melhoria de qualidade de software livre e aberto*. Tese de Doutoramento, Tese (Doutorado)—Instituto de Ciências Matemáticas e de Computação-ICMC-USP . . . , 2012.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, , n. 4, p. 308–320, 1976.

MERCER, E.; PERSON, S.; RUNGTA, N. Computing and visualizing the impact of change with java pathfinder extensions. *ACM SIGSOFT Software Engineering Notes*, v. 37, n. 6, p. 1–5, 2012.

MERINO, L.; GHAFARI, M.; ANSLOW, C.; NIERSTRASZ, O. A systematic literature review of software visualization evaluation. *Journal of systems and software*, v. 144, p. 165–180, 2018.

MOHAMAD, R. N. *A change impact analysis approach using visualization method*. Tese de Doutoramento, Universiti Teknologi Malaysia, 2010.

OCHODEK, M.; HEBIG, R.; MEDING, W.; FROST, G.; STARON, M. Recognizing lines of code violating company-specific coding guidelines using machine learning. *Empirical Software Engineering*, v. 25, n. 1, p. 220–265, 2020.

ORSO, A.; APIWATTANAPONG, T.; LAW, J.; ROTHERMEL, G.; HARROLD, M. J. An empirical comparison of dynamic impact analysis algorithms. In: *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 2004, p. 491–500.

OWENS, M. *The definitive guide to sqlite*. Apress, 2006.

PFLEEGER, S. L.; BOHNER, S. A. A framework for software maintenance metrics. In: *Software Maintenance, 1990, Proceedings., Conference on*, IEEE, 1990, p. 320–327.

PRESSMAN, R.; MAXIM, B. *Software engineering: A practitioner's approach*. 8 ed. New York, NY, USA: McGraw-Hill, Inc., 2014.

ROLFSNES, T.; DI ALESIO, S.; BEHJATI, R.; MOONEN, L.; BINKLEY, D. W. Generalizing the analysis of evolutionary coupling for software change impact analysis. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2016, p. 201–212.

RUNGTA, N.; PERSON, S.; BRANCHAUD, J. A change impact analysis to characterize evolving program behaviors. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, p. 109–118.

SCHEIBEL, W.; TRAPP, M.; LIMBERGER, D.; DÖLLNER, J. A taxonomy of treemap visualization techniques. In: *Proc. International Conference on Information Visualization Theory and Applications*, 2020.

SEREF, B.; TANRIOVER, O. Software code maintainability: Aliterature review. *International Journal of Software Engineering & Applications (IJSEA)*, v. 7, n. 3, 2016.

SHNEIDERMAN, B.; WATTENBERG, M. Ordered treemap layouts. In: *IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.*, IEEE, 2001, p. 73–78.

SI, I.; TANVEER, B.; VOLLMER, A. M.; BRAUN, S.; ALI, N. B. An evaluation of effort estimation supported by change impact analysis in agile software development. *Journal of Software: Evolution and Process*, v. 31, n. 5, p. e2165, 2019.

STREČANSKỲ, P.; CHREN, S.; ROSSI, B. Comparing maintainability index, sig method, and sqale for technical debt identification. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, p. 121–124.

SUN, X.; LI, B.; TAO, C.; WEN, W.; ZHANG, S. Change impact analysis based on a taxonomy of change types. In: *2010 IEEE 34th Annual Computer Software and Applications Conference*, IEEE, 2010, p. 373–382.

TURVER, R. J.; MUNRO, M. An early impact analysis technique for software maintenance. *Journal of Software: Evolution and Process*, v. 6, n. 1, p. 35–52, 1994.

VELARDE, H.; SANTIESTEBAN, C.; GARCIA, A.; CASILLAS, J. Software development effort estimation based-on multiple classifier system and lines of code. *IEEE Latin America Transactions*, v. 14, n. 8, p. 3907–3913, 2016.

WALSHAW, C. A multilevel algorithm for force-directed graph drawing. In: *International Symposium on Graph Drawing*, Springer, 2000, p. 171–182.

WELKER, K. D.; OMAN, P. W.; ATKINSON, G. G. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, v. 9, n. 3, p. 127–159, 1997.

WILKERSON, J. W. A software change impact analysis taxonomy. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, IEEE, 2012, p. 625–628.

ZHU, N. Q. *Data visualization with d3. js cookbook*. Packt Publishing Ltd, 2013.