UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

ANÁLISE PROBABILÍSTICA DE IMPACTO DE MUDANÇAS BASEADA EM HISTÓRICOS DE MUDANÇAS DO SOFTWARE

LILE PALMA HATTORI

CAMPINA GRANDE – PB
FEVEREIRO DE 2008

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Informática

Análise probabilística de impacto de mudanças baseada em históricos de mudanças do software

Lile Palma Hattori

Dissertação submetida à Coordenação de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero

(Orientador)

Jorge César Abrantes de Figueiredo

(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

H366a
2008 Hattori, Lile Palma
Análise probabilística de impacto de mudanças baseada em históricos de mudanças de software /Lile Palma Hattori. — Campina Grande: 2008.
120f. : il

Dissertação (Mestrado em Informática) – Universidade Federal de Campina Grande, Centro Engenharia Elétrica e Informática.
Referências.
Orientadores: Dr. Dalton Dario Serey Guerrero e Dr. Jorge César Abrantes de Fiqueiredo.

1. Técnicas de Estimativa. 2. Análise de Impacto de Mudanças. I. Título.

CDU 004.413.5(043)

"ANÁLISE PROBABILÍSTICA DE IMPACTO DE MUDANÇAS BASEADA EM HISTÓRICOS DE MUDANÇAS DE SOFTWARE"

LILE PALMA HATTORI

DISSERTAÇÃO APROVADA EM 27.02.2008

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO , D.Sc
Orientador

PROF. MARCUS COSTA SAMPAIO, Dr.
Examinador

PROF. MANOEL GOMES DE MENDONÇA NETO, Ph.D
Examinador

CAMPINA GRANDE – PB

Resumo

Para atender à demanda contínua por evolução, sistemas de software estão sujeitos a constantes mudanças. Tanto durante o processo de desenvolvimento, quanto na manutenção de um sistema, há a necessidade de incorporar mudanças, que podem ter diversas causas como, por exemplo, uma alteração nos requisitos ou correção de erros de design. Para incorporar uma mudança no sistema, é preciso compreendê-lo e prever quais conseqüências essa mudança pode causar. Essa atividade é chamada de análise de impacto de mudanças. Informações extraídas da análise de impacto podem ser utilizadas para planejar e realizar mudanças, bem como rastrear os efeitos que elas podem causar. A análise de impacto pode ser aplicada após a realização de uma mudança para avaliar seus efeitos através das técnicas dinâmicas, baseadas nos rastros de execução do software. Porém, sua abordagem mais proativa é quando prevê os impactos de uma mudança antes de sua realização, através de técnicas estáticas. No entanto, ao avaliar uma mudança, as técnicas de análise de impacto estáticas identificam um grande número de impactos que não ocorrem de fato, chamados de falso-positivos. Neste trabalho, propomos e avaliamos uma técnica de análise de impacto probabilística, que identifica os impactos de uma mudanças antes de sua implementação e atribui probabilidades de ocorrência aos impactos através do uso de informação sobre o histórico de mudanças do software analisado. Assim, ela permite a ordenação dos impactos por probabilidade e a conseqüente redução do número de falso-positivos. Para avaliar nossa técnica, redefinimos duas métricas da recuperação de informação, chamadas precisão e revocação, que mensuram o número de falso-positivos e falso-negativos gerados. Falso-negativos são impactos que ocorrem de fato, mas não são identificados pela técnica. Os resultados da avaliação comprovam que nossa técnica foi capaz de reduzir o número de falso-positivos gerados e, assim, aumentar a precisão da análise de impacto realizada antes da implementação.

Abstract

In order to attend the continuous need for evolution, software systems are subjected to constant changes. Both during software development and maintenance there is a need to incorporate changes, which can be caused by, for example, requirements change or design errors. To incorporate a change, we need to comprehend the system and foresee the consequences of that change to the system. This activity is called change impact analysis. Information extracted from impact analysis can be used to plan and execute a change, as well as to track the effect caused by it. Impact analysis can be applied after the implementation of a change to evaluate its effects through dynamic techniques based on execution traces. However, impact analysis is more proactive when applied before the implementation of a change to predict its impacts through static techniques. When assessing a change, static analysis techniques identify a great amount of impacts that do not occur in practice, these are called false-positives. In this work, we propose and evaluate a probabilistic impact analysis technique that identifies the impacts of a change before its implementation and assigns probability of occurrence to the impacts based on the software change history. Thus, with this technique, the impacts can be ordered by the probability of occurrence and the number of false-positives can be reduced. To evaluate our technique, we redefine two measures from information retrieval, called precision and recall, to assess the number of false-positives and false-negatives produced. False-negatives are impacts that occur in practice, but are not identified by the impact analysis technique. The results show that our probabilistic technique was able to reduce the number of false-positives produced and, consequently, increase precision of the impact analysis applied before the implementation of a change.

Agradecimentos

Agradeço primeiramente a meus pais Lígia e Likiso, eternos incentivadores de minha formação.

A Lauro, meu amado companheiro.

A meus familiares. Em especial a Michelle, minha prima querida.

A meus orientadores Dalton e Jorge, por me guiarem nessa longa e tortuosa caminhada.

A meus queridos "escravinhos" Jemerson e João, por me ajudarem a concretizar idéias, implementando as ferramentas deste trabalho.

Aos professores Francilene, Francisco (matemática), Jacques e Marcus Sampaio, que contribuíram com críticas e sugestões para a realização deste trabalho.

A Fernando e Gilson, que colaboraram neste trabalho através da disciplina de mineração de dados.

Aos amigos que fiz em Campina Grande: Amanda, Ana Emília, Andréa, Ayla, Bel, Cássio, Cheyenne, Degas, Eliane, Fábio, Fireman, Karina, Karol, Isabella, Milena, Mirna, Nigini, Pablo, Paulo, Renata, Roberto, Rodrigo, Sidney, Zé de Guga. A Antônio e André, meus sobrinhos adotivos. À Socorro. Àqueles que esqueci de pôr o nome, mas que também são importantes para mim.

Aos meus amigos que estão distante, mas continuam guardados no meu coração: Guinho, Ju, Lalinha, Laurinho, Leca, Liz, Mano e Paulinhas.

Aos profissionais que me ajudaram a superar minhas dores de coluna: Dr. Jaguaracy, Dr. Lin, Dr. Pimenta, Dra. Shini, Clara, Jaqueline, Júnior, Renatinha e Zoraide.

Aos profissionais da CPMBraxis, que me ajudaram a tornar este trabalho útil na prática: Ana Paula Braun, André Lima, Carol Passos e Geovani Santagelo.

Aos patrocinadores deste trabalho: CAPES, CNPq, FINEP e CPMBraxis.

Aos professores e funcionários da COPIN e do DSC.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

A progressiva exigência e a dependência dos usuários por sistemas de software têm aumentado a demanda por evoluí-los e modificá-los de forma rápida e confiável. Como conseqüência, as atualizações de software tornaram-se experiência comum aos usuários de computador. Além disso, a crescente dependência organizacional por software de qualidade resultou no reconhecimento da necessidade de uma co-evolução contínua dos negócios e do software [Lehman e Ramil 2001].

Para atender à demanda contínua por evolução, sistemas de software estão sujeitos a constantes mudanças. Tanto durante o processo de desenvolvimento, quanto durante a manutenção de um sistema, há a necessidade de incorporar mudanças. Do ponto de vista do engenheiro de software, a necessidade de uma mudança pode surgir em diferentes momentos do ciclo de desenvolvimento para atender a diferentes propósitos. Alterações dos requisitos, redesign, identificação e correção de erros de design durante a codificação e correções de erros apontados por testes são exemplos de situações que criam a necessidade de mudanças.

Segundo Bohner, uma mudança no código do sistema compreende: 1) entender como a mudança afeta o software; 2) implementar a mudança proposta; e 3) testar o sistema alterado [Bohner 2002]. Tipicamente, a primeira etapa envolve especificar a mudança e avaliar como ela afetará o software, se for implementada. A atividade de avaliar como a mudança afeta o software é denominada de *análise de impacto de mudanças*.

Embora, originalmente, tenha sido criada para analisar mudanças antes de sua implementação, a análise de impacto é empregada em diferentes momentos no processo de mudança

e com diferentes finalidades. Quando realizada antes de a mudança ocorrer, pode facilitar algumas atividades dentro do processo de desenvolvimento. Por exemplo, pode ajudar o engenheiro do software a compreender melhor o sistema em relação à mudança. Além disso, permite avaliar diferentes formas de realizar uma modificação, para que o engenheiro escolha a que cause menor impacto ao sistema. Nesse contexto, a análise de impacto pode auxiliar a estimativa de custo e tempo necessários para realizar mudanças. Quando a análise de impacto é realizada após a implementação da mudança, ela ajuda no processo de testes de regressão, identificando os testes que necessitam ser re-executados para verificar a integridade do software. No entanto, sua abordagem mais proativa é quando prevê os impactos de uma mudança antes de sua realização. Neste trabalho enfocamos a atividade de análise de impacto realizada antes da implementação da mudança.

Ao se propor uma mudança, é necessário planejar sua implementação, o que inclui estimar precisamente seu escopo (tamanho e complexidade) e o custo de sua implementação. Tradicionalmente, engenheiros de software determinam os efeitos de uma mudança após examinar, manualmente, o código e a documentação do projeto, de acordo com o conhecimento que eles possuem do sistema em questão [Arnold e Bohner 1996]. Essa abordagem pode ser suficiente para programas pequenos. Porém, quando os sistemas são grandes, o engenheiro dificilmente é capaz de armazenar conhecimento suficiente sobre seus subsistemas e suas relações de dependência. Assim, o risco de erro na estimativa pode ser grande a ponto de causar prejuízos.

O problema da análise de impacto é que se pode errar na identificação dos elementos do sistema possivelmente impactados de duas formas diferentes: 1) apontar como impactados elementos que não são afetados quando a mudança é realizada; e 2) deixar de identificar elementos que são impactados pela mudança. Os elementos falsamente identificados como impactados, denominados falso-positivos, podem provocar estimativas de escopo e custo muito maiores do que deveriam ser, desencorajando a implementação da mudança e reduzindo a competitividade da empresa. Analogamente, elementos que não são apontados, mas que sofrem impactos, são chamados de falso-negativos. Estes, por sua vez, podem produzir estimativas abaixo do custo real, causando prejuízos para a empresa que desenvolve o software.

Ferramentas e técnicas têm sido propostas para melhorar o processo de análise de im-

pacto, tanto em relação à automatização, quanto à acurácia da identificação dos impactos. Elas podem ser divididas em duas categorias: análise de impacto estática ou dinâmica. As técnicas estáticas analisam o código-fonte do software, enquanto as técnicas dinâmicas analisam rastros (*traces*) de execução do sistema.

A análise de impacto estática [Arnold e Bohner 1996; Lee, Offutt e Alexander 2000; Ryder e Tip 2001; Turver e Malcolm 1994] baseia-se em técnicas de fatiamento de código ou no fechamento transitivo em grafos de chamadas. As técnicas de fatiamento geralmente analisam os grafos de fluxo de dados e de fluxo de controle do sistema em nível de sentença. São ditas abrangentes porque devem considerar todas as possibilidades de execução do sistema. As técnicas que utilizam o fechamento transitivo em grafos de chamadas analisam o sistema em nível de rotinas ao invés de sentenças. Há situações, contudo, em que as técnicas estáticas tendem a produzir falso-positivos. Por exemplo, em situações de polimorfismo, quando uma classe chama um método de um subtipo através de seu supertipo, é possível que não se tenha a informação de qual subtipo será instanciado, pois ele pode ser passado como parâmetro para a classe. Nesse caso, as técnicas estáticas devem ter uma abordagem conservadora incluindo todos os subtipos como elementos possivelmente impactados, se o supertipo for impactado.

As técnicas de análise dinâmica [Korel e Laski 1990; Orso et al. 2004; Apiwattanapong, Orso e Harrold 2005; Breech, Tegtmeyer e Pollock 2005] baseiam-se nas execuções do sistema e geram menos falso-positivos do que as abordagens estáticas. O problema do polimorfismo, por exemplo, não existe nas abordagens dinâmicas, pois elas obtêm a informação de qual subtipo está sendo usado pela classe através do rastro de execução. Por outro lado, as técnicas dinâmicas dependem dos valores de entrada utilizados nas execuções e podem falhar na identificação de alguns impactos, produzindo falso-negativos. Semelhante ao que acontece com testes, nunca se pode afirmar que todas as possíveis execuções do sistema foram consideradas.

Na prática, técnicas estáticas e dinâmicas atendem a diferentes propósitos. Técnicas estáticas são geralmente aplicadas antes da implementação de uma mudança e as dinâmicas são geralmente utilizadas após sua implementação. Para que a análise de impacto permita o planejamento de uma mudança e a estimativa de seus custos, ela precisa ser realizada antes de a mudança ocorrer e fornecer resultados precisos. Contudo, as técnicas de análise de

impacto estática atuais geram muitos falso-positivos e não existem técnicas dinâmicas (mais precisas) que sejam aplicáveis antes da implementação da mudança.

Várias abordagens vêm sendo propostas para reduzir o número de falso-positivos em análise estática. Por exemplo, Lee e Offut propuseram ponderar os impactos a partir das características dos tipos de mudanças e como cada tipo pode influenciar outras partes do sistema [Lee 1999]. Em paralelo, a comunidade de engenharia de software vem descobrindo a relevância de analisar informações históricas para melhorar a compreensão da evolução do software [Madhavji, Fernandez-Ramil e Perry 2006]. Técnicas de mineração de repositórios vêm sendo empregadas para descobrir relações de dependências entre entidades que não podem ser obtidas por análise estática de código [Fluri, Gall e Pinzger 2005; Ying, Ng e Chu-Carroll 2004]. É neste contexto que se situa o trabalho apresentado nesta dissertação. Mais especificamente, investigamos a hipótese de que

> *é possível aumentar a precisão da análise de impacto realizada antes da implementação de uma mudança, através do uso de informações históricas do comportamento das mudanças do sistema.*

Para isso, propomos uma técnica de análise de impacto probabilística. Ela combina a análise de impacto estática baseada em fechamento transitivo em grafos de chamadas com análise histórica da evolução do software em relação ao comportamento das mudanças.

Apesar de existirem trabalhos recentes sobre análise estática [Lee 1999; Breech, Tegtmeyer e Pollock 2006], não encontramos ferramentas disponíveis que implementassem alguma das técnicas apresentadas nesses trabalhos. Tampouco, encontramos algoritmos suficientemente detalhados que nos permitissem implementá-los.

Dessa forma, propomos uma técnica de análise de impacto estática para sistemas orientados a objetos que se baseia em grafos de chamadas e realiza tratamentos específicos para características inerentes à orientação a objetos, como herança e polimorfismo. A técnica proposta foi implementada em uma ferramenta, chamada Impala, utilizando a linguagem Java. Assim, os algoritmos de análise de impacto foram propostos para analisar, especificamente, sistemas orientados a objetos desenvolvidos em Java. Por essa razão, algumas características da orientação a objetos que não existem em Java, como herança múltipla, não foram consideradas.

Após a criação da técnica de análise de impacto estática, refinamos seus resultados, através da atribuição de probabilidades aos elementos identificados como possivelmente impactados. A probabilidade de um elemento ser impactado é calculada através da aplicação de um algoritmo de mineração de dados ao histórico de mudanças do sistema a ser modificado. O histórico de mudanças do sistema é extraído de seu repositório (por exemplo, CVS) e apenas informações de mudanças na estrutura do código são utilizadas para ponderar os impactos. O algoritmo de mineração que propomos para calcular a probabilidade de impacto é baseado no teorema de Bayes [Meyer 1970].

Para avaliar a técnica probabilística, redefinimos duas métricas da recuperação de informação para mensurar a quantidade de falso-positivos e falso-negativos gerados pela técnica: precisão (*precision*) — que mede a quantidade de falso-positivos gerados; e revocação (*recall*) — que avalia a quantidade de falso-negativos gerados. A proposta destas métricas é parte da contribuição deste trabalho e foi necessária porque nenhum dos trabalhos avaliados usa métricas claras e objetivas para mensurar a precisão das técnicas de análise de impacto.

A avaliação da técnica probabilística foi realizada em duas etapas. Na primeira, avaliamos somente a técnica de análise de impacto estática. Para isso, realizamos experimentos que envolveram a aplicação da técnica de análise estática em três softwares distintos. Esses experimentos foram executados seguindo uma metodologia definida. Na segunda etapa, avaliamos a técnica probabilística. Para isso, realizamos um estudo de caso para aplicar a técnica em dois dos softwares utilizados na etapa anterior. Os resultados demonstraram que a análise probabilística obteve maior precisão e menor revocação em relação à estática. Embora as perdas na revocação tenham variado entre 20 e 30 pontos percentuais (em nível de entidade), os ganhos na precisão variaram entre 30 a 55 pontos percentuais (em nível de entidade), o que justifica a aplicação da técnica.

Para finalizar a segunda etapa, realizamos uma avaliação subjetiva sobre as probabilidades de impacto sugeridas pela análise probabilística. A partir dela, concluímos que, nos softwares avaliados, os impactos muito prováveis de ocorrer aparecem com o valor de probabilidade dentro do intervalo [50%, 100%] e os impactos poucos prováveis de acontecer têm probabilidade dentro do intervalo [0%, 50%). Dessa forma, considerando os cenários avaliados, concluímos que a análise probabilística melhora os resultados da análise de impacto realizada antes da implementação e acrescenta informação útil ao engenheiro de software,

bem como fornece resultados mais precisos para uma análise de estimativa de custos.

Este trabalho foi composto por uma série de atividades, das quais, podemos destacar as mais importantes. A elaboração da técnica de análise estática envolveu, além da pesquisa, a implementação da ferramenta Impala. Em seguida, propomos as métricas precisão e revocação, publicadas através do artigo intitulado "On the precision and accuracy of impact analysis techniques" [Hattori et al. 2008], que reporta os resultados da primeira etapa da avaliação. Em paralelo, investigamos e desenvolvemos uma solução para a extração das informações históricas do software. Essa segunda solução gerou a ferramenta Impala *Plug-in*. Em seguida, pesquisamos como combinar os resultados da análise de impacto estática às informações coletadas pelo Impala *Plug-in* e qual modelo probabilístico se adequava melhor à proposta de atribuir probabilidades de impacto aos resultados fornecidos pelo Impala. Após a escolha do teorema de Bayes, implementamos um algoritmo de mineração, que chamamos de ImpalaMiner. Por fim, realizamos a avaliação da técnica proposta.

O restante da dissertação está organizado da seguinte forma. No capítulo 2, expomos o quadro teórico relevante para o entendimento da solução. Os principais conceitos e técnicas de análise de impacto existentes são apresentados, bem como as características inerentes à orientação a objetos que dificultam o processo de análise de impacto. Uma visão geral sobre evolução de software é descrita, juntamente ao estado da arte referente à exploração das informações históricas do software. Uma rápida revisão sobre probabilidade condicional e teorema de Bayes é realizada. O capítulo 3 descreve a solução proposta de análise de impacto estática para sistemas orientados a objetos. Apresentamos técnica de atribuição de probabilidade de impacto a partir do histórico do sistema no capítulo 4. No capítulo 5, retomamos e avaliamos a hipótese deste trabalho, através da realização de experimentos. O capítulo 6 apresenta uma análise comparativa entre a solução proposta nesta dissertação e outras existentes. Por fim, no capítulo 7, estão as considerações finais e sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

A técnica de análise de impacto probabilística, proposta neste trabalho, envolve os conceitos básicos de análise de impacto e aspectos explorados pela evolução de software. A análise de impacto estática permite a localização de potenciais entidades impactadas ao se propor uma mudança no software. Informações do histórico de mudanças desse software são úteis para ponderar os possíveis impactos. Dessa forma, estipulamos probabilidades de cada impacto ocorrer de acordo com seu histórico de mudanças.

Neste capítulo, abordamos os fundamentos teóricos sobre os quais este trabalho foi desenvolvido. Inicialmente, apresentamos os conceitos básicos de análise de impacto de mudanças. Discutimos sobre as diferenças entre as abordagens estática e dinâmica e ponderamos as vantagens e desvantagens de cada uma. Em seguida, apresentamos alguns conceitos da orientação a objetos que necessitam de atenção especial ao se propor uma técnica de análise de impacto de mudanças. Discutimos, também, conceitos sobre a evolução de software e suas vertentes: teórica e prática. Enfatizamos a importância de estudar como o software evolui ao longo do tempo com o intuito de melhorá-lo, evitar seu envelhecimento, facilitar a manutenção, propor refatoração, dentre outros. Para complementar, apresentamos soluções que exploram as informações históricas das mudanças de um software através da aplicação de técnicas de mineração de dados a sistemas de controle de versão. Essas soluções posicionam as mudanças no centro do processo de desenvolvimento para compreender como elas ocorreram e prever futuras mudanças. Por fim, relembramos as definições de probabilidade condicional e do teorema de Bayes. Este último será aplicado para ponderar os impactos do software através do uso das informações históricas das mudanças de um software.

2.1 Definição de Análise de Impacto

Apesar de a análise de impacto de mudanças ser praticada há algumas décadas, não existe uma definição comumente usada pelos pesquisadores. Turver e Munro definem a análise de impacto como "a avaliação das conseqüências de uma mudança feita em um módulo, nos outros módulos do sistema" [Turver e Malcolm 1994]. Pfleeger e Bohner definem como "a avaliação dos riscos associados a uma mudança, incluindo a estimativa de recursos, esforço e cronograma" [Pfleeger e Bohner 1990]. Eles afirmam que a análise de impacto permite não apenas avaliar as conseqüências de uma mudança planejada, como também, ponderar entre diferentes abordagens de uma mesma mudança. Usando diferentes definições, todos os supracitados conectam a análise de impacto à avaliação das estimativas de custo, tempo ou esforço relacionadas às mudanças [Lee 1999; Apiwattanapong 2007].

A definição adotada neste trabalho é a de Ajila, que considera a análise de impacto como "uma atividade que identifica as potenciais conseqüências (efeitos-colaterais) de uma mudança e estimar o quê, no código, precisa ser modificado para realizá-la" [Ajila 1995].

A análise de impacto é uma atividade que pode estar presente no processo de desenvolvimento e na manutenção de um sistema. A figura 2.1 ilustra o ciclo de desenvolvimento e em quais fases a análise de impacto é empregada. Nessa figura, consideramos que a análise de impacto é empregada quando se propõe uma mudança no código do software. Essa mudança pode ter sido solicitada por uma alteração nos requisitos, por um erro de projeto identificado durante a fase de codificação ou de testes.

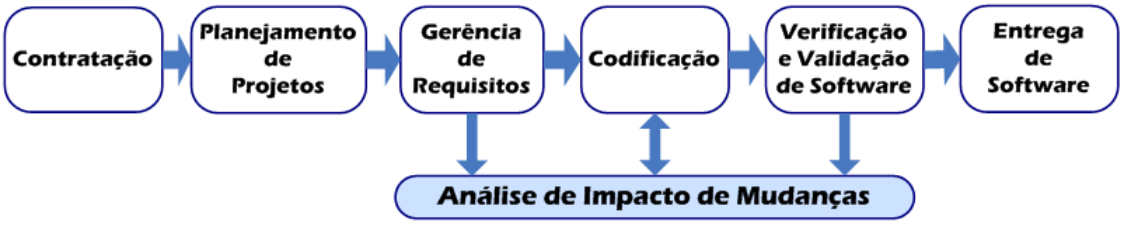


Figura 2.1: Emprego da análise de impacto dentro do ciclo de desenvolvimento

A análise de impacto deve determinar o escopo de uma mudança requisitada para permitir o planejamento de recursos e cronograma [Moreton 1996]. Ela pode ser dividida em quatro fases. Na primeira, o escopo da mudança é determinado através da verificação da documentação e do software em si. Na segunda, estima-se o custo e tempo necessários para realizar

a mudança, de acordo com características como o tamanho e a complexidade do software, processo de desenvolvimento e características da equipe de desenvolvimento. A terceira fase corresponde à analise dos custos e benefícios da mudança. Na quarta, os custos e benefícios são reportados aos gerentes ou responsáveis por autorizar ou rejeitar a mudança. A técnica desenvolvida neste trabalho permite aprimorar a primeira fase, contribuindo na obtenção de estimativas mais precisas na segunda fase.

2.2 Áreas da Análise de Impacto

As duas principais áreas da análise de impacto são a análise de dependência e a análise de rastreabilidade. Essas áreas complementares abordam a análise de impacto de perspectivas diferentes e têm suas respectivas vantagens para aumentar o potencial de identificação dos impactos no software. A figura 2.2 ilustra essas duas áreas, que podem ser combinadas para aumentar a precisão da análise de impacto.

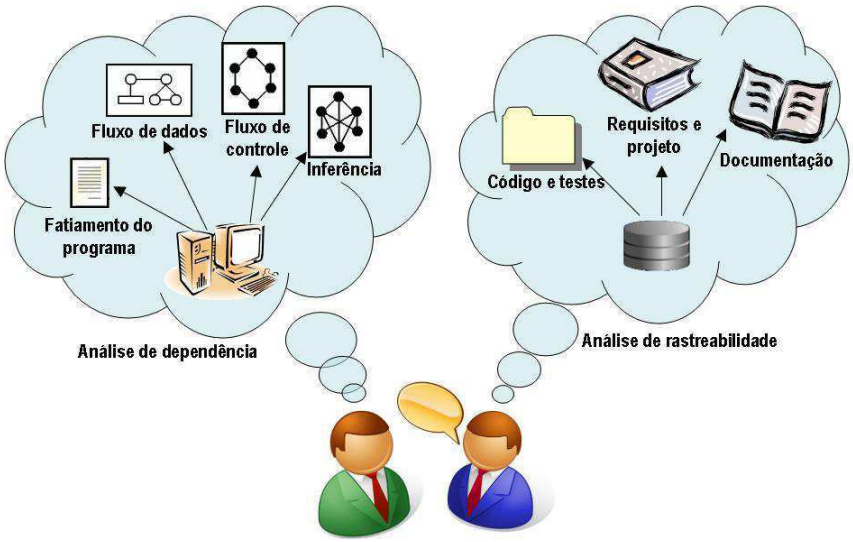


Figura 2.2: Principais áreas da análise de impacto de mudanças

Análise de Rastreabilidade envolve o exame das relações de dependência entre todos os artefatos de um sistema. Ela pode relacionar, por exemplo, requisitos com os componentes do design associado.

Análise de Dependência, foco deste trabalho, envolve o exame detalhado das relações de dependência entre as entidades do programa. Provê uma avaliação detalhada das dependên-

cias de baixo nível no código, mas faz pouco para outros artefatos, como documentação, requisitos ou testes. Então, quando se refere à análise de dependência, fala-se, tipicamente, da análise do código fonte [Arnold e Bohner 1996].

A análise de dependência analisa as relações de dependência entre controle, dados e os componentes de um programa. Mapeando para um programa em Java, os dados são os atributos e as variáveis, o controle considera todas as alternativas de fluxo em cada método das classes e componentes podem ser os módulos do programa.

Técnicas de análise de dependência incluem, dentre outras, a análise de fluxo de dados (*data flow analysis*), a análise de fluxo de controle (*control flow analysis*), análise baseada nos grafos de chamadas, fatiamento de programa (*program slicing*), análise de testes cobertura e referência cruzada. Neste trabalho, será utilizado o fechamento transitivo em grafos de chamadas.

As técnicas de análise de impacto baseadas na análise de dependência [Lee 1999; Li e Offutt 1996; Lee, Offutt e Alexander 2000; Ryder e Tip 2001; Turver e Malcolm 1994] procuram avaliar os efeitos de uma mudança, em relação a dependências dinâmicas de entidades do programa, identificando dependências sintáticas que podem sinalizar a presença de tais dependências dinâmicas [Law e Rothermel 2003].

2.3 Abordagens: Estática e Dinâmica

Existem, atualmente, duas abordagens de análise de impacto: estática e dinâmica. A análise estática avalia o código fonte do software para identificar os impactos possivelmente causados por uma mudança no software. Ela é empregada, geralmente, antes da implementação da mudança, para estimar os esforços necessários na implementação ou ajudar o especialista a escolher a opção menos custosa dentre as possíveis opções de mudança.

As técnicas de análise de impacto dinâmicas são aplicadas após a mudança ser implementada no software [Orso et al. 2004; Apiwattanapong, Orso e Harrold 2005]. Elas coletam os rastros das execuções do software e analisam quais entidades chamam as entidades modificadas para adicioná-las ao conjunto de impactos. Por isso, quanto maior for a cobertura das possíveis execuções, maior é a precisão do resultado obtido.

A seguir, detalhamos as análises de impacto estática e dinâmica. Realizamos, também,

uma análise comparativa entre as duas abordagens, apontando vantagens e desvantagens de adotar cada uma.

2.3.1 Análise de Impacto Estática

As técnicas estáticas analisam o código-fonte do software para identificar os elementos possivelmente impactados por uma mudança. Elas normalmente mapeiam o sistema para um conjunto de grafos, no qual os nós representam as entidades e as arestas são as dependências entre as entidades. Esses grafos podem ser construídos em nível de método ou em um nível mais detalhado, o de sentenças.

Análises estáticas em nível de sentenças usam o fatiamento de programas para segmentar o sistema em grafos de fluxo de controle e dados. Elas são consideradas de granularidade fina, pois realizam uma análise detalhada e o resultado inclui todas as dependências de uma mudança em nível de sentença. Essa é uma abordagem que gera um grande número de falso-positivos, pois considera todas as possíveis entradas e todos os possíveis comportamentos do sistema [Breech, Tegtmeyer e Pollock 2005]. Lembramos que falso-positivo é definido como o possível impacto identificado pela técnica de análise de impacto, que não ocorre de fato.

Já as análises estáticas em nível de método, ou design, usam a técnica de fechamento transitivo em grafos de chamadas, ou técnicas semelhantes para identificar as chamadas possivelmente impactadas por uma mudança. São consideradas de granularidade grossa, pois analisam o software com menor nível de detalhe. Por exemplo, o conteúdo e a semântica de um método não são analisados nessas abordagens.

Para a técnica de fechamento transitivo, assume-se que uma mudança em um procedimento p de um programa P pode impactar, potencialmente, qualquer nó alcançável a partir de p no grafo de chamadas G de P. O fechamento transitivo de G calcula as relações de possíveis impactos entre todos os procedimentos de P alcançáveis a partir de p.

Para exemplificar, o código 1 mostra um conjunto de cinco classes: A, B, C, D e E. Cada classe contém um método que faz uma chamada a um método de outra classe. A figura 2.3 apresenta o grafo de chamadas G referente às cinco classes do código 1. Se `E.método5` for alterado, o conjunto de elementos possivelmente impactados é `{A.método1, B.método2, C.método3, D.método4, E.método5}`. O

mesmo conjunto de impactos se repete caso `B.método2`, `C.método3` ou `D.método4` for modificado. No entanto, se `A.método1` for alterado, o conjunto de elementos impactados é somente `{A.método1}`, pois nenhum outro método chama `A.método1`.

Código 1 Pseudocódigo para exemplificar o cálculo do fechamento transitivo de um programa

```
Classe A                    Classe D
   método1                     método4
      B.método2                   E.método5

Classe B                    Classe E
   método2                     método5
     C.método3                   B.método2

Classe C
   método3
      D.método4
```

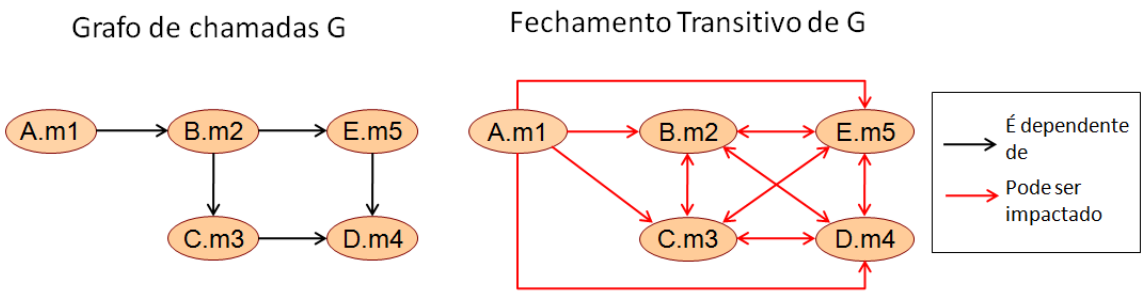


Figura 2.3: Grafo de chamadas e fechamento transitivo das classes A, B, C, D, e E

2.3.2 Análise de Impacto Dinâmica

As técnicas dinâmicas analisam rastros de execução de programas para identificar os elementos impactados por uma mudança. Há duas categorias de análise dinâmica: *online* e *offline*. Elas se assemelham em relação aos dados coletados, à forma de analisar os impactos de uma mudança e aos resultados encontrados. A diferença é que a *offline* coleta os dados dos rastros para analisá-los após o fim da execução, enquanto a *online* analisa os rastros durante a execução do programa [Breech, Tegtmeyer e Pollock 2005].

O conjunto dinâmico de elementos impactados é definido como o subconjunto das entidades do programa que é afetado por uma mudança durante, pelo menos, uma de suas

execuções [Apiwattanapong, Orso e Harrold 2005]. A análise de impacto dinâmica computa, aproximadamente, o conjunto dinâmico de elementos impactados, pois ela depende do conjunto de execuções e da entrada para cada execução. Como não é possível assegurar que a análise dinâmica capture rastros para todas as possíveis execuções de um programa e todos os possíveis valores de entrada, sabe-se que técnicas dinâmicas geram falso-negativos. Lembramos que falso-negativo é definido como o elemento impactado que não é identificado pela técnica de análise de impacto.

Para exemplificar como seria a análise dinâmica para o mesmo código 1 da análise estática, supõe-se duas execuções. Na primeira, A.método1 chama B.método2, que chama C.método3. O rastro de execução encontra-se na figura 2.4(a). Na segunda execução E.método5 chama B.método2. O rastro de execução pode ser visto na figura 2.4(b). Observa-se que, apesar de o grafo de chamadas G, da figura 2.3, afirmar que C.método3 é dependente de D.método4, essa dependência não aparece nos rastros de execução analisados. Nesse exemplo, o conjunto dinâmico de entidades impactadas I_1 ={A.método1, B.método2, C.método3} e I_2 ={E.método5, B.método2}. O conjunto final I ={A.método1, B.método2, C.método3, E.método5} é a união de I_1 e I_2.

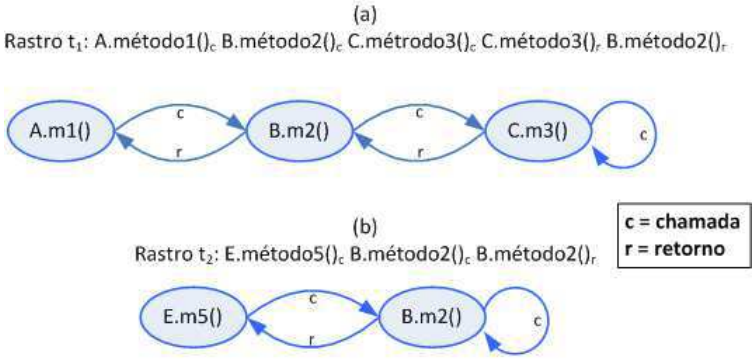


Figura 2.4: Rastros de execução usados pela análise de impacto dinâmica

2.3.3 Relações Entre as Abordagens de Análise de Impacto

A análise de impacto estática é considerada uma abordagem conservativa, pois identifica todos os possíveis impactos de uma mudança, para todas as possíveis execuções de um programa. No entanto, seu conservantismo pode gerar um grande número de falso-positivos, visto que o conjunto de possíveis impactos pode incluir até 90% do software analisado [Api-

wattanapong, Orso e Harrold 2005]. Nesses casos, o resultado da análise de impacto é de pouca utilidade, porque não ajuda nas estimativas de custo e tempo, tampouco identifica partes do software para análise criteriosa do engenheiro de software.

Já a análise de impacto dinâmica obtém resultados mais precisos do que a estática, visto que esses são calculados a partir de possíveis execuções do programa [Orso et al. 2004]. Entretanto, essa também é a razão para que a análise dinâmica gere um grande número de falso-negativos. Esse número está estritamente ligado à cobertura das execuções e da qualidade dos testes; quanto mais execuções diferentes forem analisadas, menos falso-negativos existirão.

O diagrama de Venn da figura 2.5 ilustra as relações entre os conjuntos de impactos retornados pelas abordagens estática e dinâmica e o conjunto mínimo de elementos impactados. O conjunto máximo de elementos impactados representa todo o sistema e o conjunto de mudanças engloba apenas as mudanças propostas. Os elementos realmente impactados estão representados pelo conjunto mínimo de elementos impactados.

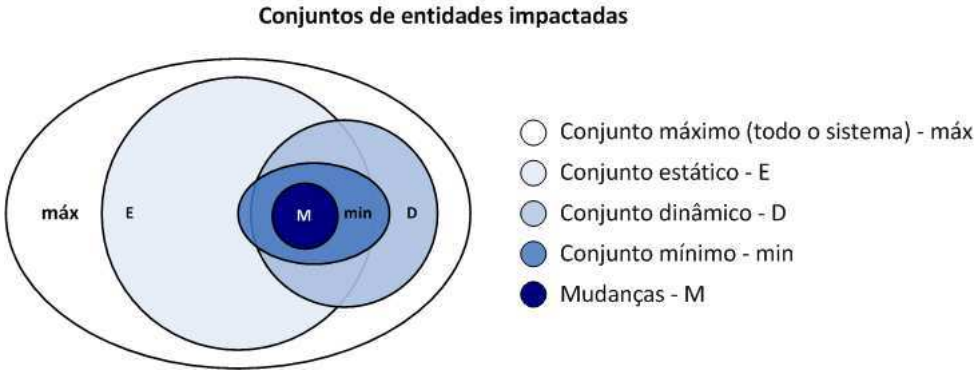


Figura 2.5: Relações entre conjuntos de impactos retornados pelas abordagens estática e dinâmica

Observa-se que os conjuntos estático e dinâmico geram tanto falso-positivos quanto falso-negativos, ilustrados pelas áreas que estão fora das interseções. O número de falso-positivos do conjunto estático é grande. Um exemplo da explosão dos falso-positivos é um caso que envolva herança, pois todos os subtipos de um supertipo deverão entrar no conjunto, mesmo que apenas um subtipo seja realmente usado. Os falso-negativos da análise estática são poucos. É o caso do *very late binding*, que pode ser exemplificada por reflexão na linguagem Java [Forman, Forman e Forman 2004].

Para a análise dinâmica, a quantidade de falso-positivos é bastante reduzida em relação à estática. Por isso, ela é considerada mais precisa. Para exemplificar um caso de falso-positivo, observa-se a figura 2.6. Nessa abordagem, todos os elementos que foram chamados antes da mudança são possíveis impactados, mesmo que não existam dependências entre eles. Na figura 2.6, C é o elemento modificado e {M, A, B, C} é o conjunto de elementos possivelmente impactados. No entanto, A não depende sintaticamente de C, portanto não é um elemento impactado por C. No caso dos falso-negativos, qualquer possível execução que for ignorada pela análise dinâmica é um caso de elemento impactado que não foi acusado pela técnica. Não se sabe ao certo qual é o tamanho desse conjunto, pois as abordagens de análise dinâmica não apresentam medidas ou métricas para quantificá-lo.

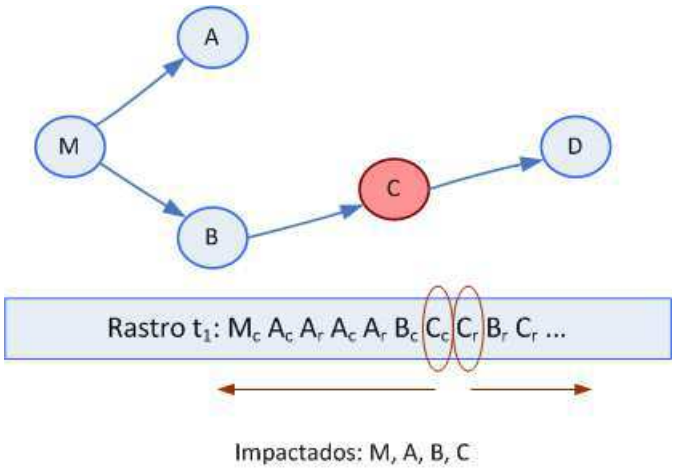


Figura 2.6: Exemplo de rastro da análise dinâmica com falso-negativo

A noção dos limites dos conjuntos estático e dinâmico será importante para compreender a técnica proposta nesta dissertação, bem como as medidas usadas para avaliar os resultados obtidos.

2.4 Análise de Impacto em Sistemas Orientados a Objetos

A análise de impacto para programas estruturados, amplamente pesquisada no passado [Pfleeger e Bohner 1990; Turver e Malcolm 1994; Ajila 1995; Moreton 1996; Arnold e Bohner 1993; Arnold e Bohner 1996], envolve soluções que constroem e analisam os grafos de fluxo de controle e de fluxo de dados e usam técnicas de fatiamento de programa. Essas soluções são eficazes, pois analisam tanto as estruturas de dados quanto o controle em

nível de procedimento e de sentenças. Contudo, elas não podem ser aplicadas diretamente a sistemas orientados a objetos.

Conceitualmente, uma classe é um tipo dado abstrato equipado com uma possível implementação parcial [Meyer 1997]. O objeto, instância de uma classe, tem, além das estruturas de dados e dos métodos, estado. Dessa forma, além das dependências de dados e de controle, existe a dependência de comportamento. Esta constitui um fator complicador, que obriga uma reestruturação das análises de dependências largamente usadas para sistemas estruturados, pois elas abordam somente as duas primeiras dependências.

O problema principal da análise de impacto em sistemas orientados a objetos é a complexidade inerente à interação entre os objetos. Os relacionamentos entre classes podem formar um grafo extremamente complexo, pois cada objeto de uma classe tem o potencial de interagir com todos os outros objetos do sistema. Essa rede de relacionamentos complexa torna difíceis as tarefas de antecipar e identificar os efeitos colaterais de uma mudança.

Além disso, características inerentes a sistemas orientados a objetos, como o polimorfismo, a ligação dinâmica, a herança e o encapsulamento contribuem diretamente para tornar a tarefa da análise de impacto ainda mais dispendiosa. O polimorfismo e a ligação dinâmica permitem que objetos diferentes sejam usados através de suas interfaces, o que será definido somente em tempo de execução. Com o encapsulamento, os objetos ocultam suas implementações, restringindo a forma de observá-los; apenas seus métodos públicos são visíveis. Já com a herança, uma mudança na superclasse pode ou não afetar a subclasse, o que complica a tarefa de analisar o impacto quando uma superclasse é modificada.

A seguir, discorremos sobre as possíveis dependências entre os objetos.

2.4.1 Dependências em Sistemas Orientados a Objetos

Linguagens orientadas a objetos complicam o processo de análise de dependência por permitirem a ligação dinâmica de chamadas a métodos. Uma variável pode referir-se a um objeto de qualquer classe que compartilhe um tipo (uma interface). Assim, o polimorfismo permite que uma chamada de método seja executada de várias formas diferentes, o que só será determinado em tempo de execução.

Uma dependência direta em um sistema de software é, informalmente, uma relação direta entre duas entidades do sistema $X \rightarrow Y$, tal que uma modificação em X pode causar efeito

colateral em Y [Wilde e Huitt 1996]. Uma dependência indireta é uma relação de duas ou mais dependências diretas. Por exemplo, $X \rightarrow Y$ e $Y \rightarrow Z$, então $X \rightarrow Z$ é uma dependência indireta na qual X pode causar efeito colateral em Z indiretamente.

Abaixo, adaptamos a classificação das dependências orientadas a objetos propostas por Lee [Lee 1999] para direcioná-la à linguagem de programação usada neste trabalho: Java.

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.

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.

Essa classificação das dependências é referência para a classificação dos tipos de mudanças apresentada na seção 3.2.

2.5 Evolução de Software

A evolução de software reflete-se numa necessidade intrínseca pelo desenvolvimento e a manutenção contínua de sistemas, para endereçar uma aplicação, ou resolver um problema no domínio do mundo real [Lehman e Ramil 2001]. O termo evolução no contexto de software tem duas abordagens distintas. Ambas reconhecem que a evolução é implementada em um processo que disciplina e controla as mudanças de um sistema e os procedimentos associados a elas [Madhavji, Fernandez-Ramil e Perry 2006]. No entanto, elas diferem na interpretação do termo evolução. A primeira abordagem, a verbal, a estuda como atividade a ser implementada, controlada, gerenciada, e a segunda, a nominal, a considera como um fenômeno.

A visão verbal, busca o desenvolvimento ou o aperfeiçoamento dos processos, procedimentos, métodos e ferramentas de planejamento, implementação, gerenciamento, suporte e controle da evolução dos processos de desenvolvimento. É o **como** da evolução que procura alcançar melhor eficiência em fatores como confiança e custo. Trabalhos nesse sentido vêm sendo largamente apresentados e discutidos pela comunidade acadêmica.

A chamada de nominal trata a evolução de software como um fenômeno a ser estudado e entendido. Investiga o **quê** e o **porquê** da evolução, que busca o melhor entendimento para fortalecer a habilidade de, por exemplo, inspirar, guiar e disciplinar processos, métodos e ferramentas de desenvolvimento. Apesar de menos explorada, essa abordagem também é importante quanto a primeira, pois quanto melhor o entendimento do processo evolutivo do software, maior a capacidade de adequar métodos e ferramentas para seu planejamento, gerenciamento e implementação.

Neste trabalho, investigaremos o software como fenômeno, através da observação de sua evolução, para guiar o processo de análise de impacto. No entanto, não discutiremos os aspectos teóricos da evolução de software [Lehman 1969; Lehman 1974; Lehman 1978; Lehman 1980; Lehman 1980; Pfleeger 1998; Lehman e Ramil 2001; Madhavji, Fernandez-

Ramil e Perry 2006]. O nosso foco é a evolução de software na prática, posicionando as mudanças no centro do processo de desenvolvimento [Fluri e Gall 2006] e a investigando. Através dessa abordagem, podemos entender como e porque as mudanças ocorrem, quais partes do sistema são fortemente acopladas, quais módulos mostram-se historicamente problemáticos e, finalmente, usar esse conhecimento para guiar a análise de impacto.

A seguir, discorremos sobre o estado da arte em investigação da evolução do software através de seu histórico de mudanças. Retomamos essa discussão no capítulo 6, quando comparamos os trabalhos apresentados a seguir com nossa solução.

2.5.1 Evolução de Software Baseada no Histórico de Mudanças

De acordo com Gall et al, uma forma eficiente de prever e evitar os efeitos negativos do envelhecimento de software é posicionar as mudanças no centro do processo de desenvolvimento do software [Fluri e Gall 2006]. Ao se fazer isso, percebe-se que o sistema de controle de versão (ex.: CVS) é peça fundamental para o acompanhamento da evolução do software. A partir das informações armazenadas nele, é possível verificar a freqüência de mudanças no sistema ou em parte dele, detectar acoplamento de código, acompanhar a evolução/degradação de sua arquitetura e, conseqüentemente, melhorar a qualidade do software produzido.

Nesse sentido, uma série de trabalhos recentes vem explorando os dados dos repositórios de projetos que usam controle de versão, através da diferenciação de código, e contribuindo com pesquisas que têm como foco principal entender como o software evolui ao longo do seu processo de desenvolvimento.

Provavelmente, o algoritmo mais conhecido e empregado para diferenciar código é o GNU diff [Myers 1986], um utilitário para comparação de arquivos de texto. Para o GNU diff o software é apenas um conjunto de arquivos de texto. Portanto, ao comparar duas versões de um trecho de código (dois arquivos de texto), o GNU diff é completamente inconsciente das mudanças lógicas e estruturais do software.

Mais recentemente, para sistemas de software orientados a objetos, apareceram algoritmos de diferenciação baseados em XML como, por exemplo, o DeltaXML [Ltd. 2006], produto lançado em 2001 para controle de mudanças. Outra abordagem é a de diferenciar mudanças através de uma classificação de palavras-chave a serem usadas nos comentários do controle de versão e posterior análise da freqüência da aparição de cada uma [Mockus e

Votta 2000].

Mais recentemente, outras abordagens de diferenciação e comparação de códigos estão sendo usadas para explorar informações do repositório. É o exemplo de Zimmermann et al, Gall et al e Ying et al, que utilizam diferentes algoritmos de mineração de dados para minerar informações históricas das mudanças com finalidades semelhantes [Zimmermann et al. 2004; Fluri, Gall e Pinzger 2005; Ying 2003]. Zimmermann et al aplicam técnicas de mineração de dados no histórico do CVS para guiar os programadores ao longo de mudanças. A filosofia do eROSE, *plug-in* do Eclipse criado por eles, é proporcionar ao programador o equivalente à função da Amazon.com que diz "x% dos clientes que compraram este produto também compraram...". No caso de programador seria "programadores que mudaram a função f() também mudaram a função g() em x% das vezes". Ying et al recomendam a inclusão de arquivos a uma tarefa de modificação após minerar o histórico de mudanças e inferir que tarefas semelhantes foram realizadas no passado. Gall et al propõem um algoritmo de granularidade fina para extrair mudanças estruturais do histórico de um sistema de controle de versão.

A seguir, discorreremos sobre cada uma das abordagens, individualmente. Descreveremos, também, conceitos introduzidos por essas abordagens que são importantes para a nossa solução.

Predizendo mudanças no código fonte através da mineração do histórico de revisões

Esse é um trabalho de mestrado desenvolvido por Annie Ying na *University of British Columbia* [Ying 2003; Ying, Ng e Chu-Carroll 2004]. O trabalho consiste em minerar padrões de mudanças – arquivos que mudaram juntos com certa freqüência – dos arquivos armazenados no sistema de controle de versão. A partir da obtenção desses padrões, recomenda-se a revisão de arquivos que obedecem aos padrões enquanto um desenvolvedor realiza uma tarefa de modificação. A motivação para aplicar a mineração de dados para predizer mudanças consiste no fato de, utilizando análises estática e dinâmica de código, ser difícil de identificar dependências entre módulos escritos em diferentes linguagens de programação. Ainda, é difícil de mensurar a força destas dependências, quando são identificadas.

A abordagem consiste em três etapas: 1 – processamento dos dados; 2 – mineração

das regras de associação; 3 – aplicação de *query* para recomendação de mudanças. Uma definição importante, utilizada no processamento dos dados, é a de conjunto de mudanças atômicas. Ela consiste em mudanças de arquivos que foram registrados (*checked in'*) pelo mesmo autor, com o mesmo comentário, dentro da mesma janela de tempo. Sistemas de controle de versão mais atuais, como o Subversion [Pilato 2004], incluem mudanças com essas características de transações atômicas. No entanto, o CVS [Foundation 2006], sistema de controle de versão largamente usado na comunidade, não inclui esses conceitos. Como os sistemas analisados por Ying estavam armazenados no CVS, a atividade de encontrar os conjuntos de mudanças atômicas fez parte da primeira etapa da abordagem.

O algoritmo de mineração usado, o FP-Tree [Han, Pei e Yin 2000], é classificado como de associação e extrai conjuntos de itens que ocorrem com freqüência suficiente entre os conjuntos de mudanças atômicas. Nesse contexto, padrões de mudanças se referem aos arquivos que tendem a mudar juntos. Após a aplicação do algoritmo, é possível consultar a base de dados minerada para obter informações de quais arquivos devem ser alterados com a mudança de certo arquivo.

As vantagens dessa abordagem são que ela independe da linguagem de programação do sistema analisado e é capaz de encontrar relações de dependências entre componentes implementados em linguagens de programação distintas. Como desvantagem, re-escritas e refatorações significantes do código podem afetar a solução, porque essas mudanças afetam os padrões que são computados, impossibilitando o rastreamento de código similar entre elas. Outra questão é que a abordagem não é capaz de distinguir entre mudanças de infra-estrutura de mudanças estruturais no software.

Minerando histórico de versões para guiar mudanças no software

Esse trabalho tem grande semelhança com o anterior. Também aplica um algoritmo de mineração de dados, com o intuito de encontrar regras de associação do tipo: programadores que mudaram função A também mudaram função B [Zimmermann et al. 2004]. A grande diferença é que sua solução é um *plug-in* do Eclipse, chamado eROSE, que se integra completamente ao ambiente de programação, facilitando o uso pelo programador. A figura 2.7 demonstra o momento posterior a uma mudança feita por um programador no código fonte. O eROSE sugere funções onde, em transações similares no passado, outras mudanças tam-

bém foram feitas.

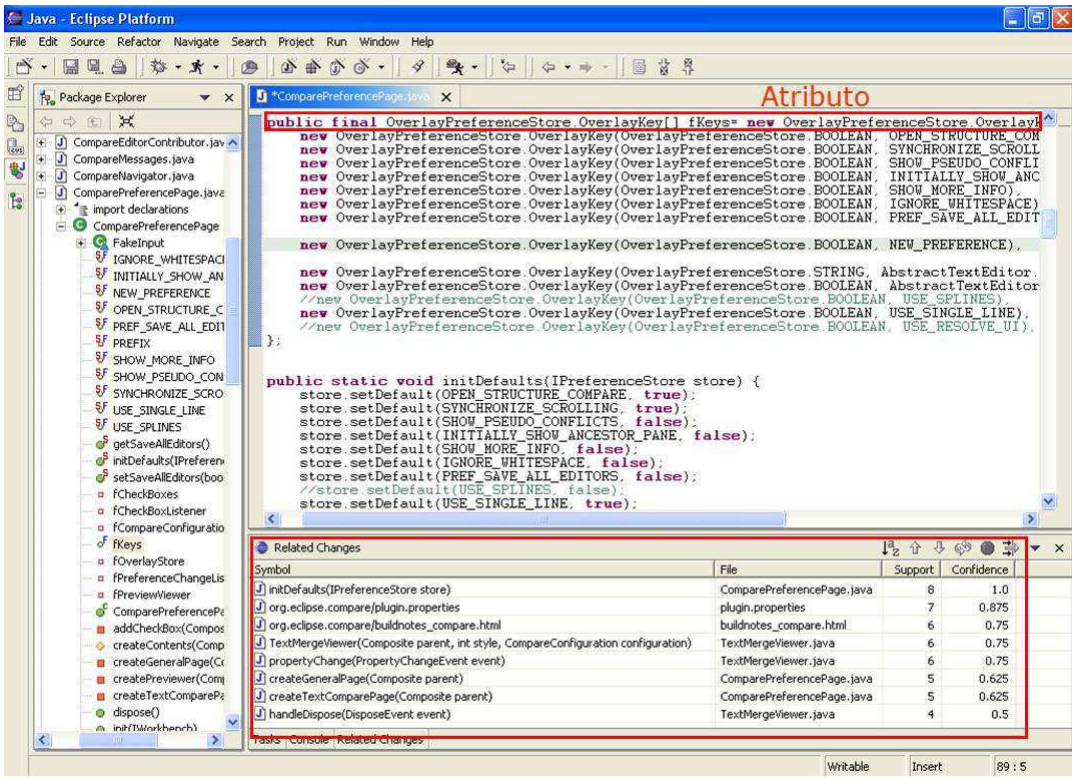


Figura 2.7: Perspectiva do Eclipse com o plug-in eROSE

O CVS é, também, o sistema de controle versão acessado por essa solução. Como visto na solução anterior, o CVS apresenta algumas limitações. Assim, o eROSE precisa preparar as informações coletadas do CVS para que sejam usadas como entrada do algoritmo de mineração. Zimmermann et al propuseram um processo de pré-processamento que define o conceito de transação [Zimmermann e Weißgerber 2004]. Este se assemelha à definição de conjunto de mudanças atômicas vista anteriormente. Além disso, eles propõem duas formas de se calcular uma transação.

1. Janela fixa de tempo: restringe a duração máxima de cada transação. O intervalo de tempo sempre começa no primeiro *checkin*. A janela de tempo usada é de 3 a 4 minutos.

2. Janela deslizante de tempo: restringe o intervalo máximo entre os *checkins* subseqüentes de uma transação. Essa abordagem pode reconhecer transações que demoram mais tempo para completar do que o tempo da janela fixa de tempo. A janela de tempo geralmente usada é de 200 segundos.

Após o pré-processamento, o eROSE utiliza o algoritmo Apriori [Agrawal e Srikant 1998] para computar as regras de associação. O Apriori computa as regras de acordo com o suporte mínimo e a confiança mínima. Ao exibir as regras de associação para o programador, o eROSE as ordena de acordo com os valores do suporte e da confiança. O suporte da regra informa quantas vezes a regra aconteceu no passado em relação ao total de registros. A confiança é o cálculo da razão entre quantidade de vezes que o lado direito da regra ocorreu pela quantidade de vezes que o lado esquerdo da regra ocorreu. Por exemplo, "classe A ocorreu 5 vezes => classe B ocorreu 4 vezes" num universo de 10 registros; então o suporte é igual a 0,4 e a confiança é igual 0,8.

Análise de acoplamento de mudanças com granularidade fina

Esse trabalho é desenvolvido pelo grupo *s.e.a.l. – Software Evolution and Architecture lab*, liderado pelo professor Harald Gall [Gall, Jazayeri e Krajewski 2003; Fluri, Gall e Pinzger 2005]. A solução proposta por eles é parecida com as duas anteriores, mas o propósito é diferente. O objetivo principal é investigar as dependências e correlações entre módulos e entidades de um sistema orientado a objetos ao longo do seu ciclo de vida. Métricas para mensurar as dependências de um sistema, como as de acoplamento e coesão, são aplicadas no software em um determinado momento. Gall et al propõem uma forma de medir acoplamentos entre entidades que surgem ao longo do tempo. Para isso, definem acoplamento lógico como classes que mudaram juntas muitas vezes ao longo do tempo de vida do software [Gall, Jazayeri e Krajewski 2003]. Esse termo é chamado, posteriormente, de acoplamento de mudanças [Fluri, Gall e Pinzger 2005].

A grande diferença entre a técnica aplicada por Ying et al e a técnica de mineração de dados aplicada neste é que este associa mudanças em nível estrutural. Em outras palavras, ele compara a estrutura das classes de um software em Java em nível de métodos e atributos. Assim, acoplamentos de mudanças que não são causados por mudanças estruturais, como mudança no termo de licença, são eliminados. A comparação estrutural entre as classes é feita através do uso de duas bibliotecas: *Java Development Tool* (`org.eclipse.jdt`), que provê a classe `JavaStructureCreator` para estruturar o código fonte em declarações de *import*, classes, interfaces, construtores e atributos; e a *Compare* (`org.eclipse.compare`) capaz de comparar duas estruturas do tipo

`JavaStructureCreator`.

Os resultados do estudo de caso, realizado por eles, apontam que mais da metade das transações no CVS não foram causadas por mudanças estruturais. Ou seja, mais da metade dos acoplamentos de mudanças não estão ligados a mudanças estruturais nas classes. No entanto, essa abordagem não considera acoplamentos entre entidades ou módulos implementados em linguagens de programação diferentes, como é o caso da primeira abordagem apresentada.

Análise Comparativa das Abordagens

A tabela 2.1 apresenta um comparativo entre as soluções apresentadas nesta seção. As duas primeiras abordagens são independentes de linguagem de programação e capazes de descobrir relações de dependências entre subsistemas desacoplados estruturalmente. Elas são bastante úteis quando o projeto em análise é composto por *frameworks*, *web services* e múltiplas linguagens de programação. Em contrapartida, a terceira solução é capaz de realizar uma análise de granularidade mais fina, que envolve aspectos estruturais do software. Para isso, ela é atrelada à linguagem de programação Java.

Tabela 2.1: Comparativo das ferramentas

Abordagens	**Ling. de programação**	**Ferramenta disponível**	**Integrada a uma IDE**	**Técnica de mineração**	**Algoritmo**
Predição de mudanças no código através da mineração do histórico de revisões	Independente	Não	Não	Regras de associação	FP-Tree
Minerando histórico de versões para guiar mudanças no software	Independente	Sim	Eclipse	Regras de associação	Apriori
Análise de acoplamento de mudanças com granularidade fina	Java	Não	Eclipse	Não informada (*clustering*)	Não informado

Não há um consenso entre as três soluções sobre qual técnica de mineração de dados (e qual algoritmo) é mais adequada ao problema. As duas primeiras utilizam dois algoritmos de associação, o clássico Apriori e o FP-Tree. A terceira solução não informa, explicitamente,

a técnica de mineração de dados utilizada. Porém, pela explicação, supomos que ela aplica a técnica de *clustering*. O consenso não existe, porque não há uma solução ótima de mineração de dados que generalize a descoberta de informações do histórico de mudanças para todas as possíveis aplicações. Há, sim, uma técnica que se adéqua melhor a cada solução.

É importante destacar que alguns conceitos gerais referentes à mineração de sistemas de controle de versão, introduzidos por essas soluções, são utilizados na solução deste trabalho. A definição de transação com janela deslizante, a comparação estrutural do código em Java e o conceito de acoplamento de mudanças serão retomados no capítulo 4 desta dissertação.

2.6 Teorema de Bayes

O teorema de Bayes foi utilizado no algoritmo que calcula as probabilidades de impacto de uma entidade considerando suas informações históricas de mudança. Ele calcula a probabilidade condicional de um evento a posteriori ocorrer dado um evento a priori. Escolhemos o emprego do teorema de Bayes, pois ele permite o cálculo da probabilidade de uma entidade ser impactada de acordo com seus registros de impactos no passado. Quanto maior o número de registros passados de impacto em relação ao conjunto de mudanças, maior será a probabilidade de seu impacto a posteriori.

Para enunciar o teorema de Bayes, necessitamos relembrar as definições de probabilidade condicional e de partições de um espaço amostral. Primeiramente, introduzimos a equação de probabilidade condicional, que calcula a probabilidade de um evento B ocorrer dado que o evento A ocorre.

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \tag{2.1}$$

dado que $P(A) > 0$.

Para exemplificar o uso da probabilidade condicional, suponhamos que uma empresa que vende picolés está fazendo a seguinte promoção: encontre um palito premiado e ganhe um picolé de graça. Para cada lote de 1000 unidades, apenas 3 picolés têm palitos premiados. Qual é a probabilidade de uma pessoa encontrar 2 palitos premiados seguidos?

Definimos os dois eventos:

A = {primeiro palito é premiado}

B = {segundo palito é premiado}

A probabilidade de uma pessoa encontrar o primeiro palito premiado é igual a 3/1000. Portanto $P(A) = 0,3\%$. Para responder à pergunta acima, precisamos calcular a probabilidade de o evento B ocorrer logo após o evento A, portanto a probabilidade condicional $P(B|A)$. Como um palito premiado já foi encontrado, agora temos 999 picolés, dentre os quais, apenas 2 têm palitos premiados. Portanto $P(B|A) = 2/999$.

Sempre que calculamos a probabilidade condicional $P(B|A)$, estamos computando $P(B)$ em relação ao espaço amostral reduzido A (999), ao invés do espaço original (1000).

Agora, apresentamos a definição de partição de um espaço amostral.

Definição 2.1 (Partição de S) *Dizemos que os eventos $B_1, B_2, ..., B_k$ representam partições do espaço amostral S se:*

a) $B_i \cap B_j = \emptyset$ *para todo* $i \neq j$.

b) $\bigcup_{i=1}^{k} B_i = S$.

c) $P(B_i) > 0$ *para todo* i.

Portanto, sejam $B_1, ..., B_k$ partições do espaço amostral S e seja A um evento associado a S, o teorema de Bayes é definido por:

$$P(B_i|A) = \frac{P(A|B_i)P(B_i)}{\sum_{j=1}^{k} P(A|B_j)P(B_j)} \tag{2.2}$$

$P(B_i|A)$ é a probabilidade de B_i ocorrer a posteriori, dado que o evento A ocorreu a priori.

O teorema de Bayes é bastante aplicado na medicina, além da computação. Portanto, mostraremos um exemplo de sua aplicação para avaliar o resultado de um teste de droga [Bayes' theorem 2008]. Suponha que certo teste de droga é 99% sensível e 99% específico. Em outras palavras, o teste identificará corretamente um usuário da droga como testando positivo 99% das vezes e irá identificar corretamente um não-usuário como testando negativo 99% das vezes. Esse parece ser um teste relativamente preciso, mas o teorema de Bayes irá revelar uma falha em potencial.

Vamos assumir que uma corporação decide testar seus empregados por uso de ópio e 10% dos empregados usam a droga. Queremos saber a probabilidade de, dado um teste de droga positivo, um empregado seja mesmo um usuário de ópio. Seja D o evento de ser um usuário de droga e N um não-usuário. Seja + o evento de um teste de droga positivo. Precisamos obter os seguintes dados:

- $P(D)$ ou a probabilidade de um empregado ser um usuário de droga, independente de qualquer outra informação. P(D) é 0,1, visto que 10% dos empregados são usuários de droga. Essa é a probabilidade a priori de D.
- $P(N)$ ou a probabilidade de um empregado não ser um usuário de droga. Isto é, 1 – P(D) ou 0,9.
- $P(+|D)$ ou a probabilidade de um teste ser positivo, dado que o empregado é um usuário da droga: 0,99, dado que o teste é 99% preciso.
- $P(+|N)$ ou a probabilidade de um teste ser positivo, dado que o empregado não é um usuário da droga: 0,01, já que o teste irá produzir um falso-positivo para 1% dos não usuários.
- $P(+)$ ou a probabilidade de um evento com teste positivo, independente de outras informações: 0,108 ou 10,8%, que é encontrado adicionando a probabilidade de o teste produzir um resultado positivo verdadeiro em um evento de uso da droga (=99% $\times$ 10% = 9,9%) mais a probabilidade de o teste produzir um falso positivo em um evento de não uso da droga (= 1% $\times$ 90% = 0,9%). Essa é a probabilidade a priori de +.

Dada essa informação, podemos computar a probabilidade a posteriori $P(D|+)$ de um empregado que testou positivo ser realmente um usuário de droga:

$$\begin{aligned} P(D|+) &= \frac{P(+|D)P(D)}{P(+)} \\ &= \frac{P(+|D)P(D)}{P(+|D)P(D) + P(+|N)P(N)} \\ &= \frac{0,99 \times 0,1}{0,99 \times 0,1 + 0,01 \times 0,9} \\ &= 0,92 \end{aligned}$$

Interpretando o resultado, a probabilidade de um empregado que testou positivo realmente consumir ópio é de 92%, o que é um resultado elevado. Agora, suponha que apenas 0,5% dos empregados usam a droga. Portanto, $P(D)$ é 0,5% e $P(N)$ é 99,5%. Aplicando o teorema de Bayes aos novos valores:

$$\begin{aligned} P(D|+) &= \frac{P(+|D)P(D)}{P(+)} \\ &= \frac{P(+|D)P(D)}{P(+|D)P(D) + P(+|N)P(N)} \\ &= \frac{0,99 \times 0,005}{0,99 \times 0,005 + 0,01 \times 0,995} \\ &= 0,33 \end{aligned}$$

Apesar da alta precisão do teste, a probabilidade de um empregado que testou positivo realmente usar droga é, agora, de apenas 33%. Assim, para uma população reduzida a apenas 0,05% de usuários de ópio, é mais provável que um empregado que testou positivo não seja um usuário. Portanto, quanto mais rara é a condição para a qual estamos testando, maior é a percentagem de testes positivos que serão falso-positivos.

2.7 Considerações Finais

Os fundamentos teóricos apresentados nesse capítulo fornecem os meios necessários para o entendimento deste trabalho.

A técnica de análise de impacto probabilística envolve a análise de impacto de mudanças estática e a extração de informações históricas das mudanças no software. Informações do histórico de mudanças desse software serão úteis para ponderar os possíveis impactos. Dessa forma, estipularemos probabilidades de cada impacto ocorrer de acordo com suas mudanças no passado. Definições como a de transação com janela deslizante, a comparação estrutural do código em Java e o conceito de acoplamento de mudanças serão utilizadas na solução que atribui probabilidades de ocorrência de uma mudança. No entanto, não utilizaremos um algoritmo de mineração de dados já proposto anteriormente. Como nossa solução é bastante específica, propomos um algoritmo de associação baseado no teorema de Bayes que calcula a probabilidade de uma entidade ser impactada, dado um conjunto de mudanças proposto.

Capítulo 3

Técnica de Análise de Impacto Estática para Sistemas Orientados a Objetos

A análise de impacto probabilística que propomos é composta por uma técnica de análise de impacto estática e da análise do histórico de mudanças de um software. Neste capítulo apresentamos a técnica de análise de impacto estática para sistemas orientados a objetos. Ela permite localizar entidades que podem ser impactadas por um conjunto de mudanças proposto e que, possivelmente, também serão modificadas. Primeiramente, apresentamos uma visão geral da solução, que descreve as etapas da análise. Em seguida, detalhamos a técnica, explanando como é especificado o conjunto de mudanças proposto e, também, como é feita a extração da representação do software que será analisado. Por fim, os algoritmos de análise de impacto são apresentados e discutidos.

3.1 Visão Geral

A técnica proposta consiste em analisar o código de um sistema, extrair relações de dependência entre as entidades e navegar pelas relações, a partir das mudanças proposta, para identificar possíveis entidades impactadas. Em um nível mais detalhado, o algoritmo de análise de impacto recebe como entrada o código do software e uma especificação das mudanças propostas. Uma representação do software em termos de entidades e relacionamentos é produzida e utilizada pelos algoritmos de análise de impacto para buscar por entidades que possuem relacionamentos de dependência com as mudanças propostas. Como saída, ele

produz um conjunto de entidades possivelmente impactadas. Como prova de conceito, desenvolvemos a ferramenta Impala, que implementa a técnica descrita.

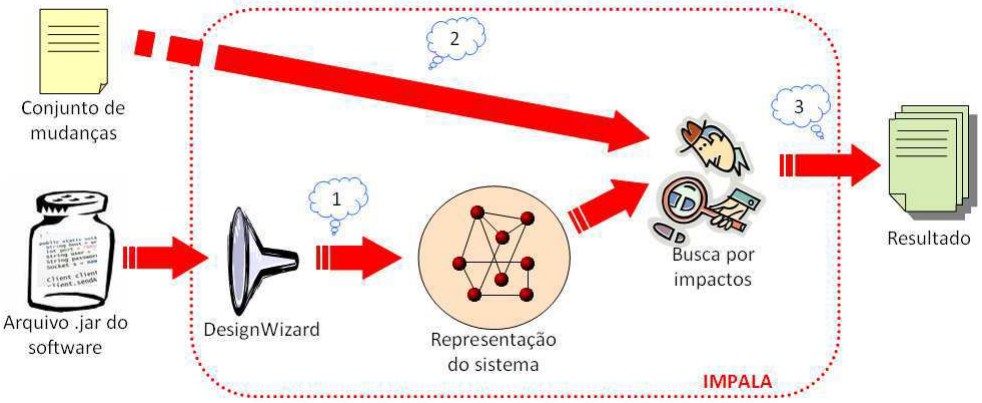


Figura 3.1: Visão geral do processo de análise de impacto

A figura 3.1 mostra uma visão geral do processo de análise de impacto. Detalhando um pouco a implementação do Impala, o sistema a ser analisado é recebido como um arquivo no formato *.jar*. Seu *bytecode* (código objeto gerado pelo compilador de Java) é lido e sua representação em forma de entidades e relacionamentos é produzida através do uso da biblioteca DesignWizard [Brunet e Guerrero 2007] (passo 1). Além disso, o Impala recebe a especificação do conjunto de mudanças, que consiste em uma lista de nomes de entidades a serem modificadas e o tipo de mudança que se quer realizar (passo 2). A identificação das entidades possivelmente impactadas é feita por algoritmos específicos, de acordo com cada tipo de mudança. Como saída, o Impala produz uma lista com todas as entidades do software que poderão ser impactadas pela mudança proposta (passo 3).

A seguir, as etapas do processo de análise de impacto serão detalhadas.

3.2 Conjunto de Mudanças

O conjunto de mudanças propostas, recebido como entrada pelo Impala, é representado por um arquivo de texto que segue uma especificação pré-definida de acordo com o tipo de mudança. A classificação dos tipos de mudanças, ilustrada na figura 3.2, é relacionada com os três tipos de entidades considerados neste trabalho: classe, método e atributo.

Há três tipos de mudanças: adição, remoção e alteração de entidade. A classificação

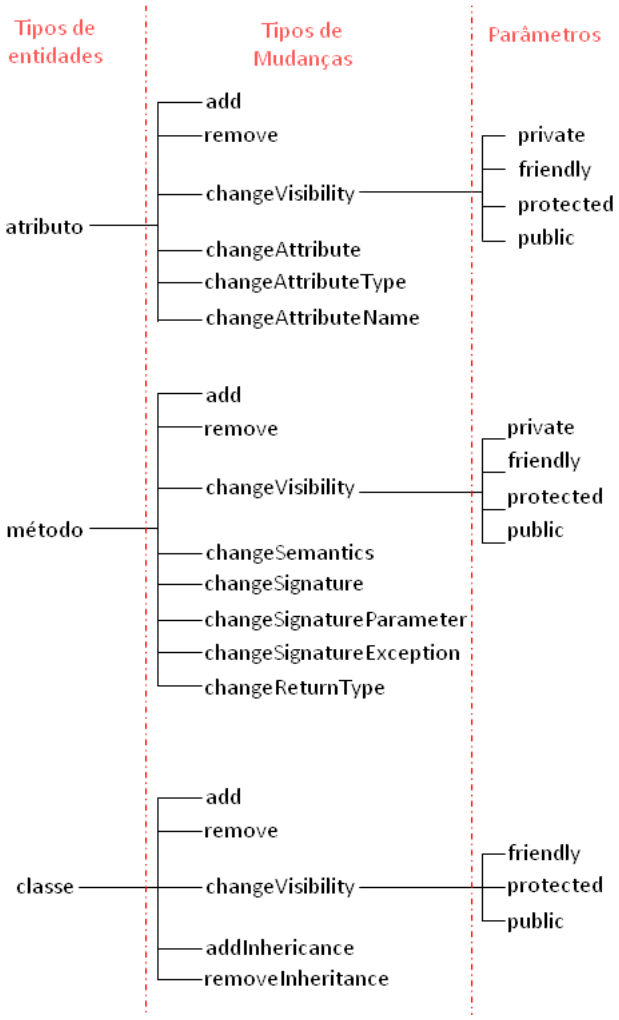


Figura 3.2: Classificação dos tipos de mudanças de acordo com cada tipo de entidade

proposta por este trabalho os esmiúça para determinar os possíveis tipos de mudanças estruturais em um sistema orientado a objetos. Essa classificação tem relação com as possíveis dependências entre as entidades de um sistema orientado a objetos, apresentada na seção 2.4.1. Por exemplo, as dependências de classe para classe que descrevem as relações de subclasse, superclasse e classe ancestral estão relacionadas aos tipos de mudança *addInheritance* e *removeInheritance*. As dependências que envolvem método relacionam-se com a grande maioria dos tipos de mudança, como é o caso do *changeSignature*, que envolve dependência de método para método.

A classificação dos tipos de mudanças estruturais é importante, também, para eliminar alterações em uma classe que não representam mudanças na estrutura do sistema. Essas alterações são, em sua grande maioria, de infra-estrutura, como: alteração em comentários (Javadocs), alteração na formatação da classe, alteração na licença. Essas e todas as outras alterações que não refletirem uma mudança na estrutura do sistema são ignoradas pela técnica de análise de impacto proposta.

Em nível de implementação, o conjunto de mudanças deve estar em um arquivo, no qual cada linha representa uma mudança proposta. Ainda, cada linha deve obedecer a um formato físico pré-definido:

```
<comando> <entidade> [<lista de argumentos>]
```

O `comando` informa o tipo específico de mudança que se quer realizar. A `entidade` deve ser descrita com seu nome completo. Por fim, a `lista de argumentos` contém informações auxiliares que servem de complemento ao comando, como os parâmetros de visibilidade do tipo *changeVisibility* (ver figura 3.2). A tabela 3.1 expõe a especificação completa dos possíveis tipos de mudanças.

A figura 3.3 exemplifica um arquivo de mudanças para o respectivo trecho de código. Nesse exemplo, existem três classes, das quais `Pessoa` é superclasse de `Estudante` e `Professor`. O arquivo de mudanças propõe as seguintes alterações: remoção da relação de herança entre as classes `Estudante` e `Pessoa`; remoção do atributo `Professor.sala`, mudança de visibilidade do atributo `Pessoa.nome` para protegido; e adição de uma nova classe, `Curso`.

Tabela 3.1: Especificação completa dos tipos de mudanças

Item	**Mudança**	**Descrição**
1	add pacote.Classe	adição de classe
2	add pacote.Classe.atributo	adição de atributo
3	add pacote.Classe.método()	adição de método/construtor
4	remove pacote.Classe	remoção de classe
5	remove pacote.Classe.atributo	remoção de atributo
6	remove pacote.Classe.<init>()	remoção de construtor/método
7	changeVisibility pacote.Classe to Public	mudança de visibilidade de classe (*public/protected/package*)
8	changeVisibility pacote.Classe.atributo to Protected	mudança de visibilidade de atributo (*public/protected/package/private*)
9	changeVisibility pacote.Classe.método(java.lang.String,int) to Friendly	idem ao item 8 para método
10	changeVisibility pacote.Classe.<init>(int) to Private	idem ao item 8 para construtor
11	addInheritance pacote.Subtype pacote2.Supertype	adição de herança à classe
12	removeInheritance pacote.Subtype pacote.Supertype	remoção de herança de uma classe
13	changeSignatureParameter pacote.Classe.<init>() to pacote.<init>(int)	adição/remoção/mudança de parâmetro de construtor/método
14	changeSignatureException pacote.Classe.metodo() throws Exception to pacote.Classe.metodo() throws AnotherException	adição/remoção/mudança de exceção lançada por método/construtor
15	changeSignature pacote.Classe.metodo() to pacote.Classe.metodo2()	mudança no nome do método/construtor
16	changeSemantics pacote.Classe.metodo()	mudança semântica no método/construtor
17	changeReturnType pacote.Classe.metodo(float) int to void	mudança no tipo de retorno do método
28	changeAttributeType pacote.Classe.atributo to int	mudança do tipo do atributo
19	changeAttributeName pacote.Classe.atributo to pacote.Classe.atributo2	mudança no nome do atributo
20	changeAttribute pacote.Classe.atributo	mudança no valor do atributo

```
class Pessoa {
   private String nome;
   Pessoa(String n){ nome = n; }
   public String getNome() { return nome;}
   public String toString(){ return nome; }
}

class Estudante extends Pessoa {
   Estudante(String nome){
      super(nome);
   }
}

class Professor extends Pessoa {
   private String departamento, sala;
   Professor(String nome, String d, String numero){
      super(nome); departamento = d;
      sala = numero; cursos = new HashSet();
   }
   public String toString(){
      return (super.toString() + ", sala é " +
      sala + " departamento é " + departamento);
   }
}
```

removeInheritance Estudante Professor

remove Professor.sala

changeVisibility Pessoa.nome to protected

add Curso

Arquivo de mudanças

Figura 3.3: Exemplo de um arquivo de mudanças

3.3 Extração do Sistema

A extração do sistema constitui-se da geração de um modelo do software a ser analisado a partir de seu código fonte. Esse modelo de representação do software é composto por conjuntos de entidades e relacionamentos. Uma entidade pode ser: classe, método, atributo. Um relacionamento conecta duas entidades e representa uma relação de dependência que a entidade chamadora possui com a chamada (ou o inverso). Essa geração do modelo do software é feita utilizando o DesignWizard.

Os relacionamentos são classificados de acordo com a classificação de dependências em sistemas orientados a objetos apresentada na seção 2.4.1. A lista completa dos tipos de relacionamentos considerados nesse trabalho e suas respectivas descrições encontra-se na tabela 3.2. A figura 3.4 ilustra os cinco diagramas de entidades e relacionamentos, que combinam as entidades duas a duas. Observa-se que, para alguns tipos de relacionamentos, também consideramos o relacionamento inverso. Por exemplo, `<atributo> is_declaredOn <classe>` é uma relação inversa a `<classe> contains <atributo>`, pois se um atributo é declarado em uma classe é porque ela o contém. Os relacionamentos inversos (direita para esquerda) estão destacados em cor vermelha na figura. Essa redundância é essencial para facilitar a navegabilidade do design durante o processo de análise de impacto.

Em nível de implementação, as entidades e os relacionamentos são organizados da

Tabela 3.2: Lista dos tipos de relacionamentos entre entidades do sistema. E. chamadora = entidade chamadora; E. chamada = entidade chamada

Relacionamento	**E. chamadora**	**E. chamada**	**Descrição**
`instanceOf`	atributo	classe	atributo é instância de uma classe
`contains`	classe	atributo, método	classe contém atributo/método
`extends`	classe	classe	subclasse herda características da superclasse
`implements`	classe	classe	subclasse implementa todos os métodos da interface
`getStatic`	método	atributo	método acessa atributo estático
`putStatic`	método	atributo	método modifica atributo estático
`getField`	método	atributo	método acessa atributo
`putField`	método	atributo	método modifica atributo
`invokeVirtual`	método	método	método de uma classe faz chamada a um método de outra classe
`invokeSpecial`	método	método	método chama o construtor ou método de uma superclasse ou um método privado
`invokeStatic`	método	método	método chama método estático
`invokeInterface`	método	método	método de uma classe chama método de uma interface
`is_accessedBy`	atributo	método	atributo é acessado ou modificado por um método, podendo esse tipo de relação classificada como: `getStatic`, `putStatic`, `getField` e `putField`
`is_invokedBy`	método	método	método é invocado pelo outro método. Pode ser classificada como: `invokeVirtual`, `invokeSpecial`, `invokeStatic` e `invokeInterface`
`is_superclass`	classe	classe	superclasse é pai de subclasse
`catch`	método	classe	método possui em seu corpo um *catch* de exceção do tipo de uma classe
`throws`	método	classe	método lança uma exceção do tipo de uma classe
`is_declaredOn`	método, atributo	classe	método/atributo é declarado em uma classe
`load`	método	classe	método carrega uma classe
`is_implementedBy`	classe	classe	superclasse é implementada por subclasse

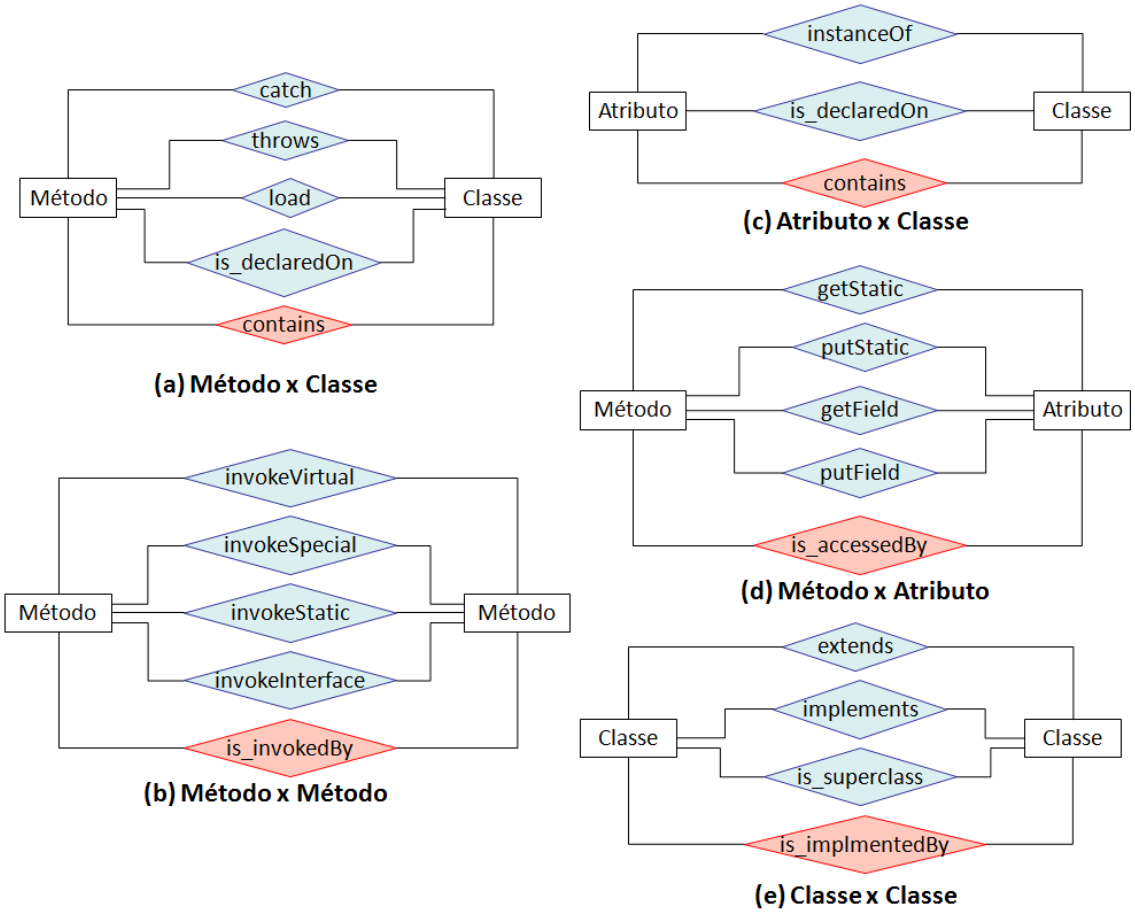


Figura 3.4: Diagramas de entidades e relacionamentos

seguinte forma: o Impala armazena um conjunto de entidades e cada entidade possui um conjunto de relacionamentos. Para facilitar o entendimento, o modelo pode ser visto como um grafo, no qual as entidades representam os nós e os relacionamentos representam os arcos. Cada entidade pode ser ligada a outra por vários arcos diferentes. Assim, os algoritmos que buscam por entidades impactadas percorrem o grafo através dos relacionamentos de interesse.

A seguir, explanamos como a herança e o polimorfismo foram tratados durante a geração do modelo de entidades e relacionamentos.

3.3.1 Herança e Polimorfismo

A herança e o polimorfismo são duas características das linguagens orientadas a objetos que dificultam a análise estática do código. O polimorfismo permite que uma chamada de método seja executada de várias formas diferentes. Entretanto, esse é o principal causador de falso-positivos na análise de impacto estática, pois, quando adotado, torna-se impossível determinar qual subtipo será usado para executar um método de um supertipo apenas com o *bytecode*. De tudo, essa informação só pode ser obtida em tempo de execução. Nesse caso, todos os subtipos devem ser incluídos no conjunto de impactos por nossa técnica de análise de impacto.

A herança também complica a análise de impacto. As relações hierárquicas precisam ser analisadas sempre que um subtipo ou um supertipo está no conjunto de impactos. Em Java, existem seis tipos possíveis de relações de herança, listados na tabela 3.3.

Tabela 3.3: Tipos de herança presentes na linguagem Java

	Subtipo	**Relação**	**Supertipo**
1	interface	estende	interface
2	classe abstrata	implementa	interface
3	classe	implementa	interface
4	classe abstrata	estende	classe abstrata
5	classe	estende	classe abstrata
6	classe	estende	classe

Para exemplificar os problemas supracitados, a figura 3.5 mostra um código Java com herança e polimorfismo. A relação de herança existe entre o su-

pertipo `Pessoa` e os subtipos `Estudante` e `Professor`. O polimorfismo pode ser observado em `Universidade.addPessoa(Pessoa p)` e `Universidade.findProfessor(String nome)`. No primeiro método, o parâmetro `Pessoa p` pode ser um objeto do tipo `Estudante` ou `Professor`. No segundo caso, o conjunto `Universidade.pessoas` é iterado e, nele, também podem existir objetos de ambos os subtipos.

Nesse cenário, propomos e analisamos duas mudanças diferentes: M1 e M2. M1 é uma alteração do método `getNome()` do supertipo `Pessoa` para receber como parâmetro um `int i`. A notação dessa mudança, seguindo a especificação da tabela 3.1, seria `changeSignatureParameter Pessoa.getNome() to Pessoa.getNome(int i)`. M2 é uma alteração do método `getNome()` do subtipo `Professor` para receber como parâmetro um `Formato f`. A notação dessa mudança, seguindo a especificação da tabela 3.1, seria `changeSignatureParameter Professor.toString() to Professor.toString(Formato f)`.

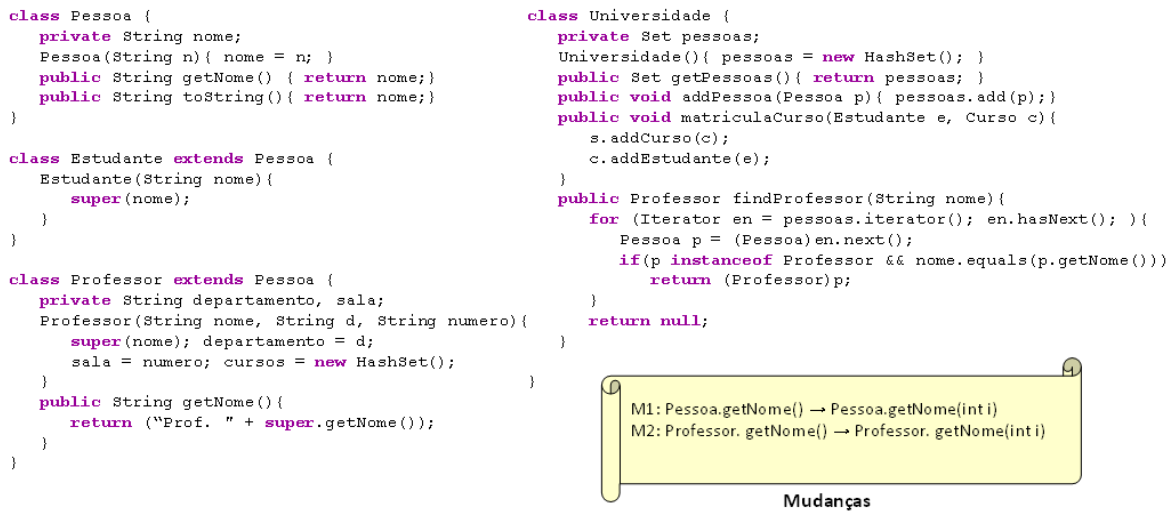


Figura 3.5: Trecho de código com herança e polimorfismo

Primeiro, analisemos M1: `Pessoa.getNome()` será alterado para `Pessoa.getNome(int i)`. `Pessoa` mantém uma relação de herança do tipo 6 (ver tabela 3.3) com `Estudante` e `Professor`. Dessa forma, a princípio, o método `Professor.getNome()` não precisaria ser alterado, pois a classe `Professor` herdaria o novo método de `Pessoa`. No entanto, o método `Universidade.findProfessor(String nome)` faz uma chamada

`p.getNome()`, que é impactada por M1 e deve passar a ser `p.getNome(int i)`. Nesse caso, quando a pessoa for um professor de nome José, o método mostrará "José" ao invés de "Prof. José", mudando a semântica do sistema. Nossa técnica deve identificar os elementos de forma a evitar que mudanças semânticas ocorram sem que o engenheiro de software esteja consciente disso. Caso a mudança semântica tenha sido proposta de forma consciente, o engenheiro irá ignorar os impactos apontados pela técnica. Então, para evitar mudanças semânticas, consideramos que `Professor.getNome()` é uma entidade impactada. O mesmo ocorre para `Estudante.getNome()`.

O mesmo cenário se repete para as relações de herança 1, 2 e 4 da tabela 3.3. Para as relações 3 e 5, há quebra de contrato se `Professor.getNome()` não mudar, causando erro sintático. Portanto, em todos os casos, se o método de um supertipo é modificado, os métodos equivalentes (herdados ou re-implmentados) dos seus subtipos devem ser inclusos no conjunto de impactos.

Agora, analisemos M2: `Professor.getNome()` será alterado para `Professor.getNome(int i)`. `Professor` mantém uma relação de heranças do tipo 6 com `Pessoa`. Então, a princípio, nada mudaria em `Pessoa` e a classe `Professor` passaria a ter os métodos: `getNome()`, herdado de `Pessoa`, e o novo `getNome(int i)`. Esse cenário causa o mesmo problema semântico descrito anteriormente, pois, apesar de a mudança não ter causado erro sintático, ela mudou a semântica dos métodos. O método `Universidade.findProfessor(String nome)`, que faz uma chamada `p.getNome()`, deve ser modificado para chamar `p.getNome(int i)` e a resposta recebida quando uma pessoa for um professor irá mudar. Portanto, nesses casos, consideramos que `Pessoa.getNome()` deve mudar e, por conseqüência, `Estudante.getNome()` também.

Isso também se repete para as relações de herança 1, 2 e 4. Para as relações 3 e 5, há quebra de contrato se `Pessoa.getNome()` e `Estudante.getNome()` não mudarem também, causando erro sintático. Portanto, em todos os casos de herança, se o método de um subtipo é modificado, o método de seu supertipo equivalente (que foi re-implementado no subtipo) deve ser incluso no conjunto de impactos.

Dessa forma, nossa abordagem considera que uma mudança em qualquer nível da hierarquia de herança impacta todos os outros elementos contidos nessa hierarquia. Para permitir

a busca de entidades impactadas acima e abaixo em uma hierarquia de herança, realizamos alguns tratamentos nos subtipos e supertipos presentes no modelo produzido pelo Design-Wizard. Esses tratamentos são abordados a seguir.

Tratamento dos Métodos Herdados

Quando um subtipo não implementa todos os métodos que existem no supertipo, esses métodos que não foram implementados são herdados pelo subtipo. No *bytecode* de Java, o que ocorre é que o método herdado não existe de fato no subtipo, assim como vemos no código fonte. A figura 3.6 ilustra uma dessas situações. A classe `Estudante` é subtipo de `Pessoa` e não implementa os métodos `getNome()` e `toString()`. Conseqüentemente, esses métodos são herdados por `Estudante`.

O problema é que dentro do método `Universidade.findEstudante(String nome)` há uma chamada `e.getNome()`. No momento da geração do modelo, um relacionamento do tipo <`Universidade.findEstudante(String nome)`> *invokeSpecial* <`Estudante.getNome()`> será criado. No entanto, esse relacionamento não existe, visto que a entidade <`Estudante.getNome()`> também não existe. O que existe, na verdade é o relacionamento <`Universidade.findEstudante(String nome)`> *invokeSpecial* <`Pessoa.getNome()`>.

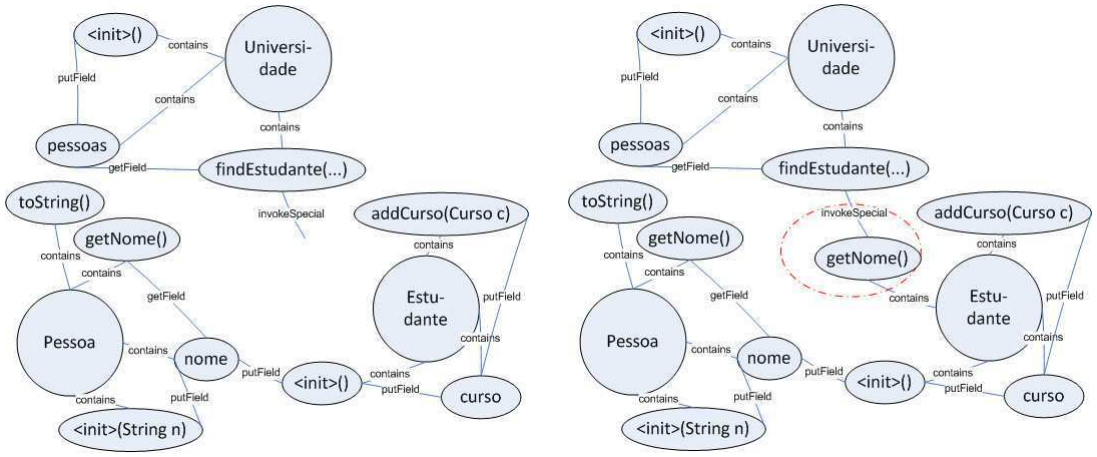
```
class Pessoa {
   private String nome;
   Pessoa(String n){ nome = n; }
   public String getNome() { return nome;}
   public String toString(){ return "";}
}
class Estudante extends Pessoa {
   private Set cursos;
   Estudante(String nome){
      super(nome);
      cursos = new HashSet();
   }
   public void addCurso(Curso c){ cursos.add(c);}
}
```

```
class Universidade {
   private Set pessoas;
   Universidade(){ pessoas = new HashSet(); }
   public Estudante findEstudante(String nome){
      for (Iterator en = pessoas.iterator();
     en.hasNext(); ){
          Pessoa p = (Pessoa)en.next();
          if (p instanceof Estudante ){
             Estudante e = (Estudante)p;
             if (nome.equals(e.getNome())
                 return e;
          }
      }
      return null;
   }
}
```

Figura 3.6: Trecho de código para tratamento de métodos herdados

Para contornar esse problema, primeiramente, adicionamos as entidades herdadas a seus devidos subtipos. A figura 3.7 ilustra como eram os relacionamentos entre as entidades da figura 3.6 e como é após as adições propostas. Em nível de implementação, esse tratamento é feito através do algoritmo presente no código 3.1. Para cada supertipo, o algoritmo busca

todos os subtipos. Então, para cada subtipo, os atributos e os métodos do supertipo, que não estiverem na subclasse, são adicionados.



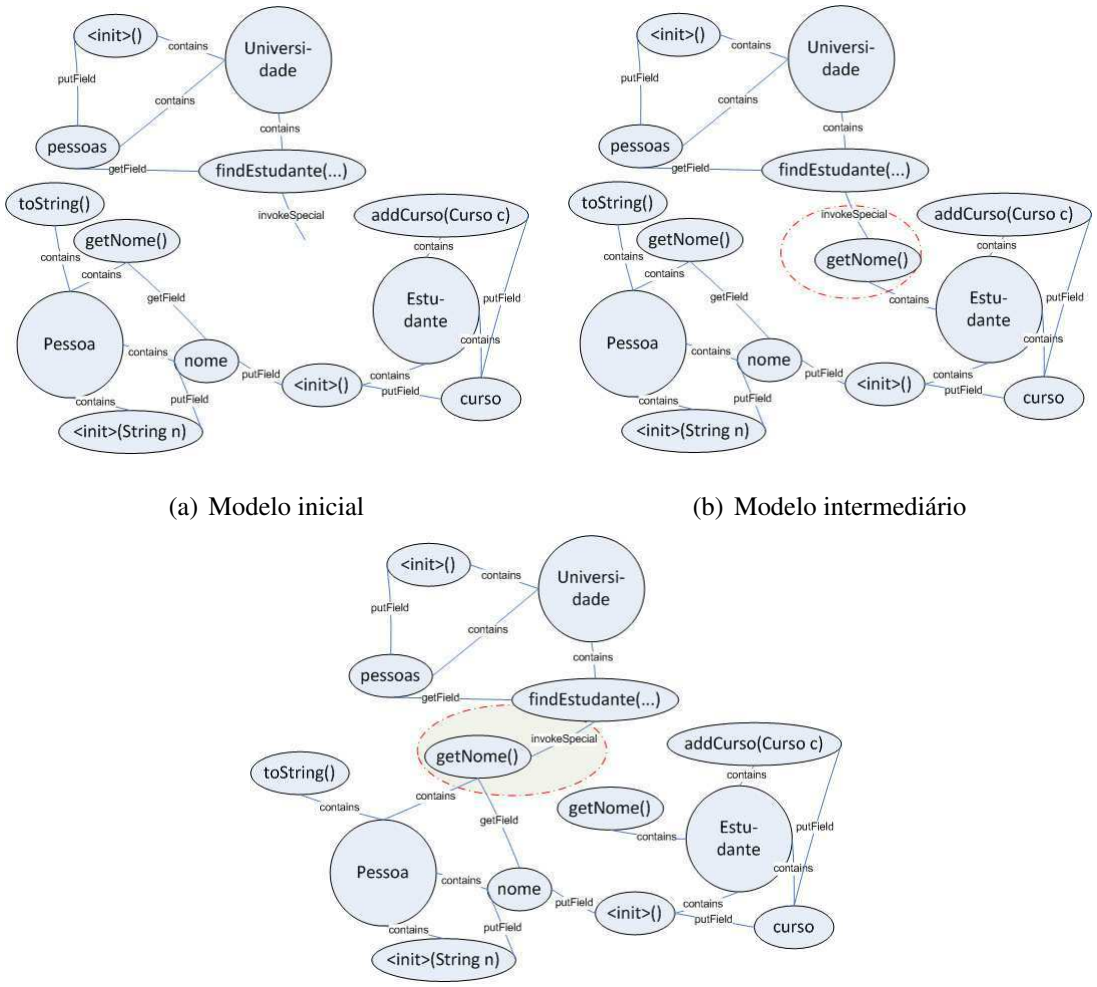
(a) Modelo antes das adições (b) Modelo após adições

Figura 3.7: Grafo de entidades e relacionamentos do código da figura 3.6

```
1  adicionaMétodosHerdados(NóClasse classe)
2     NóClasse subClasses[] <= classe.getSubClasses()
3     PARA CADA subClasse em subClasses[] FAÇA
4        NóAtributo atributos[] <= classe.getTodosAtributos()
5        PARA CADA atributo em atributos[] FAÇA
6           INCLUI atributo em subClasse.atributos[]
7        FIM_PARA
8        NóMetodo métodos[] <= classe.getTodosMétodos()
9        PARA_CADA método em métodos FAÇA
10             INCLUI método em subClasse.métodos[]
11       FIM_PARA
12       adicionaMétodosHerdados(subClasse)
13    FIM_PARA
```

Código 3.1: Tratamento de métodos herdados

A partir da adição dos métodos herdados aos subtipos, é possível encontrarmos em qual supertipo o método herdado está implementado de fato e, assim, corrigir o relacionamento. No exemplo da figura 3.6, a entidade `Estudante.getNome()` é removida do relacionamento <`Universidade.findEstudante(String nome)`> *invokeSpecial* <`Estudante.getNome()`> e a entidade `Pessoa.getNome()` é adicionada em seu lugar. A figura 3.8 ilustra as três versões do grafo de entidades e relacionamentos desse exemplo, sendo que a última é o resultado o tratamento que realizamos.



(a) Modelo inicial

(b) Modelo intermediário

(c) Modelo final

Figura 3.8: Grafo de entidades e relacionamentos do código da figura 3.6 após correção do relacionamento

O problema que acabou de ser descrito também ocorre para atributos. Portanto, os relacionamentos que podem ser envolvidos nesse problema são:

- INVOKE – *invokeVirtual*, *invokeSpecial*, *invokeStatic*, *invokeInterface* – método invoca outro método;
- ACCESS – *getStatic*, *putStatic*, *GetField*, *putField* – método acessa um atributo.

Em nível de implementação, o algoritmo `inverteRelacionamentos()`, apresentado no código 3.2, retira o relacionamento com o subtipo e adiciona o relacionamento com o supertipo. Para isso, ele verifica todos os relacionamentos em que um método é o chamador (INVOKE e ACCESS). Se a classe do método ou atributo chamado não o redefine (linhas 4 a 6), então o supertipo que o define é buscado (linha 7). Em seguida, o relacionamento com o supertipo é adicionado e o equivalente com o subtipo é excluído (linhas 8 a 15).

```
1  inverteRelacionamentos()
2     PARA CADA método do sistema FAÇA
3        PARA CADA relacionamento INVOKE e ACCESS do método FAÇA
4           NóEntidade entidade <= relacionamento.getEntidadeChamada()
5           NóClasse dono <= entidade.getClasse()
6           SE dono não redefine entidade ENTÃO
7              NóClasse superClasse <= dono.getSuperClasse()
8              SE entidade é atributo ENTÃO
9                 ADICIONA(relacionamento.getTipo(),
10                          método, superClasse.getAtributo(entidade))
11             SENÃO
12                ADICIONA(relacionamento.getTipo(),
13                          método, superClasse.getMétodo(entidade))
14             FIM_SE_SENÃO
15             EXCLUI relacionamento
16          FIM_SE
17       FIM_PARA
18    FIM_PARA
```

Código 3.2: Tratamento de chamadas de métodos dentro de uma hierarquia de herança

Após a execução dos dois algoritmos relacionados à herança, os conjuntos de entidades e relacionamentos estarão completos e corretos para serem submetidos aos algoritmos de análise de impacto, que são descritos a seguir.

3.4 Algoritmos

Após receber o conjunto de mudanças proposto e produzir a representação do sistema, o próximo passo é buscar os impactos para cada mudança, individualmente. Para isso, apresentamos um conjunto de algoritmos que busca por impactos de acordo com o tipo da mudança proposta. O pseudocódigo 3.3 representa o algoritmo principal, responsável por identificar o tipo de mudança que se quer realizar em cada entidade do conjunto de mudanças e chamar o algoritmo adequado. Por exemplo, se o tipo de mudança é uma remoção da herança, então o `inicializador()` chamará o algoritmo `removeInheritance(...)`, como pode ser observado nas linhas 12 a 14 do pseudocódigo.

```
1  Lista entidadesImpactadas //global - armazena todas as entidade que já
2                            //foram identificadas como impactadas
3  Lista impactos // estrutura que armazena os impactos
4  Mudança mudanças[] // conjunto de mudanças

5  inicializador()
6    Árvore impacto
7    PARA CADA mudança em mudanças[] FAÇA
8      SE mudança.getTipo() = changeVisibility ENTÃO
9        impacto <= changeVisibility(
10                  mudança.getEntidade(), mudança.getVisibilidade())
11     SENÃO
12       SE mudança.getTipo() = removeInheritance ENTÃO
13         impacto <= removeInheritance(
14                  mudança.getEntidade(), mudança.getSuperClasse())
15       SENÃO
16         SE mudança.getTipo() = addInheritance ENTÃO
17           impacto <= addInheritance(
18                  mudança.getEntidade(), mudança.getSuperClasse())
19         SENÃO
20          impacto <= modification(mudança.getEntidade())
21         FIM_SE_SENÃO
22       FIM_SE_SENÃO
23     FIM_SE_SENÃO
24     INCLUI(impactos, impacto)
25   FIM_PARA
```

Código 3.3: Estruturas usadas nos algoritmos e método inicializador

Existem quatro algoritmos que analisam os impactos de acordo com os tipos de mudanças: `modification(...)`, `changeVisibility(...)`, `addInheritance(...)` e `removeInheritance(...)` (linhas 8 a 20 do código 3.3). Todos eles utilizam as estruturas de dados declaradas nas linhas 1 a 4 para

calcular os impactos. A estrutura `entidadesImpactadas` é uma lista global que armazena todas as entidades que já foram adicionadas em alguma árvore. A estrutura `mudancas[]` armazena as mudanças propostas e `impactos[]` é uma lista que armazena as árvores de impactos, cada uma correspondente a uma mudança.

Os conjuntos de entidades e relacionamentos que representam o sistema podem ser vistos como um grafo no qual os nós são representados pelas entidades e as arestas são os relacionamentos. O princípio de busca dos algoritmos de análise de impacto é percorrer o grafo do sistema buscando por entidades chamadoras. O que difere a busca de cada algoritmo é o tipo de relacionamento considerado, que varia de acordo com o tipo de mudança.

A figura 3.9 ilustra um exemplo simples da identificação dos impactos dado um conjunto de mudanças. Para cada mudança, um algoritmo é executado e cria uma árvore de entidades impactadas, na qual a raiz é a própria mudança e as folhas e os nós intermediários são entidades chamadoras. Uma entidade pode ser impactada diretamente ou indiretamente por uma mudança. Se ela estiver no primeiro nível da árvore, é impactada diretamente pela mudança. Se estiver no segundo nível ou abaixo, é impactada indiretamente, ou seja, a mudança causa impacto direto em uma entidade, que, ao mudar, causa impacto em outra entidade. O resultado final é uma coleção de árvores, cada uma referente a uma mudança proposta.

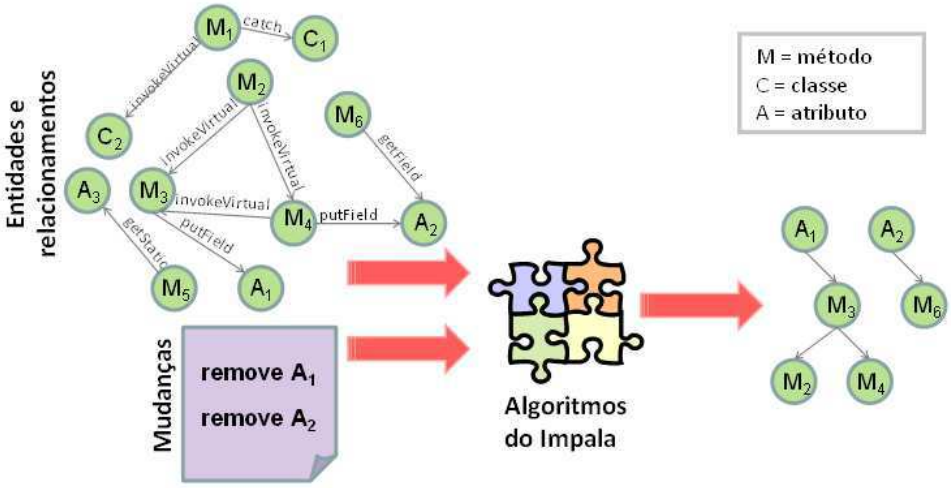


Figura 3.9: Entradas e saída dos algoritmos de análise de impacto

No exemplo da figura 3.9, a remoção A_1 impacta diretamente M_3 e indiretamente M_2 e M_4. A remoção de A_2 impacta diretamente M_6 e M_4, mas M_4 já está na lista de entidades impactadas e não é adicionada novamente. Dessa forma, para o conjunto de mudanças

$\{A_1, A_2\}$, o algoritmo `modification(...)`, chamado em cada remoção, identifica o conjunto de impactos $\{A_1, A_2, M_3, M_4, M_6\}$.

A seguir, descrevemos com detalhes os quatro algoritmos de análise de impacto.

3.4.1 Algoritmo Modification

O algoritmo `modification(...)` é utilizado para mudanças do tipo: *remove*, *changeAttribute*, *changeAttributeType*, *changeAttributeName*, *changeSemantics*, *changeSignature*, *changeSignatureParameter*, *changeSignatureException* e *changeReturnType* (ver tabela 3.1 para obter lista completa). Mudanças do tipo *add* não causam nenhum impacto no sistema, visto que consideramos todas as mudanças como atômicas. Assim, uma adição de um método, uma classe ou um atributo não acrescenta nenhuma dependência, nenhum relacionamento. O código 3.4 mostra o algoritmo `modification(...)`. Ele recebe como parâmetro a entidade modificada e busca todos os métodos que usam a entidade da seguinte forma, de acordo com seu tipo:

- método – busca, no conjunto de relacionamentos do método, os relacionamentos do tipo `is_invokedBy` e retorna os métodos chamadores – linhas 14 a 16;
- atributo – busca, no conjunto de relacionamentos do atributo, os relacionamentos do tipo `is_accessedBy` e retorna os métodos chamadores – linhas 9 a 12;
- classe – busca, no conjunto de relacionamentos da classe, os relacionamentos do tipo `is_invokedBy` e retorna os métodos chamadores – linhas 14 a 16.

Um método M_1 é considerado chamador de um método M_2 quando, dentro do corpo de M_1 existe uma chamada para M_2. Um método M_1 é considerado chamador de um atributo A_1 quando M_1 atribui valor a A_1 ou lê. Por fim, M_1 é considerado chamador de uma classe C_1 quando ele carrega (*load*) a classe. Por exemplo, no trecho de código abaixo, o método `suite()` carrega a classe `RelationTest` através de reflexão, na linha 2.

```
public static Test suite() {
  TestSuite suite = new TestSuite("Test for DesignWizard");
  suite.addTest(new TestSuite(RelationTest.class));
}
```

```
árvore modification(entidade)
  Árvore nó //nó da árvore que armazena o resultado
  NóMétodo método
  SE entidadesImpactadas não contém entidade ENTÃO
    nó.entidade <= entidade
    INCLUI(entidadesImpactadas, entidade) //lista global de impactos
    Relacionamento relacionamentos[] <= entidade.getRelacionamentos()
    PARA CADA relacionamento em relacionamentos[] FAÇA
      SE entidade.tipo = "atributo"
                  E relacionamento = "is\_accessedBy" ENTÃO
       método <= relacionamento.getChamador()
        INCLUI(nó.folhas[], modification(método))
      SENÃO
        SE relacionamento = "is\_invokedBy" ENTÃO
          método <= relacionamento.getChamador()
          INCLUI(nó.folhas[], modification(método))
        FIM_SE
      FIM_SE_SENÃO
    FIM_PARA
  FIM_SE
  retorna nó
```

Código 3.4: Algoritmo modification(...)

3.4.2 Algoritmo ChangeVisibility

O algoritmo `changeVisibility(...)` trata mudanças do tipo *changeVisibility* para os três tipos de entidades. A mudança pode envolver aumento ou redução de visibilidade de uma entidade. Em Java, as palavras reservadas *public*, *protected* e *private* são modificadores de acesso utilizados para mudar a visibilidade de uma entidade. As permissões para cada tipo de visibilidade, seguindo a especificação da linguagem Java, são informadas a seguir.

- *Private* – atributo e método privado podem ser acessados apenas por entidades que estiverem na mesma unidade de compilação, ou na mesma classe. Uma classe não pode ser declarada privada.

- *Package* – uma entidade é *package* quando tem visibilidade padrão, ou seja, é declarada sem modificador de acesso. Entidades com visibilidade padrão podem ser acessadas por outras entidades que estiverem no mesmo pacote.

- *Protected* – atributo e método protegidos podem ser acessados por outras entidades que estiverem no mesmo pacote ou por subclasses por meio de herança, mesmo que elas estejam em outro pacote. Uma classe não pode ser declarada como protegida.

- *Public* – entidade pública pode ser acessada por qualquer outra entidade do sistema.

Existe uma ordem de restrição dos modificadores de acesso: *public* > *protected* > *Package* > *Private*. Quanto menor a visibilidade, mas restrito é o acesso. Quando a mudança é um aumento de visibilidade, não há impacto nas entidades chamadoras. No entanto, quando é uma redução de visibilidade, vários impactos podem ser causados, porque as entidades chamadoras podem perder o acesso à entidade modificada. Se uma entidade muda de *public* para *package*, por exemplo, nenhuma outra entidade que está fora de seu pacote poderá mais usá-la. Portanto, todas essas entidades serão impactadas.

O algoritmo `changeVisibility(...)`, exibido no código 3.5, busca por impactos na hierarquia de herança de acordo com a visibilidade proposta. Se a visibilidade for do tipo *Public*, então não há impactos, como se pode observar na linha 8 do código. Para os outros tipos de visibilidade é preciso verificar as relações de dependência de cada membro da hierarquia de herança que for superclasse ou supertipo da entidade que será mudada. Assim, nas linhas 7 e 10, a estrutura `classesHerança[]` armazena a entidade em questão e todas as suas superclasses. Essa estrutura é iterada na linha 11. Para cada classe, há uma verificação dos métodos que a chamam de acordo com as características específicas para cada visibilidade restante.

Nas linhas 19 a 21, verificamos se o método chamador está no mesmo pacote da classe. Caso não esteja, então ele é uma entidade impactada por uma redução de visibilidade de pública para padrão. A verificação do tipo *Protected*, que é feita nas linhas 23 a 30, obedece à mesma regra do tipo *Package* e outra: se a classe à qual o método chamador pertence não for uma subclasse da classe iterada, então ela é impactada (linhas 28 a 30). Por fim, o algoritmo busca por entidades impactadas por mudança de visibilidade para *Private* nas linhas 32 a 37. A regra é: se a classe à qual o método chamador pertence não for a própria classe iterada, então ela é impactada e o algoritmo `modification()` é chamado para buscar seus chamadores.

3.4.3 Algoritmo RemoveInheritance

O algoritmo `removeInheritance(...)`, exibido no código 3.6, trata mudanças do tipo *removeInheritance*, que descreve a remoção da herança de uma classe. A remoção da he-

```
árvore changeVisibility(entidade, visibilidade)
  Árvore nó //nó da árvore que armazena o resultado
  NóMétodo método
  SE entidadesImpactadas não contém entidade ENTÃO
    nó.entidade <= entidade
    INCLUI(entidadesImpactadas, entidade) //lista global de impactos
    Entidade classesHeranca[] <= entidade
    SE visibilidade <> PUBLIC ENTÃO
      //inclui toda a hierarquia acima da entidade em classesHerança[]
      INCLUI (classesHerança[], entity.getTodasSuperClasses())
      PARA CADA classe em classesHerança[] FAÇA
        SE entidadesImpactadas não contém entidade ENTÃO
          INCLUI(nó.folhas[], classe)
          INCLUI(entidadesImpactadas, classe) //lista global de impactos
        FIM_SE
        NóMétodo métodos[] <= classe.getChamadores()
        PARA CADA método em métodos[] FAÇA
          NóClasse classeMétodo
          SE visibilidade = PACKAGE FAÇA
            SE classe.getNomePacote() <> método.getNomePacote() ENTÃO
              INCLUI(nó.folhas[], modification(método))
            SENÃO
              SE visibilidade = PROTECTED FAÇA
                SE classe.getNomePacote() <> método.getNomePacote() ENTÃO
                  INCLUI(nó.folhas[], modification(método))
                FIM_SE
                classeMétodo <= método.getClasseQueDeclara()
                SE classe.getSubclasses() não contém classeMétodo ENTÃO
                  INCLUI(nó.folhas[], modification(método))
                FIM_SE
              SENÃO
                SE visibilidade = PRIVATE FAÇA
                  classeMétodo <= método.getClasseQueDeclara()
                  SE classeMétodo <> classe ENTÃO
                    INCLUI(nó.folhas[], modification(método))
                  FIM_SE
                FIM_SE
              FIM_SE_SENÃO
            FIM_SE_SENÂO
          FIM_SE
        FIM_PARA
      FIM_PARA
    FIM_SE
  FIM_SE
  retorna nó
```

Código 3.5: Algoritmo changeVisibility(...)

rança envolve um entre os dois tipos de herança existentes em Java: subtipo implementa (`implements`) um supertipo; subclasse estende (`extends`) superclasse.

```
1  árvore removeInheritance(subClasse, superClasse)
2    Árvore nó
3    NóMétodo método
4    SE entidadesImpactadas não contém subClasse ENTÃO
5      nó.entidade <= subClasse
6      INCLUI(entidadesImpactadas, entidade)
7      chamadores[] = superClasse.getChamadores()
8      PARA CADA chamador em chamadores[] FAÇA
9        SE chamador.getClassNome() = subClasse.getNome() ENTÃO
10       método <= relacionamento.getChamador()
11         INCLUI(nó.folhas[], modification(método))
12       FIM_SE
13     FIM_PARA
14     NóMétodo construtores[] <= subclasse.getConstrutores()
15     PARA CADA construtor em construtores[] FAÇA
16         INCLUI(nó.folhas[], modification(construtor))
17     FIM_PARA
18   FIM_SE
19   retorna nó
```

Código 3.6: Algoritmo removeInheritance(...)

A remoção de herança a uma subclasse impacta, diretamente, todos os métodos da superclasse que foram herdados pela subclasse. Dessa forma, se existe alguma chamada do tipo `super`, em que um método da subclasse chama, explicitamente, uma entidade da superclasse, esse método chamador deve entrar no conjunto de impactos (linhas 7 a 13 do código 3.6. O mesmo acontece se, em um método da subclasse, houver uma chamada a um método da superclasse que não foi redefinido na subclasse.

Um método de outra classe pode instanciar um subtipo a partir da declaração de um supertipo, como é ilustrado a seguir:

```
public void instanciaFoo() {
  Foo foo = new FooColorido();
}
```

A partir do momento em que um subtipo deixa de herdar um supertipo, ele quebra o contrato que tinha com esse subtipo. Dessa forma, um método que declara um atributo do tipo Foo (supertipo), mas instancia a classe FooColorido (subtipo), será impactado pela quebra de contrato. Para resolver esse problema, adotamos uma abordagem conservativa. O algoritmo `removeInheritance(...)` inclui todos os construtores da subclasse como

entidades impactadas (linhas 14 a 17) e chamamos o algoritmo `modification()` para buscar por seus métodos chamadores.

3.4.4 Algoritmo AddInheritance

O algoritmo `addInheritance(...)`, apresentado no código 3.7, busca por impactos gerados pela adição de herança a uma classe. A adição da herança envolve um entre os dois tipos de herança existentes em Java: subtipo implementa (`implements`) um supertipo; subclasse estende (`extends`) superclasse.

Para o segundo caso, se a subclasse já estender de outra superclasse, esta deverá ser removida. Essa verificação é feita nas linhas 7 a 11. 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 (linhas 12 a 18 do código).

```
1  árvore addInheritance(subClasse, superClasse)
2    Árvore nó
3    NóMétodo método
4    SE entidadesImpactadas não contém subClasse ENTÃO
5      nó.entidade <= subClasse
6      INCLUI(entidadesImpactadas, entidade)
7      SE subClasse estende outraSuperClasse ENTÃO
8        outraSuperClasse <= subClasse.getSuperClasse()
9        INCLUI(nó.folhas[],
10                removeInheritance(subClasse, outraSuperClasse))
11     FIM_SE
12     SE superClasse é abstrata ou interface ENTÃO
13       métodos[] <= superClasse.getMétodosAbstratos()
14       PARA CARA método em métodos[] FAÇA
15         INCLUI(subClasse.métodos, método)
16         INCLUI(nó.folhas[], subclasse.método)
17       FIM_PARA
18     FIM_SE
19   FIM_SE
20   retorna nó
```

Código 3.7: Algoritmo addInheritance(...)

3.5 Considerações Finais

Neste capítulo apresentamos nossa técnica de análise de impacto de mudanças para sistemas orientados a objetos. Essa recebe como entrada o software a ser analisado e o conjunto de

mudanças propostas e fornece como saída um conjunto de elementos possivelmente impactados. Para isso, ela extrai informações do design de um software, desenvolvido na linguagem Java, para produzir uma representação em forma de conjuntos de entidades e relacionamentos. A partir dessa representação, a técnica busca por entidades que podem ser impactadas por cada mudança, individualmente.

Essa técnica é considerada de granularidade grossa (*coarse-grained*) porque analisa apenas o fluxo de controle do software. Em outras palavras, ela realiza a busca por entidades impactadas em nível de método, analisando os métodos que chamam outros métodos ou que usam outras classes ou atributos. Não entram na análise questões como o estado do objeto ou a semântica de um método.

Algumas características da orientação a objetos não são tratadas por nossa técnica, como a herança múltipla. A passagem de parâmetro por valor, que pode ser usada na linguagem C++, por exemplo, também não é considerada por nossa técnica. Para ambos os casos, o motivo para não abordarmos é que a linguagem Java não tem suporte a essas características. Como analisamos softwares escritos em Java, nos restringimos a analisar as características da orientação a objetos presentes na linguagem Java. Porém, nossa técnica pode ser facilmente estendida para considerar essas características. Quando uma entidade fosse passada como parâmetro por valor, por exemplo, a técnica teria que identificar todas as entidades que a usam e incluir no conjunto de possíveis impactos.

É possível notar que a técnica de análise de impacto estática gera falso-positivos, principalmente quando casos de herança estão envolvidos. Quando uma subclasse muda ou é impactada, toda a hierarquia de herança da qual ela faz parte é analisada. Como optamos por uma abordagem conservativa, muitas das entidades envolvidas na herança são consideradas impactadas por segurança. Para ponderar os falso-positivos, no próximo capítulo descreveremos uma análise probabilística que atribui probabilidades de impacto a cada uma das entidades possivelmente impactadas de acordo com o histórico de mudanças do sistema.

Capítulo 4

Atribuição de Probabilidade Baseada nas Características Evolutivas do Software

Neste capítulo propomos uma abordagem para atribuição de probabilidades de impacto aos resultados obtidos pela técnica de análise de impacto estática, que chamamos de análise histórica. A análise histórica é baseada em informação histórica de mudanças do sistema obtidas do repositório (CVS), do qual apenas as mudanças estruturais são consideradas. A partir da extração das mudanças estruturais, construímos uma matriz que relaciona as transações de *commits* que ocorreram até o momento da análise. A matriz, juntamente com o conjunto de mudanças e o conjunto de impactos são entrada para o algoritmo de cálculo das probabilidades.

4.1 Visão Geral

A técnica de análise de impacto de mudanças, proposta e desenvolvida neste trabalho, identifica, através da análise estática do código do software, um conjunto de entidades que pode ser impactado por um conjunto de mudanças. Sabe-se que a análise estática, contudo, identifica um grande número de possíveis impactos que não ocorre quando as mudanças são implementadas. Abordagens mais conservadoras podem incluir até 90% do software analisado [Apiwattanapong, Orso e Harrold 2005], a depender da abrangência do conjunto de mudanças.

Para refinar o resultado obtido pela técnica de análise de impacto, propomos calcular a

probabilidade de cada entidade do conjunto de impacto ser realmente impactada, baseada no seu histórico de mudanças. Para isso, acessamos o sistema de controle de versão (CVS) do projeto e coletamos informações sobre as mudanças estruturais ocorridas em cada validação (*commit*). Uma mudança estrutural é qualquer mudança sintática no código, como mudança na assinatura de um método, adição ou remoção de classe, método ou atributo, ou remoção de uma relação de herança. Não são mudanças estruturais alterações em comentários (Javadocs), mudanças nos termos de licença ou de formatação do código. As informações coletadas são utilizadas para calcular a probabilidade de que cada entidade, do conjunto de impactos, seja afetada.

Uma entidade do conjunto de impactos pode ser afetada por apenas uma entidade do conjunto de mudanças. Outra possibilidade é que a entidade do conjunto de impactos seja afetada por duas ou mais entidades do conjunto de mudanças. Portanto, para cada entidade do conjunto de impactos, calculamos a probabilidade de ela ser impactada por qualquer uma das entidades do conjunto de mudanças. Nossa abordagem é semelhante às adotadas por Ying et al e Zimmermann et al [Ying 2003; Zimmermann et al. 2004]. No entanto, ao invés de aplicar algoritmos já conhecidos de mineração de dados, como o Apriori [Agrawal e Srikant 1998] e o FP-Tree [Han, Pei e Yin 2000], optamos por propor um algoritmo que aplica diretamente o teorema de Bayes. Como prova de conceito, implementamos um protótipo do cálculo de probabilidades, que chamamos de ImpalaMiner.

Optamos por propor um novo algoritmo porque adotar o Apriori ou o FP-Tree para calcular especificamente a probabilidade de uma entidade ser impactada por qualquer entidade de um conjunto de mudanças seria computacionalmente custoso e desnecessariamente complexo. Ambos os algoritmos de mineração supracitados computam conjuntos formados por conjunções de entidades que mudaram juntas no passado, mas o que queremos é a conjunção entre a entidade impactada e o conjunto formado pela disjunção de todas as entidades do conjunto de mudanças.

O grande problema desses dois algoritmos é que o número de regras a ser gerado é igual a $\sum_{k=2}^{n} \frac{n!}{(n-k)!}$, onde n é o número total de entidades do software e k é o tamanho das possíveis regras de associação. Como nossa abordagem permite que a análise seja em feita em nível de entidade (métodos e atributos), o número de possíveis regras tende a ser muito grande e seu cômputo custoso.

Por essa razão, optamos por construir um algoritmo, baseado no teorema de Bayes, que calcula diretamente a freqüência que um impacto foi modificado com pelo menos uma entidade do conjunto de mudanças. A figura 4.1 ilustra uma visão geral da solução proposta. O CVS do software analisado é acessado pelo Impala *Plug-in*, que compara versão por versão de cada classe do sistema para extrair as mudanças estruturais (passo 1). Destas, extraímos as transações de *commits* de acordo com o conceito de transações atômicas: entidades que foram validadas pelo mesmo autor, com o mesmo comentário, dentro da mesma janela de tempo (passo 2). O algoritmo responsável por atribuir probabilidades tem como entrada as transações, o conjunto de mudanças e o conjunto de impactos (passo 3). O resultado fornecido é o conjunto de impactos com as entidades ordenadas de acordo com as probabilidades encontradas pelo ImpalaMiner.

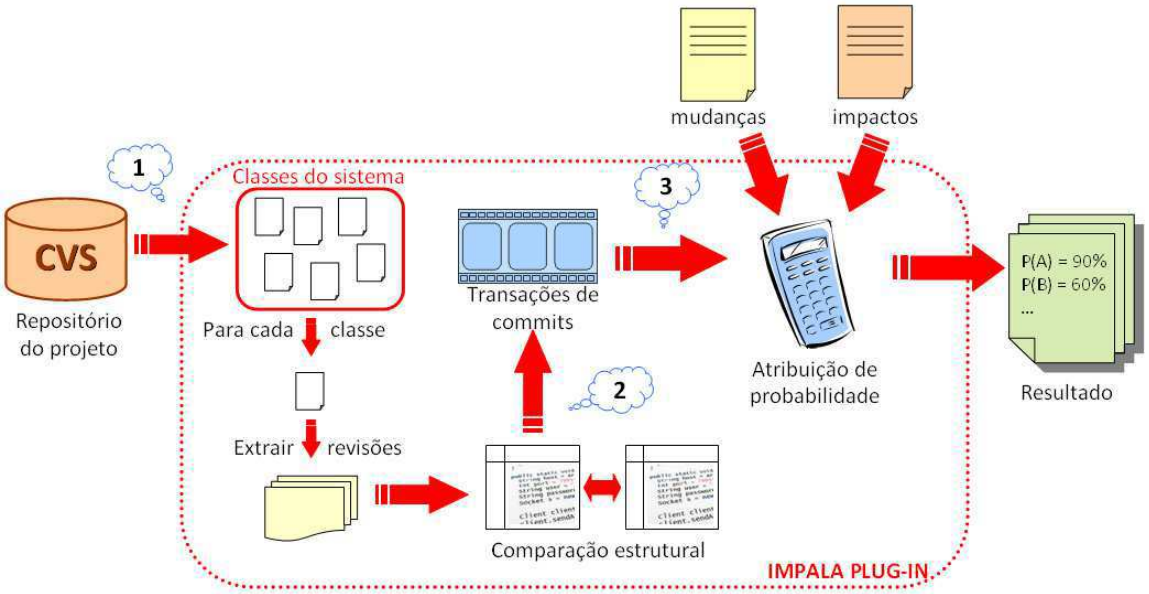


Figura 4.1: Visão geral da solução que calcula as probabilidades de impacto para cada entidade resultante da análise de impacto

A seguir, explanamos como a extração das informações históricas do software é realizada. Posteriormente, abordamos o raciocínio utilizado para elaborar o algoritmo de cálculo das probabilidades. Por fim, mostramos um exemplo da aplicação da técnica de análise de impacto probabilística completa, que engloba as abordagens propostas no capítulo 3 e neste.

4.2 Extração e Comparação Estrutural do Histórico de Mudanças

A idéia geral do processo de extração e comparação estrutural do histórico de mudanças é encontrar todos os registros de quais entidades do software mudaram juntas no passado. Para isso, comparamos, para cada classe, as estruturas de todas as revisões, duas a duas, da mais antiga para a mais recente. A comparação estrutural é realizada após converter cada revisão para árvores sintáticas abstratas (ASTs – *Abstract Syntax Trees*). As informações coletadas sobre cada comparação são armazenadas no banco de dados para, posteriormente, serem consultadas para formar as transações atômicas que serão usadas pelo algoritmo probabilístico. A figura 4.2 ilustra o processo de extração e comparação das mudanças estruturais da classe `MyIResourceVisitor`.

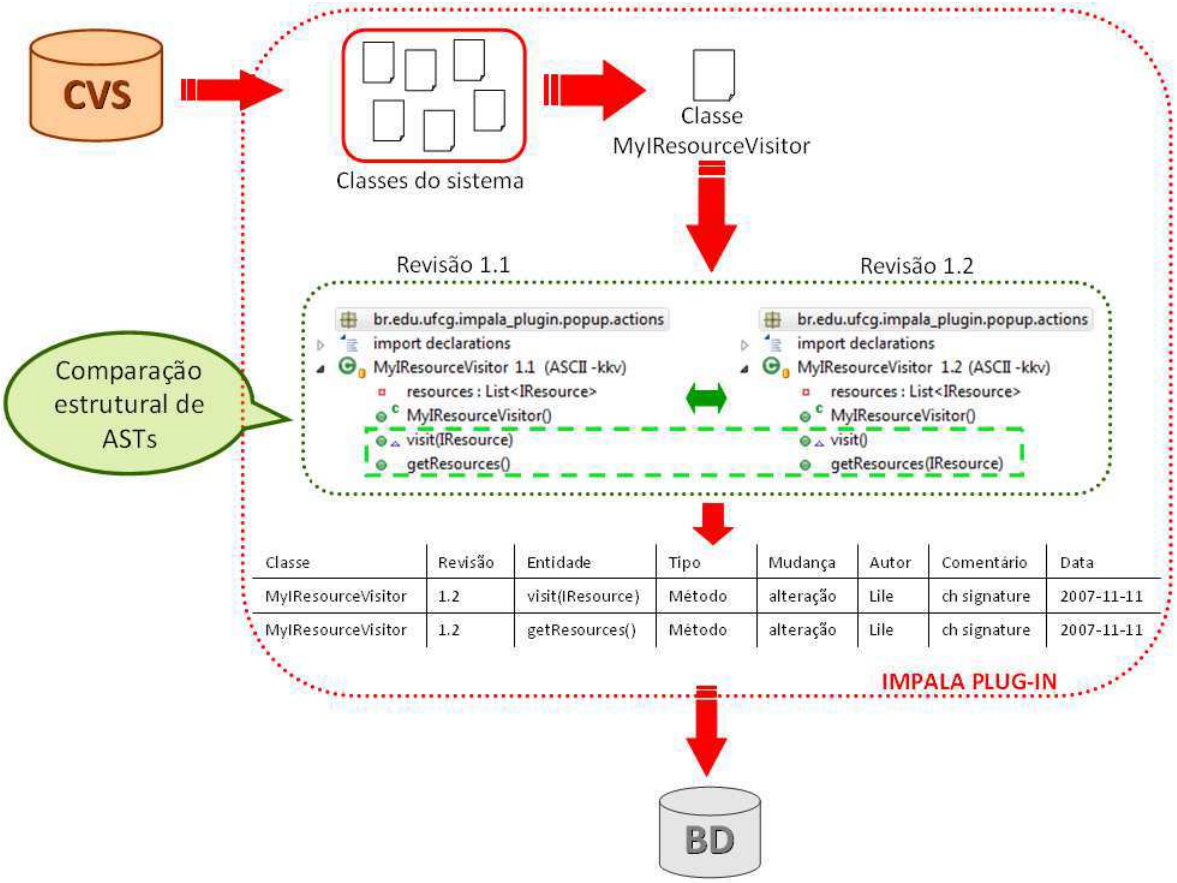


Figura 4.2: Processo de extração e comparação das revisões das classes de um sistema

Ao comparar as ASTs das revisões 1.1 e 1.2, o processo de extração armazena as

seguintes informações no banco de dados: nome da classe, a revisão mais recente, a entidade modificada, tipo de alteração, o tipo da entidade (classe, método ou atributo), autor, comentário e data. Uma tupla contendo essas informações é gerada para cada entidade modificada. Então, se duas entidades foram modificadas de uma revisão para outra, duas tuplas são criadas, como mostra a tabela da figura 4.2.

Uma AST é uma árvore sintática hierárquica, que pode ter como nó raiz um pacote ou uma classe do software. Métodos e atributos são nós intermediários e informações em nível de sentença, como laços, sentenças condicionais e variáveis locais, são nós folha. Na nossa solução, a raiz é uma classe e os nós folha são métodos e atributos. Informações em nível de sentença são ignoradas, pois nossa solução realiza a comparação entre duas revisões em nível de design. Portanto, localizamos as mudanças do tipo adição, remoção ou alteração de classe, método ou atributo (mudanças estruturais) e armazenamos no banco de dados.

Um ponto negativo, que deve ser destacado, é que nossa solução atual não é capaz de identificar alteração de nome. Por exemplo, se o método `visit()`, da figura 4.2 fosse alterado para `visitar()`, a solução consideraria que `visit()` foi removido e `visitar()` foi adicionado. Dessa forma, quando houver uma mudança no nome de um atributo, uma classe, ou um método, as informações de acoplamentos de mudanças com outras entidades que existirem até então serão completamente perdidas. A entidade com o nome alterado será vista como uma nova entidade de forma equivocada. Soluções para esse problema são discutidas na seção 7.2.

As informações das mudanças estruturais salvas no banco de dados estão praticamente prontas para serem utilizadas pelo algoritmo de cálculo das probabilidades. O que falta é encontrar as transações de *commits*, que seguem o conceito apresentado na seção 2.5.1: arquivos que foram validados pelo mesmo autor, com o mesmo comentário e na mesma janela de tempo. Adotamos janela deslizante com 200 segundos de intervalo máximo. Nossa solução tem um diferencial em relação às apresentadas na seção 2.5.1, pois é possível extrair as transações por classe ou por mudança estrutural. Isso permite que o cálculo da probabilidade utilize informações com granularidade mais grossa ou mais fina. A acurácia das duas abordagens é discutida no capítulo de avaliação.

Todas as transações do software são representadas por uma matriz de histórico de mudanças $H_{[|C|\times|E|]}$, na qual:

$C = \{c_1, c_2, ..., c_i | i =$ total de *commits*$\}$

$E = \{e_1, e_2, ..., e_j | j =$ total de entidades$\}$

Para cada *commit* c_i, se uma entidade e_j foi validada, seu valor na matriz $H(i, j)$ será 1. Caso contrário, o valor é igual a 0. Por exemplo, para as mudanças listadas na tabela 4.1, as duas matrizes, em nível de classe e em nível de entidade (mudança estrutural), são apresentadas a seguir.

Tabela 4.1: Mudanças estruturais

Classe	**Revisão**	**Entidade**	**Tipo**	**Mudança**	**Autor**	**Comentário**	**Data**
ResourceVisitor	1.2	visit()	método	remoção	Lile	refactoring	2007-11-11
ResourceVisitor	1.2	visitar()	método	adição	Lile	refactoring	2007-11-11
GenericVisitor	1.4	visit()	método	alteração	Lile	refactoring	2007-11-11
MyComparator	1.3	compare()	método	alteração	Jemerson	-	2007-11-12
MyComparator	1.3	find()	método	alteração	Jemerson	-	2007-11-12
MyContainer	1.2	comment	atributo	remoção	João	ch. type	2007-11-13

Para o conjunto de entidades $E = \{ResourceVisitor.visit(), ResourceVisitor.visitar(), GenericVisitor.visit(), MyComparator.compare(), MyComparator.find(), MyContainer.comment\}$, a matriz é a seguinte:

$$H_{3\times6} = \begin{array}{c|cccccc|c} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & \\ \hline & 1 & 1 & 1 & 0 & 0 & 0 & c_1 \\ & 0 & 0 & 0 & 1 & 1 & 0 & c_2 \\ & 0 & 0 & 0 & 0 & 0 & 1 & c_3 \end{array}$$

Para o conjunto de classes $E = \{ResourceVisitor, GenericVisitor, MyComparator, MyContainer\}$, a matriz é um pouco menor:

$$H_{3\times4} = \begin{array}{c|cccc|c} & e_1 & e_2 & e_3 & e_4 & \\ \hline & 1 & 1 & 0 & 0 & c_1 \\ & 0 & 0 & 1 & 0 & c_2 \\ & 0 & 0 & 0 & 1 & c_3 \end{array}$$

A matriz H é entrada para o algoritmo de cálculo da probabilidade de cada entidade no conjunto de impactos através do teorema de Bayes, que é apresentado a seguir.

4.3 Cálculo da Probabilidade Através do Teorema de Bayes

Retomando a hipótese deste trabalho, queremos aumentar a precisão da análise de impacto estática através do uso das informações históricas de mudanças. Para que isso seja possível, aplicamos a técnica de análise de impacto apresentada no capítulo 3 e extraímos as mudanças estruturais que o software sofreu no passado, como descrito na seção 4.2. Nesta seção, descrevemos como combinar essas duas informações para calcular a probabilidade de cada entidade do conjunto de impactos ser realmente impactada.

A técnica de análise de impacto recebe como entrada o conjunto de mudanças e o software a ser analisado, e fornece como saída o conjunto de impactos. A extração das mudanças estruturais fornece uma matriz H que relaciona *commits* com as entidades do sistema, estas podendo ser em nível de classe ou de método. Queremos calcular, para cada entidade do conjunto de impacto a probabilidade de ela ser impactada por pelo menos uma entidade do conjunto de mudanças. Dessa forma, o cálculo da probabilidade envolve o conjunto de mudanças, o conjunto de impactos e a matriz H. A definição desses elementos é apresentada a seguir.

$E = \{e_1, e_2, ..., e_i | i =$ total de entidades do sistema$\}$

$M = \{e_1, e_2, ..., c_j | j =$ total de entidades do conjunto de mudanças$\}$

$I = \{e_1, e_2, ..., e_k | k =$ total de entidades impactadas$\}$

$M \subseteq I \subseteq E$

$C = \{c_1, c_2, ..., c_l | l =$ total de *commits*$\}$

$c_l \subseteq E$ para todo $c_l \in C$

Todas as transações do software são representadas por uma matriz do histórico de mudanças $H_{[|C| \times |E|]}$, na qual $a_{i,j} = 1$ se $e_j \in c_i$ e $a_{i,j} = 0$ se $e_j \notin c_i$. Por exemplo, dados os conjuntos $E = \{A, B, C\}$ e $C = \{\{A, B\}, \{A, C\}, \{B\}, \{A, B, C\}, \{A, B\}\}$, a matriz H é igual a:

$$H_{5\times3} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Devemos atribuir probabilidades nas entidades do conjunto I. Como as entidades do conjunto M mudarão com 100% de certeza, então, se $e \in M$, então $P(e) = 1$. Dessa forma, se $e \in I - M$, então $P(e)$ é desconhecida e deve ser calculada baseada no histórico de mudanças. Para isso, utilizamos o teorema de Bayes.

Para explanar como o teorema de Bayes foi aplicado para calcular a probabilidade de uma entidade ser impactada, utilizaremos um exemplo. Posteriormente, formalizaremos a aplicação do teorema de Bayes e apresentaremos o pseudocódigo do algoritmo ImpalaMiner.

Suponha o seguinte cenário, no qual E, M, I e H foram definidos anteriormente neste capítulo:

$E = \{A, B, C, D, E, F\}$

$M = \{C, E, F\}$

$I = \{A\}$

$$H_{10\times6} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

A questão a ser respondida é: **qual a probabilidade de a entidade A ser impactada, dado que as entidades C, E e F foram modificadas?** A tabela 4.2 resume as informações da

matriz H. Houve um total de 10 transações, sendo estas de 7 tipos diferentes. As transações dos tipos 2, 3 e 7 ocorreram duas vezes cada, enquanto as outras ocorreram apenas 1 vez cada.

Tabela 4.2: Resumo das informações da matriz H

Entidades							
Transações	A	B	C	D	E	F	**Total**
Transação 1	1	1	0	1	0	0	1
Transação 2	0	0	0	1	0	0	2
Transação 3	1	1	1	1	0	0	2
Transação 4	0	0	0	0	1	1	1
Transação 5	0	1	1	1	1	1	1
Transação 6	1	0	0	0	1	0	1
Transação 7	0	1	0	0	0	1	2

Para que o teorema de Bayes possa ser aplicado, dividimos as transações em duas partições:

$com(A)$ = quantidade de transações que contêm a entidade A = 1 x T1 + 2 x T3 + 1 x T6 = 4

$sem(A)$ = quantidade de transações que não contêm A = 2 x T2 + 1 x T4 + 1 x T5 + 2 x T7 = 6.

Além disso, precisamos calcular quantas vezes C ou E ou F aparecem em cada uma das partições, que chamaremos de com(M), sendo M o conjunto de mudanças. A entidade A pode ser impactada por qualquer uma das mudanças ou qualquer combinação entre elas (apenas C, apenas E, apenas F, C e E, C e F, E e F, C e E e F). Portanto, com(M) será incrementado sempre que uma transação conter pelo menos um dos elementos de M.

$com(A)$ = 4, $com(M_c)$ = 1 x T6 + 2 x T3 = 3;

$sem(A)$ = 6, $com(M_s)$ = 1 x T4 + 1 x T5 + 2 x T7 = 4.

Assim, a aplicação do teorema de Bayes para o cálculo da probabilidade de A é realizada da seguinte forma:

$$
\begin{aligned}
P(com(A)|com(M)) &= \frac{P(com(M_c)|com(A))P(com(A))}{P(com(M_c)|com(A))P(com(A))+P(com(M_s)|sem(A))P(sem(A))} \\
&\frac{\frac{3}{4} * \frac{4}{10}}{(\frac{3}{4} * \frac{4}{10}) + (\frac{4}{6} * \frac{6}{10})} \\
&\frac{\frac{3}{10}}{(\frac{3}{10}) + (\frac{4}{10})} \\
&\frac{3}{7} = 43\%
\end{aligned}
$$

Esse cálculo informa que A tem 20% probabilidade de ser impactada pelo conjunto de mudanças M, que contém as entidades C, E e F.

Formalização da aplicação do teorema de Bayes

Dados os conjuntos M, I e C, o cálculo da probabilidade será realizado para cada $e_i \in I - M$. Assim, o espaço amostral S de e_i será:

$S_{e_i} = \{com(e_i), sem(e_i)\}$

$com(e_i) = \{\sum_{j=0}^{k} c_j | c_j \in C \bigwedge c_j \supset e_i\}, k = |C|$

$sem(e_i) = \{\sum_{l=0}^{k} c_l | c_l \in C \bigwedge c_l \not\supset e_i\}, k = |C|$

O subconjunto $com(M)$ de cada partição de $S = \{com(e_i), sem(e_i)\}$ é:

$com(M_c) = \{\sum_{n=0}^{k} c_n | c_n \in com(e_i) \bigwedge c_n \supset m_i\}, k = |com(e_i)|, m_i \in M$

$com(M_s) = \{\sum_{n=0}^{k} c_n | c_n \in sem(e_i) \bigwedge c_n \supset m_i\}, k = |sem(e_i)|, m_i \in M$

O Teorema de Bayes é definido por:

$$P(com(e_i)|com(M)) = \frac{P(com(M_c)|com(e_i))P(com(e_i))}{P(com(M_c)|com(e_i))P(com(e_i)) + P(com(M_s)|sem(e_i))P(sem(e_i))} \tag{4.1}$$

Algoritmo ImpalaMiner

Formalizada a aplicação do teorema de Bayes para calcular a probabilidade de uma entidade ser impactada por um conjunto de mudanças, apresentamos o código 4.1 do algoritmo ImpalaMiner. Ele recebe como entrada a matriz de *commits* H , o conjunto de mudanças M e o conjunto de impactos I e calcula a probabilidade de impacto cada entidade em I. Para aplicar o teorema de Bayes, o algoritmo calcula, primeiramente, os conjuntos $com(A)$, $sem(A)$, $com(M_c)$ e $com(M_s)$, formalizados anteriormente. A variável `total` representa o número total de *commits* (ou transações) existentes até o momento da análise de impacto, ou seja, `total`=$|C|$, representado pela variável `i`.

```
1  impalaMiner(H[i,j], I, M)
2    //H[i,j] = matriz, I = conj. de impactos, M = conj. de mudanças
3    PARA CADA e em I FAÇA
4      SE M contém e ENTÃO
5        P(e) <= 1
6      SENÃO
7        total <= i;
8        com(e), sem(e), com(Mc), com(Ms)=0
9        PARA Ci em H[i,j] FAÇA
10         SE Ci contém e ENTÃO
11           com(e) <= com(e) + 1;
12         SENÃO
13           sem(e) <= sem(e) + 1;
14         FIM_SE_SENÃO
15         contémM <= false;
16         PARA CADA m em M FAÇA
17           SE Ci contém m ENTÃO
18             contémM <= true;
19           FIM_SE
20         FIM_PARA
21         SE contémM = true ENTÃO
22           com(Mc) <= com(Mc) + 1;
23         SENÃO
24           sem(Mc) <= sem(Mc) + 1;
25         FIM_SE_SENÃO
26       FIM_PARA
27       P(e) <= ((com(Mc)/com(e)) * (com(e)/total)) /
28               (((com(Mc)/com(e)) * (com(e)/total)) +
29               ((com(Ms)/sem(e)) * (sem(e)/total)))
30     FIM_SE_SENÃO
31   FIM_PARA
```

Código 4.1: Pseudocódigo do algoritmo ImpalaMiner

As probabilidades calculadas pelo ImpalaMiner podem ser utilizadas pelo engenheiro de

software, durante o processo de análise de impacto, como um guia de prioridades que ordena as entidades impactadas. Entidades com probabilidades muito baixas podem ser descartadas, enquanto entidades com probabilidades maiores podem merecer uma análise mais detalhada.

Como a análise probabilística é totalmente baseada das informações históricas do software em análise, quanto maior for o histórico do software, menor o erro do cálculo da probabilidade. Por isso, a técnica de análise probabilística deve ser adotada para projetos com algum tempo de desenvolvimento. Esse tempo varia de acordo com o processo de desenvolvimento, o número de desenvolvedores, o tamanho do software, o comportamento da equipe, dentre outros fatores. Por exemplo, para os projetos avaliados na seção 5, três meses foi tempo suficiente para que houvesse um histórico de mudanças significativo para a aplicação do ImpalaMiner.

4.4 Exemplo de Aplicação da Técnica de Análise Probabilística

Para esclarecer a aplicação da técnica de análise probabilística, encerramos este capítulo com um exemplo que engloba as abordagens propostas neste e no capítulo 3. A figura 4.3 ilustra um programa que gerencia professores e estudantes de uma universidade.

Neste exemplo, analisamos o impacto no software em nível de entidade. Portanto, conjunto de entidades E desse programa é:

```
class Curso {
   private String id;
   private int creditos;
   private Set estudantes;
   private Professor p;
   Curso(String n; Professor pp; int c){
       id = n; p = pp;
       creditos = c; estudantes = new HashSet();
   }
   public void addEstudante(Estudante
     x){estudantes.add(x);}
   public String getId(){ return id; }
   public HashSet getEstudantes(){ return estudantes; }
   public Professor getProfessor(){ return p; }
   public void setProfessor(Professor pp){ p = pp; }
   public int getCreditos(){ return creditos; }
}
class Pessoa {
   private String nome;
   Pessoa(String n){ nome = n; }
   public String getNome() { return nome;}
   public String toString(){ return nome; }
}
class Professor extends Pessoa {
   private String departamento, sala;
   private Set cursos;
   Professor(String nome, String d, String numero){
      super(nome); departamento = d;
      sala = numero; cursos = new HashSet();
   }
   public void addCurso(Curso c){ cursos.add(c); }
   public int load(){ return cursos.size(); }
   public String toString(){
      return (super.toString() + ", sala é " +
      sala + " departamento é " + departamento);
   }
}
```

```
class Estudante extends Pessoa {
   private Set cursos;
   Estudante(String nome){
      super(nome);
      cursos = new HashSet();
   }
   public void addCurso(Curso c){ cursos.add(c); }
   public int totalCreditos(){
      int soma = 0;
      for (Iterator en=cursos.iterator(); en.hasNext();){
         Curso c = (Curso)en.next(); soma +=
    c.getCreditos();
      }
      return soma;
   }
}
class Universidade {
   private Set pessoas;
   Universidade(){ pessoas = new HashSet(); }
   public Set getPessoas(){ return pessoas; }
   public void addPessoa(Pessoa p){ pessoas.add(p);}
   public Professor findProfessor(String nome){
      for (Iterator en = pessoas.iterator(); en.hasNext();
       ){
          Pessoa p = (Pessoa)en.next();
          if(p instanceof Professor &&
    nome.equals(p.getNome()))
             return (Professor)p;
      }
      return null;
   }
   public Estudante findEstudante(String nome){
      for (Iterator en = pessoas.iterator(); en.hasNext();
       ){
          Pessoa p = (Pessoa)en.next();
          if (p instanceof Estudante ){
             Estudante e = (Estudante)p;
             if (nome.equals(e.getNome())
                 return e;
          }
      }
      return null;
   }
}
```

Figura 4.3: Exemplo de código de um programa de gerência de universidade

e_1	Curso.id
e_2	Curso.creditos
e_3	Curso.estudantes
e_4	Curso.Professor
e_5	Curso.Curso(String n,Professor pp,int c)
e_6	Curso.addEstudante(Estudante x)
e_7	Curso.getId()
e_8	Curso.getEstudantes()
e_9	Curso.getProfessor()
e_{10}	Curso.setProfessor(Professor pp)
e_{11}	Curso.getCreditos()
e_{12}	Pessoa.nome
e_{13}	Pessoa.Pessoa(String n)
e_{14}	Pessoa.getNome()
e_{15}	Pessoa.toString()
e_{16}	Professor.departamento
e_{17}	Professor.sala
e_{18}	Professor.cursos
e_{19}	Professor.Professor(String nome,String d,String numero)
e_{20}	Professor.addCurso(Curso c)
e_{21}	Professor.load()
e_{22}	Professor.toString()
e_{23}	Estudante.cursos
e_{24}	Estudante.Estudante(String nome)
e_{25}	Estudante.addCurso(Curso c)
e_{26}	Estudante.totalCreditos()
e_{27}	Universidade.pessoas
e_{28}	Universidade.Universidade()
e_{29}	Universidade.getPessoas()
e_{30}	Universidade.addPessoa(Pessoa p)
e_{31}	Universidade.findProfessor(String nome)
e_{32}	Universidade.findEstudante(String nome)

Queremos realizar as seguintes mudanças nesse código:

- remover o atributo Curso.creditos;
- mudar a semântica do método `Professor.toString()` para retornar apenas o nome: `super.toString()`;
- mudar a assinatura do método `Universidade.findProfessor(String nome)` para passar como parâmetro um objeto do tipo `Pessoa` ao invés de um `String`.

Portanto, o conjunto de mudanças M é:

e_2	remove Curso.creditos
e_{22}	changeSemantics Professor.toString()
e_{31}	changeSignatureParameter Universidade.findProfessor(String nome) to Universidade.findProfessor(Pessoa p)

O processo de aplicação da técnica de análise de impacto probabilística encontra-se na figura 4.4.

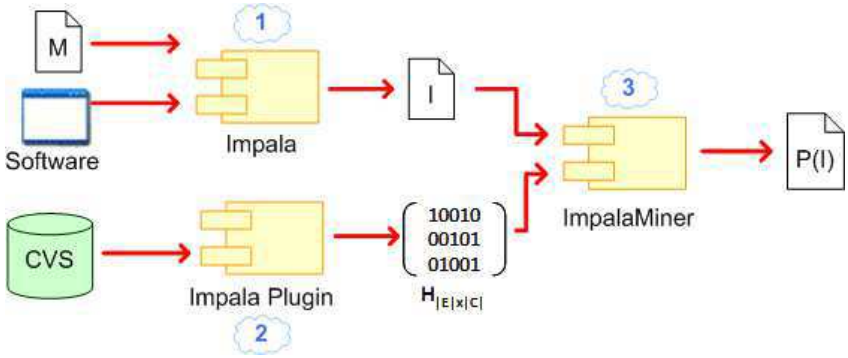


Figura 4.4: Processo da análise de impacto probabilística

O primeiro passo é executar os algoritmos de análise de impacto do Impala, fornecendo com entrada o conjunto M e o software. A saída obtida é o conjunto I:

e_2	Curso.creditos
e_5	Curso.Curso(String n,Professor pp,int c)
e_{11}	Curso.getCreditos()
e_{26}	Estudante.totalCreditos()
e_{22}	Professor.toString()
e_{15}	Pessoa.toString()
e_{31}	Universidade.findProfessor(String nome)

Portanto, o conjunto $I - M$ é:

e_5	Curso.Curso(String n,Professor pp,int c)
e_{11}	Curso.getCreditos()
e_{26}	Estudante.totalCreditos()
e_{15}	Pessoa.toString()

O segundo passo é extrair a matriz de históricos de mudanças $H_{[|C|\times|E|]}$ do CVS do programa. Como esse é um exemplo fictício, simulamos a extração das informações históricas e a geração da matriz H. Para não ultrapassar a margem da folha, mostramos a matriz transposta H^T apenas a titulo de ilustração. Portanto, suponha que a matriz gerada é:

$H^T =$

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	
0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	e_1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_2
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_3
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_4
0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	e_5
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_6
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_7
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_8
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_9
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{10}
0	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	e_{11}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{12}
1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	e_{13}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{14}
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	e_{15}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{16}
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	e_{17}
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	e_{18}
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	e_{19}
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	e_{20}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{21}
0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	0	0	0	e_{22}
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	e_{23}
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	e_{24}
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{25}
0	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	e_{26}
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	e_{27}
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{28}
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_{29}
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	e_{30}
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	e_{31}
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	e_{32}

Após a geração da matriz H, o último passo é calcular a probabilidade de cada entidade do conjunto I ser impactada. Como as entidades pertencentes ao conjunto $M \cap I$ mudarão com 100% de certeza, elas recebem P(e)=1. A seguir aplicamos o algoritmo ImpalaMiner nas entidades de $I - M$.

e_5 – Curso.Curso(String n,Professor pp,int c):

$$
\begin{aligned}
P(com(e_5)|com(M)) &= \frac{P(com(M_c)|com(e_5))P(com(e_5))}{P(com(M_c)|com(e_5))P(com(e_5))+P(com(M_s)|sem(e_5))P(sem(e_5))} \\
&\frac{\frac{4}{4} * \frac{4}{18}}{(\frac{4}{4} * \frac{4}{18}) + (\frac{1}{14} * \frac{14}{18})} \\
&\frac{4}{5} = 80\%
\end{aligned}
$$

e_{11} – Curso.getCreditos():

$$
\begin{aligned}
P(com(e_{11})|com(M)) &= \frac{P(com(M_c)|com(e_{11}))P(com(e_{11}))}{P(com(M_c)|com(e_{11}))P(com(e_{11}))+P(com(M_s)|sem(e_{11}))P(sem(e_{11}))} \\
&\frac{\frac{2}{5} * \frac{5}{18}}{(\frac{2}{5} * \frac{5}{18}) + (\frac{2}{13} * \frac{13}{18})} \\
&\frac{2}{4} = 50\%
\end{aligned}
$$

e_{26} – Estudante.totalCreditos():

$$
\begin{aligned}
P(com(e_{26})|com(M)) &= \frac{P(com(M_c)|com(e_{26}))P(com(e_{26}))}{P(com(M_c)|com(e_{26}))P(com(e_{26}))+P(com(M_s)|sem(e_{26}))P(sem(e_{26}))} \\
&\frac{\frac{3}{5} * \frac{5}{18}}{(\frac{3}{5} * \frac{5}{18}) + (\frac{3}{13} * \frac{13}{18})} \\
&\frac{3}{6} = 50\%
\end{aligned}
$$

e_{15} – Pessoa.toString():

$$
\begin{aligned}
P(com(e_{15})|com(M)) &= \frac{P(com(M_c)|com(e_{15}))P(com(e_{15}))}{P(com(M_c)|com(e_{15}))P(com(e_{15}))+P(com(M_s)|sem(e_{15}))P(sem(e_{15}))} \\
&\frac{\frac{2}{2} * \frac{2}{18}}{(\frac{2}{2} * \frac{2}{18}) + (\frac{4}{16} * \frac{16}{18})} \\
&\frac{2}{6} = 33\%
\end{aligned}
$$

Para finalizar o exemplo de aplicação da técnica de análise de impacto probabilística, apresentamos o resultado final, que é o conjunto de entidades impactadas I ordenado por probabilidade:

Curso.creditos	100%
Professor.toString()	100%
Universidade.findProfessor(String nome)	100%
Curso.Curso(String n,Professor pp,int c)	80%
Curso.getCreditos()	50%
Estudante.totalCreditos()	50%
Pessoa.toString()	33%

Esse exemplo foi criado exclusivamente para facilitar o entendimento do passo-a-passo da técnica apresentada neste trabalho. Portanto, os valores encontrados nele são fictícios e não faz sentido avaliá-los em mais profundidade. Porém, a técnica probabilística é avaliada no capítulo seguinte.

4.5 Considerações Finais

Neste capítulo, concluímos a apresentação da técnica de análise de impacto probabilística, explanando como utilizamos as informações históricas do software para atribuir probabilidades de impacto às entidades identificadas como impactadas pelos algoritmos de análise de impacto estática. Primeiramente, as informações históricas das mudanças estruturais do software, armazenadas no CVS, são extraídas e armazenadas no banco de dados. Em seguida, as transações de *commits* são encontradas, o que viabiliza a criação das matrizes que contêm todas as transações de mudanças ocorridas até o momento da análise. As duas matrizes fornecidas são em nível de classe e em nível de entidade. Por fim, o ImpalaMiner recebe como entrada uma dessas matrizes, o conjunto de mudanças e o conjunto de impactos e calcula, para cada entidade do conjunto de impactos, a probabilidade de ela ser impactada por pelo menos uma das entidades do conjunto de mudanças.

O principal objetivo de atribuir probabilidades de impacto é ponderar o resultado fornecido pelos algoritmos de análise de impacto para reduzir o número de falso-positivos gerados por eles. Essa redução pode ser feita de forma subjetiva, a partir da análise do engenheiro de software, ou de forma objetiva, através do uso de uma classificação que indique quais entidades devem ser ignoradas e quais devem ser consideradas. Essa classificação

deve ser criada a partir da análise de resultados coletados de experimentações da técnica de análise probabilística em softwares diversos. O capítulo 5 inclui tanto as análises das experimentações realizadas como parte da validação da técnica de análise probabilística, como uma proposta de classificação de falso-positivos.

Capítulo 5

Avaliação da Técnica Proposta

Este capítulo apresenta a avaliação da técnica de análise de impacto probabilística, descrita nos capítulos 3 e 4. Para avaliá-la, redefinimos duas métricas da recuperação de informação, precisão e revocação e as usamos para mensurar os falso-positivos e falso-negativos gerados, respectivamente. A avaliação foi realizada em duas etapas. Primeiramente, avaliamos a técnica de análise de impacto estática, proposta no capítulo 3. Para isso, parametrizamos os algoritmos de análise de impacto com o intuito de mensurar a quantidade de falso-positivos e falso-negativos gerada por eles. A segunda etapa consiste na avaliação da técnica de análise probabilística completa (análise estática + análise histórica), que envolve os conceitos apresentados nos capítulos 3 e 4. Para isso, criamos um estudo de caso para avaliar tarefas de modificação de dois projetos, tanto em relação às métricas precisão e revocação, quanto através de uma análise subjetiva dos resultados obtidos.

A avaliação da técnica de análise de impacto proposta neste trabalho tem como objetivo mensurar a quantidade de falso-positivos e falso-negativos gerados. Trabalhos anteriores sobre análise de impacto argumentam que técnicas estáticas geram muitos falso-positivos [Breech, Tegtmeyer e Pollock 2006; Apiwattanapong, Orso e Harrold 2005]. Porém, nenhum deles usa métricas claras e objetivas para mensurar a precisão das técnicas de análise de impacto. Por isso, redefinimos duas métricas da recuperação de informação [Rijsbergen 1979], precisão (*precision*) e revocação (*recall*), para medir a acurácia dos resultados obtidos pela técnica proposta. A definição das duas métricas é apresentada a seguir.

5.1 Precisão e Revocação

Precisão e revocação são duas medidas da recuperação de informação largamente usadas para avaliar soluções da mineração de dados [Tan, Steinbach e Kumar 2005]. Na recuperação de informação, precisão é a razão entre o número de documentos relevantes retornados e o número total de documentos retornados por uma consulta. Revocação é a razão entre o número de documentos relevantes retornados e o número total de documentos relevantes. A tabela de contingência relaciona os documentos retornados B aos documentos relevantes A [Rijsbergen 1979].

	Relevante	Não-relevante	
Retornado	$A \cap B$	$\overline{A} \cap B$	B
Não-retornado	$A \cap \overline{B}$	$\overline{A} \cap \overline{B}$	$\overline{B}$
	A	$\overline{A}$	

A partir da tabela acima, as equações da precisão e da revocação são:

$$P = \frac{|A \cap B|}{|B|} \qquad\qquad R = \frac{|A \cap B|}{|A|}$$

Antes de introduzir nossa releitura das definições de precisão e revocação definiremos falso-positivos e falso-negativos em relação ao conjunto de impactos I, definido na seção 4.3, e o conjunto de elementos realmente impactados W. Este representa os elementos que foram realmente modificados na prática. Uma premissa deste trabalho é que o conjunto W pode ser obtido do CVS que armazena o software em análise.

Falso-positivo: Seja I o conjunto de elementos impactados e W o conjunto de elementos realmente modificados, falso-positivo é um elemento que em I, que não está em W. Esse elemento foi localizado pelo Impala, mas não foi modificado quando o conjunto de mudanças proposto foi implementado.

$$\text{Falso-positivo} = I - W$$

Falso-negativo: Seja I o conjunto de elementos impactados e W o conjunto de elementos realmente modificados, falso-negativo é um elemento em W, que não está em I. Esse

elemento sofreu alteração quando o conjunto de mudanças proposto foi implementado, mas o Impala não foi capaz de identificá-lo.

$$\text{Falso-negativo} = W - I$$

A figura 5.1 ilustra o diagrama da Venn dos conjuntos I e W e mostra os falso-positivos e falso-negativos. Quanto maior for a interseção desses conjuntos, menor é o número de falso-positivos e falso-negativos.

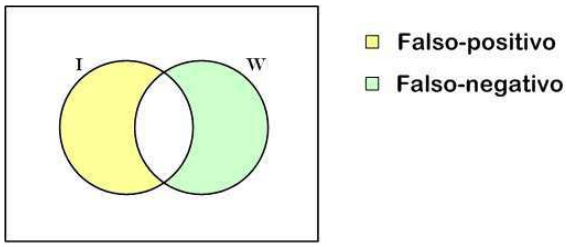


Figura 5.1: Diagrama de Venn dos conjuntos I e W

A seguir, redefinimos precisão e revocação em termos de impactos e relacionamos precisão a falso-positivos e revocação a falso-negativos.

Precisão: a precisão P indica qual fração das entidades identificadas como impactadas I foi realmente modificada W. Essa relação é definida pela seguinte equação:

$$P = \frac{|I \cap W|}{|I|}$$

A relação entre falso-positivo e precisão é: quanto menos falso-positivos são produzidos, maior é a precisão.

Revocação: a revocação R indica a proporção das entidades modificadas W que foi estimada corretamente como impactada I. Essa relação é definida pela equação:

$$R = \frac{|I \cap W|}{|W|}$$

A relação entre falso-negativo e revocação é: quanto menos falso-negativos são produzidos, maior é a revocação.

O objetivo é que os algoritmos de análise de impacto obtenham alta precisão e alta revocação. Isso significa que os algoritmos estimaram todas as entidades realmente impactadas e somente elas. Dessa forma, quanto maiores os valores da precisão e da revocação, maior a acurácia dos algoritmos do Impala. Portanto, consideramos que a acurácia é obtida pelo par de valores P e R.

Optamos por utilizar as métricas precisão e revocação, pois elas informam valores relativos de falso-positivos e falso-negativos, que possibilitam a comparação entre os resultados de diversos casos de teste. Como os conjuntos de modificações W e impactos I variam para cada alteração, seria complexo avaliar e comparar os valores absolutos dos conjuntos de falso-positivos ($|I - W|$) e falso-negativos ($|W - I|$).

5.2 Avaliação da Técnica de Análise de Impacto Estática

A avaliação da técnica de análise de impacto estática, vista no capítulo 3, tem como objetivo mensurar a quantidade de falso-positivos e falso-negativos gerados pelos algoritmos propostos. Essa avaliação está descrita no artigo "On the precision and accuracy of impact analysis techniques" e representa um resultado parcial deste trabalho, no qual utilizamos as métricas precisão e revocação para avaliar a acurácia dos algoritmos de análise de impactos propostos [Hattori et al. 2008].

A seguir, apresentamos a avaliação da técnica de análise de impacto estática através de sua aplicação em três softwares distintos. Para isso, primeiramente, fizemos uma alteração nos algoritmos de análise de impacto para permitir a poda da busca por impactos. A idéia da poda é de testar se as entidades que dependem diretamente das entidades mudadas sofrem mais impacto do que as entidades que dependem indiretamente. A partir dessa modificação, podemos avaliar os valores obtidos das métricas propostas para buscas que param em diferentes níveis do grafo de dependências das entidades do software. Em seguida, detalhamos o estudo empírico realizado através da aplicação dos algoritmos modificados a três softwares distintos. Por fim, avaliamos os resultados obtidos.

5.2.1 Modificação dos Algoritmos

Para avaliar a técnica de análise de impacto estática utilizando as métricas propostas, conduzimos um estudo empírico com três sistemas de software distintos. Os algoritmos do Impala, apresentados nas seções 3.4.1 a 3.4.4, realizam uma busca em profundidade dos impactos na hierarquia de dependências do software. Eles buscam por dependências diretas e indiretas exaustivamente. De forma intuitiva, os algoritmos do Impala deveriam gerar um grande número de falso-positivos, reduzindo a precisão e, conseqüentemente, a acurácia. Também intuitivamente, um algoritmo que busque apenas por dependências diretas de uma mudança deve gerar menos falso-positivos, mas tem uma chance maior de gerar falso-negativos.

Para responder a essas hipóteses, modificamos os algoritmos de análise de impacto para que haja um ponto de parada na busca por impactos. Assim, os algoritmos passaram a ter um parâmetro denominado profundidade. Portanto, se o parâmetro profundidade tem valor igual 1, o algoritmo de análise de impactos só buscará por entidades que tenham dependência direta com a mudança. Caso a profundidade seja igual a 2, o algoritmo buscará por entidades que tenham dependências direta e indireta com a distância[1] menor ou igual a dois.

O código 5.1 ilustra o algoritmo `ModificationCommand` alterado. Ele tem o parâmetro `profundidade` a mais, bem como a verificação da poda nas linhas 2 a 4. Os algoritmos `ChangeVisibilityCommand`, `AddInheritanceCommand` e `RemoveInheritanceCommand` foram modificados de forma semelhante para permitir que a busca por impactos pare, também, seguindo o critério da profundidade.

5.2.2 Estudo Empírico

Com o intuito medir a acurácia dos algoritmos do Impala, conduzimos um estudo empírico de análise de impacto utilizando três sistemas de software desenvolvidos em Java. A idéia geral do estudo é identificar mudanças que ocorreram no passado, aplicar a técnica de análise de impacto ao sistema no momento do tempo imediatamente anterior à mudança e comparar os resultados da técnica com o que foi realmente modificado. Portanto, comparamos os resultados obtidos do Impala com mudanças que realmente aconteceram no passado.

Para realizar o estudo, o próprio Impala foi um dos sistemas analisados, bem como o

[1] Considere o conceito de distância em grafos.

```
árvore modificationCommand(entidade, profundidade)
  SE profundidade = 0 ENTÃO
   retorna
  FIM_SE
  Árvore nó //nó da árvore que armazena o resultado
  NóMétodo método
  SE entidadesImpactadas não contém entidade ENTÃO
    nó.entidade <= entidade
    INCLUI(entidadesImpactadas, entidade) //lista global de impactos
    Relacionamento relacionamentos[] <= entidade.getRelacionamentos()
    PARA CADA relacionamento em relacionamentos[] FAÇA
      SE entidade.tipo = "atributo"
                  E relacionamento = "is\_accessedBy" ENTÃO
       método <= relacionamento.getChamador()
        INCLUI(nó.folhas[], modificationCommand(método, profundidade-1))
      SENÃO
        SE relacionamento = "is\_invokedBy" ENTÃO
          método <= relacionamento.getChamador()
          INCLUI(nó.folhas[], modificationCommand(
                                  método, profundidade-1))
        FIM_SE
      FIM_SE_SENÃO
    FIM_PARA
  FIM_SE
  retorna nó
```

Código 5.1: Algoritmo ModificationCommand com o parâmetro profundidade

DesignWizard, que também faz parte de nossa solução [Brunet e Guerrero 2007]. O OurGrid, sistema para a execução de aplicações *Bag-Of-Task* (BoT) em grades computacionais, foi o terceiro software escolhido [Cirne et al. 2006]. A tabela 5.1 resume algumas informações importantes sobre os três sistemas.

Queremos investigar a acurácia dos algoritmos do Impala nesses três sistemas, comparando os algoritmos originais aos algoritmos modificados. Para os algoritmos modificados, limitaremos a busca às profundidades 1, 2, 3 e 6. Para a avaliação, chamaremos essas con-

Tabela 5.1: Projetos selecionados para o estudo empírico

Nome	**Descrição**	**LOCs**	**Número de classes**
Impala	Ferramenta de análise estática	1.584	45
DesignWizard	API para inspeção automática de programas em Java	3.644	44
Ourgrid	Sistema para execução de aplicações BoT em grades computacionais	48.752	627

figurações de:

- A_∞ – algoritmos originais;
- A_1 – algoritmos modificados que buscam apenas por dependências diretas;
- A_2 – algoritmos modificados que buscam por dependências diretas e indiretas até a profundidade 2;
- A_3 – algoritmos modificados que buscam por dependências diretas e indiretas até a profundidade 3;
- A_6 – algoritmos modificados que buscam por dependências diretas e indiretas até a profundidade 6.

A aplicação do Impala nos três softwares foi feita seguindo a metodologia descrita a seguir e ilustrada na figura 5.2.

1. Escolher de forma aleatória 5 a 15% das classes do software para serem analisadas.
2. Para cada classe, identificar, no sistema de controle de versão (CVS), as revisões que incorporaram pelo menos uma mudança estrutural, que é uma adição, remoção ou alteração de uma classe, um atributo ou um método.
3. Para cada revisão, obter todas as mudanças estruturais da revisão em questão. Essas mudanças compõem o conjunto de mudanças M.
4. Para cada conjunto M, aplicar o Impala para obter os conjuntos das entidades possivelmente impactadas: E_∞, E_1, E_2, E_3 e E_6.
5. Para cada conjunto E_n de elementos impactados, obter o conjunto das classes que contêm E_n: I_∞, I_1, I_2, I_3 e I_6.
6. Para cada conjunto M, extrair do CVS o que foi realmente modificado W por causa das mudanças em M.
7. Para cada I_n e o conjunto W correspondente, calcular a precisão e a revocação.

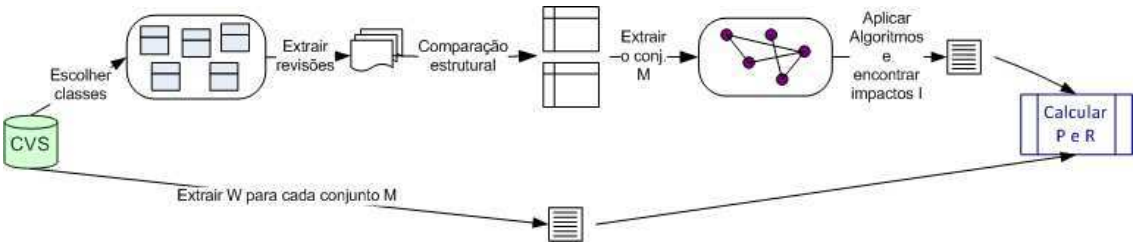


Figura 5.2: Passos da metodologia usada para avaliar os algoritmos do Impala

A seleção das classes do conjunto W é feita por uma pessoa, manualmente, comparando as revisões das classes relacionadas à classe modificada seguindo o critério: compõem o conjunto W todas as classes que foram modificadas no período entre a revisão na qual as mudanças do conjunto M (mudanças na classe analisada) ocorreram e a próxima revisão da classe e que têm alguma relação de dependência com a classe analisada. Isso significa que pode haver classes em W que foram modificadas por motivos não relacionados às mudanças na classe analisada. Por exemplo, mudança em outra classe do sistema, ou mudança nos requisitos da própria classe.

A figura 5.3 ilustra um exemplo de como os conjuntos M e W são extraídos do CVS ao longo do tempo. Suponha que a classe `Foo` foi escolhida para ser analisada. Como se pode ver, ela possui quatro revisões. Portanto, há três conjuntos de mudanças a serem analisados: $M1.1$ referente à versão V1.1 e o respectivo conjunto $W1.1$, extraído no intervalo indicado na figura; $M1.2$, referente à versão V1.2 e $W1.2$; e $M1.3$, referente à versão V1.3 e $W1.3$.

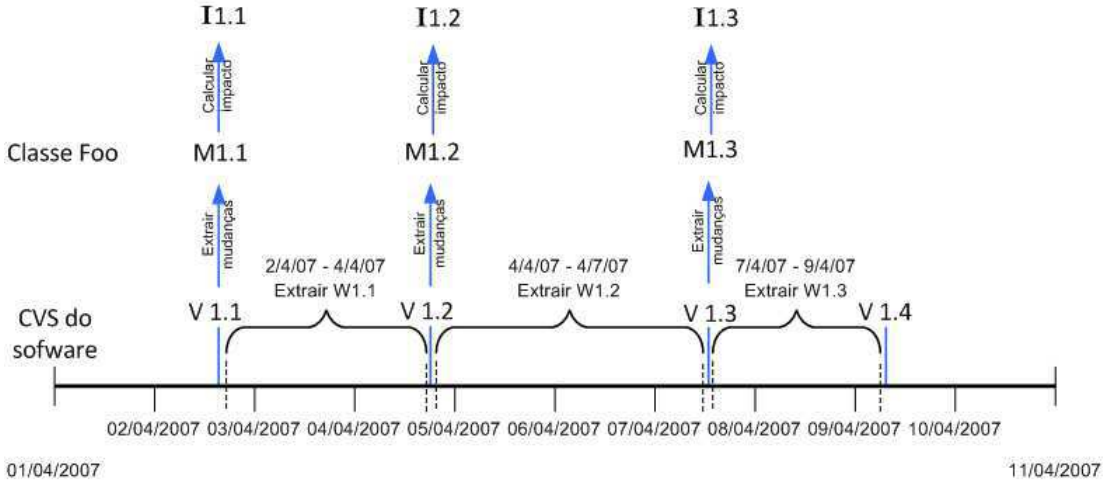


Figura 5.3: Linha do tempo ilustrando as revisões da classe Foo

A comparação entre as seis configurações dos algoritmos é feita em nível de classe, apesar de os algoritmos do Impala identificarem os impactos em nível de entidade. Em outras palavras, calculamos P e R em relação às classes impactadas I_n e o conjunto de classes realmente modificadas W. Essa opção foi escolhida, pois o Impala *Plug-in*, capaz de extrair informações das mudanças em nível de entidade do CVS, ainda não tinha sido implementado quando esse estudo empírico foi realizado. Porém, a avaliação da técnica de análise probabilística, que será apresentada na seção 5.3, já realiza a comparação em nível de entidade.

Por fim, esclarecemos que simulamos a análise de impacto para mudanças que já ocorreram no passado e comparamos os resultados da análise com o que foi realmente modificado. Todas as informações extraídas do CVS são dados passados, inclusive os conjuntos de mudanças propostas. Os casos de teste não foram elaborados e aplicados durante o período de desenvolvimento dos softwares analisados. Assim, podemos classificar nossa análise como *post mortem*, após o período de desenvolvimento.

5.2.3 Avaliação dos Resultados

A metodologia descrita na seção 5.2.2 foi aplicada a cada conjunto de mudanças, de cada classe escolhida, nos três projetos.

A tabela 5.2 resume os resultados obtidos para cada projeto através da apresentação da média, mediana e desvio padrão para as medidas de precisão e revocação. Ela informa, também, o total de casos analisados, representados pelo total de conjuntos de mudanças, e o total de modificações que realmente ocorreram para cada projeto. Estes pares de números têm o objetivo de fornecer a dimensão do estudo empírico, visto que os conjuntos de classes modificadas W são obtidos por um processo manual bastante trabalhoso.

Os valores da mediana encontrados para a revocação foram iguais a um para todas as configurações dos algoritmos. Isso significa que, na maioria dos conjuntos de mudanças, todas as classes realmente modificadas foram encontradas pelos algoritmos do Impala. Resultado semelhante pode ser observado para a precisão, exceto no próprio projeto Impala, no qual a mediana da precisão apresenta-se abaixo de 0,40 para as configurações A_2 a A_∞. Assim, pode-se afirmar que, diferentemente dos outros dois projetos, essas configurações geraram muitos falso-positivos ao serem aplicadas ao projeto do Impala.

Tabela 5.2: Média, mediana e desvio padrão da precisão e da revocação para os três projetos avaliados. TC = número total de conjuntos de mudanças analisado, TM = total de mudanças que realmente ocorreram. Estat. = estatísticas. D. Padrão = desvio padrão

Projetos	**TC**	**TM**	**Estat.**	**Profundidade**									
				1		2		3		6		∞	
				P	R	P	R	P	R	P	R	P	R
OurGrid	81	249	Média	0,88	0,80	0,86	0,93	0,86	0,98	0,85	1,00	0,85	1,00
			Mediana	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
			D. Padrão	0,20	0,3	0,22	0,18	0,22	0,07	0,22	0,00	0,22	0,00
DesignWizard	21	42	Média	0,89	0,81	0,70	0,95	0,67	1,00	0,67	1,00	0,67	1,00
			Mediana	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
			D. Padrão	0,21	0,30	0,38	0,15	0,40	0,00	0,40	0,00	0,40	0,00
Impala	20	37	Média	0,70	0,90	0,49	0,96	0,45	0,99	0,41	1,00	0,41	1,00
			Mediana	1,00	1,00	0,37	1,00	0,36	1,00	0,25	1,00	0,25	1,00
			D. Padrão	0,36	0,33	0,32	0,13	0,31	0,04	0,32	0,00	0,32	0,00

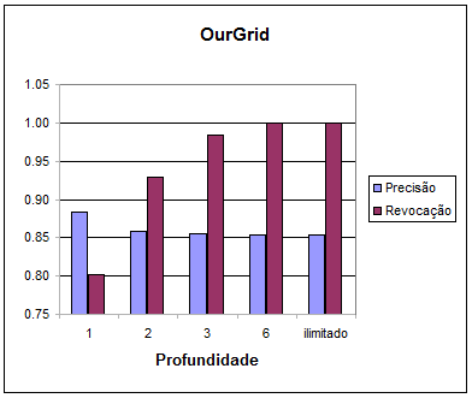
O desvio padrão da precisão calculado para as cinco configurações ficou abaixo de 0,40. Isso significa que a maioria dos valores de P ficou bastante próxima das respectivas médias. É possível notar, também que quanto mais profunda foi a busca, maior foi o desvio padrão de P. Como a média da precisão foi bastante alta na maioria dos casos (exceto no próprio Impala), podemos afirmar, que quanto mais profunda foi a busca, maior o número de casos com P menor do que a média.

O desvio padrão da revocação manteve-se mais baixo ainda, não ultrapassando o valor de 0,33. Assim, também podemos afirmar que a maioria dos valores de R ficou muito próxima das respectivas médias. Podemos observar que o comportamento inverso ao da precisão ocorre para a revocação: quando menos profunda é a busca, maior é o desvio padrão.

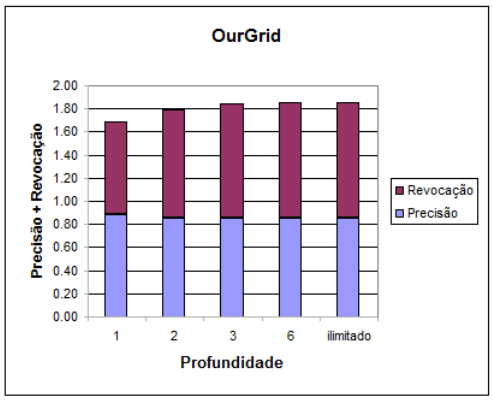
A partir das análises do desvio padrão, podemos utilizar a média como um valor significante para avaliar mais detalhadamente os resultados obtidos pelos algoritmos em cada projeto, individualmente. Dessa forma, discutiremos a acurácia dos algoritmos do Impala baseados nas médias calculadas para cada configuração dos algoritmos.

A figura 5.4 ilustra dois gráficos para cada projeto: os gráficos (a), (c) e (e) mostram o comportamento das médias da precisão e revocação de acordo com cada profundidade; já os gráficos (b), (d) e (f) exibem a soma da precisão e da revocação com o intuito de ilustrar qual configuração obteve a melhor acurácia ao considerar as duas medidas juntas.

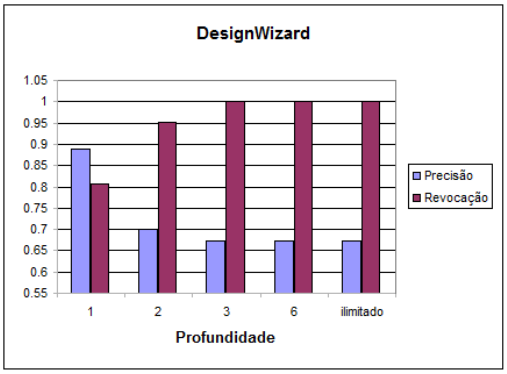
Através dos gráficos (a), (c) e (e), observamos que a precisão dos algoritmos diminui



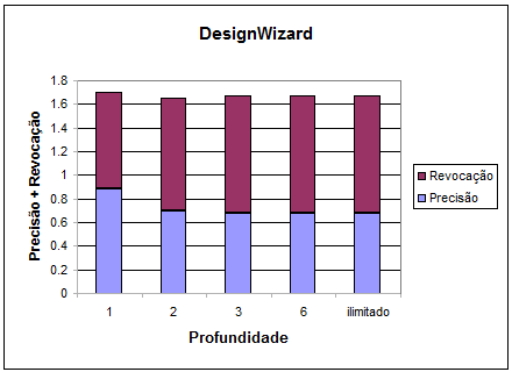
(a) Comportamentos de P e R do OurGrid



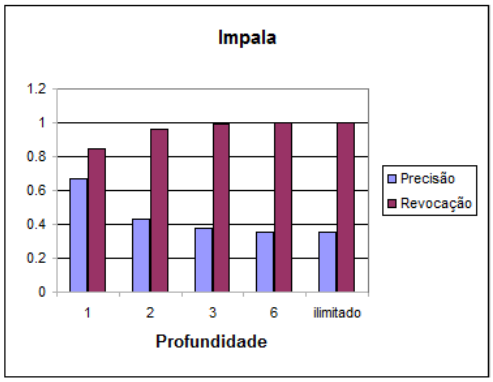
(b) Soma de P e R do OurGrid



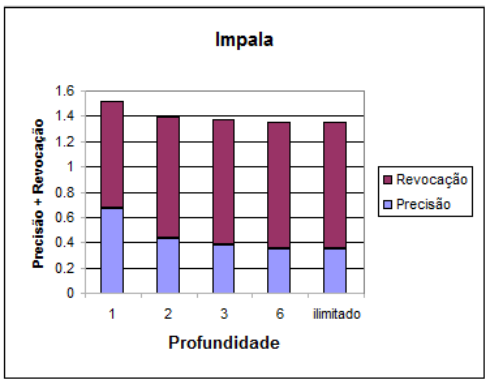
(c) Comportamentos de P e R do DesignWizard



(d) Soma de P e R do DesignWizard



(e) Comportamentos de P e R do Impala



(f) Soma de P e R do Impala

Figura 5.4: Precisão e revocação dos três projetos avaliados

com o aumento da profundidade da busca. Isso significa que as entidades com dependências indiretas em relação às mudanças são menos propícias a serem impactadas por elas. Assim, precisão de $A_1 \geq$ precisão de $A_2 \geq$ precisão de $A_3 \geq$ precisão de $A_6 \geq$ precisão de A_∞ na média.

O comportamento inverso é observado para a revocação: a revocação aumenta com o aumento da profundidade da busca. Revocação de $A_1 \leq$ revocação de $A_2 \leq$ revocação de $A_3 \leq$ revocação de $A_6 \leq$ revocação de A_∞ na média. Isso significa que quanto mais profunda for a busca, menos falso-negativos e mais falso-positivos são gerados.

Diante desses comportamentos, concluímos que o aumento da precisão está relacionado com a redução da revocação e vice-versa. No entanto, não foi possível determinar, a partir dos gráficos (b), (d) e (f), com qual profundidade de busca os algoritmos geraram menos falso-positivos e falso-negativos. Para o OurGrid, os algoritmos alcançaram maior acurácia quando buscaram com maior profundidade. Já no Impala, o comportamento inverso foi observado. Porém, no DesignWizard, observou-se um equilíbrio na soma da precisão e da revocação.

Apesar de não termos encontrado uma resposta para qual configuração gera menos falso-positivos e falso-negativos, observamos que os valores obtidos para precisão e revocação foram elevados. Com exceção da precisão para as configurações A_2 a A_∞ do projeto Impala, todas as médias de precisão e revocação foram acima de 0,65. Em média, a soma dos falso-positivos e falso-negativos gerados ficou entre 10 e 20% para o OurGrid e o DesignWizard, e entre 35 e 25% para o Impala.

Por fim, ao analisar os resultados de forma subjetiva, concluímos que a escolha entre buscar apenas por dependências diretas ou considerar as dependências indiretas está relacionada com o objetivo da análise de impacto. Se ela estiver sendo utilizada para guiar mudanças, é mais interessante que o desenvolvedor obtenha todos os possíveis impactos, mesmo que essa lista contenha muitos falso-positivos. Afinal, o desenvolvedor conhece o código do software e pode discernir entre os impactos relevantes e os irrelevantes. O mesmo raciocínio é válido para o caso de um engenheiro de software que está analisando o software para propor uma reestruturação ou uma refatoração. No entanto, se a análise de impacto objetiva estimativa de custos, é crucial que o erro dos algoritmos seja reduzido ao máximo. Nesse caso, se muitos falso-positivos são gerados, o custo da mudança será muito elevado, inibindo a aprovação do

cliente. A geração de muitos falso-negativos significará prejuízo para o desenvolvedor.

Dessa forma, optamos por manter a alteração dos algoritmos para que eles possam ser utilizados de acordo com seu objetivo final. Porém, propomos uma forma de ponderar a probabilidade de as entidades impactadas sofrerem alterações com base no histórico de mudanças do software. Essa proposta foi apresentada no capítulo 4 e será avaliada a seguir. O intuito dessa proposta é, utilizando busca exaustiva, ponderar e ordenar os resultados da análise de impacto para auxiliar o desenvolvedor ou o engenheiro de software a distinguir os impactos relevantes aos irrelevantes.

5.3 Avaliação da Técnica de Análise Probabilística

A avaliação da técnica de análise probabilística tem como objetivo aplicar as técnicas apresentadas nos capítulos 3 e 4 e discorrer sobre a hipótese levantada neste trabalho: *É possível aumentar a precisão da análise de impacto realizada antes da implementação de uma mudança, através do uso de informações históricas do comportamento das mudanças do sistema.* Para isso, realizamos um estudo de caso com os projetos DesignWizard e Impala, que será descrito na seção 5.3.1. O OurGrid não foi utilizado, pois, para esse estudo de caso, precisamos do conhecimento do código a ser analisado. Como a equipe de desenvolvimento do OurGrid mudou bastante ao longo dos anos, não havia uma pessoa que conhecesse todo o histórico dele para avaliar os resultados. Já o DesignWizard e o Impala são projetos desenvolvidos por apenas uma e três pessoas, respectivamente, que ainda estão presentes no grupo de pesquisa. Por isso, escolhemos manter apenas o DesignWizard e o Impala para esta segunda etapa da avaliação.

O cenário ideal para essa avaliação seria aplicá-la ao software ao longo do seu processo de desenvolvimento. Ou seja, aplicá-la antes de realizar uma mudança, coletar os resultados e compará-los às alterações feitas posteriormente. Isso deveria se repetir a cada mudança ao longo do tempo e, ao final do ciclo de desenvolvimento do software, essas informações seriam avaliadas. No entanto, uma avaliação desse tipo dura em torno de seis meses, o que inviabiliza sua realização neste trabalho. Portanto, devido à restrições de tempo, realizamos uma análise *post mortem* similar à anterior. Porém, como este trabalho é parte de um projeto maior, essa avaliação ainda deve ser feita futuramente.

Semelhante à análise anterior, utilizamos as métricas precisão e revocação para avaliar os resultados obtidos por esse estudo de caso. Para complementar a avaliação, realizamos, também, uma avaliação subjetiva, com o objetivo de responder se a ponderação das entidades impactadas agrega conhecimento útil para o desenvolvedor e o engenheiro de software.

5.3.1 Estudo de Caso

Neste estudo de caso, avaliamos a técnica de análise probabilística em dois níveis: de entidade (atributo e método) e de classe. No estudo empírico anterior, avaliamos os algoritmos de análise de impacto apenas em nível de classe. Porém, nesse estudo, avaliamos poucas mudanças, mas de forma detalhada. Para isso, extraímos o histórico de *commits* do CVS dos dois projetos e separamos os *commits* em transações de *commits* de acordo com a definição usada no capítulo 4. Utilizamos também uma metodologia para aplicar o Impala e o ImpalaMiner nesse estudo, que é diferente da metodologia empregada na avaliação da técnica de análise de impacto. Ela será descrita a seguir e alguns passos são ilustrados na figura 5.5.

1. Separar o período de desenvolvimento do software em duas partes: período de treinamento e período de teste.

2. Extrair o histórico do período de testes e organizá-lo em transações de *commits*; feita automaticamente pelo Impala *Plug-in*.

3. Extrair as matrizes de histórico de mudanças H_C (nível de classes) e H_E (nível de entidades) de acordo com a definição da seção 4.2.

4. Para cada transação de *commits*, identificar se existe uma tarefa de modificação. Essa tarefa consiste em duas ou mais entidades das quais, pelo menos uma foi a causa da modificação e pelo menos uma foi conseqüência.

5. Para cada tarefa de modificação:

   (a) Separar em dois conjuntos: mudanças M, composta pelas entidades causadoras; impactos reais I_R, composta pelas entidades modificadas por consequência de mudanças nas entidades em M.

   (b) Obter conjunto de classes realmente impactadas C_R a partir de I_R.

(c) Submeter M ao Impala para obter o conjunto de possíveis entidades impactadas I_E.

(d) Submeter M, I_E e H_E ao ImpalaMiner para obter as probabilidades das entidades impactadas, que chamamos de $I_{P(E)}$.

(e) Obter M_C e I_C a partir de M e I_E, respectivamente. Esses novos conjuntos são compostos apenas pelas classes que contêm as entidades de M e I_E.

(f) Submeter M_C, I_C e H_C ao ImpalaMiner para obter as probabilidades das classes impactadas, que chamamos de $I_{P(C)}$.

6. Comparar I_R com $I_{P(E)}$ e C_R com $I_{P(C)}$ em termos de precisão e revocação.

O primeiro passo é necessário, pois a análise histórica só atribuirá probabilidades às entidades que já tiverem sido validadas (*commited*) juntas no passado. Dessa forma, quanto mais transações de *commits* houverem, mais provável é que o ImpalaMiner encontre dependências históricas precisas entre os resultados da análise estática e as mudanças.

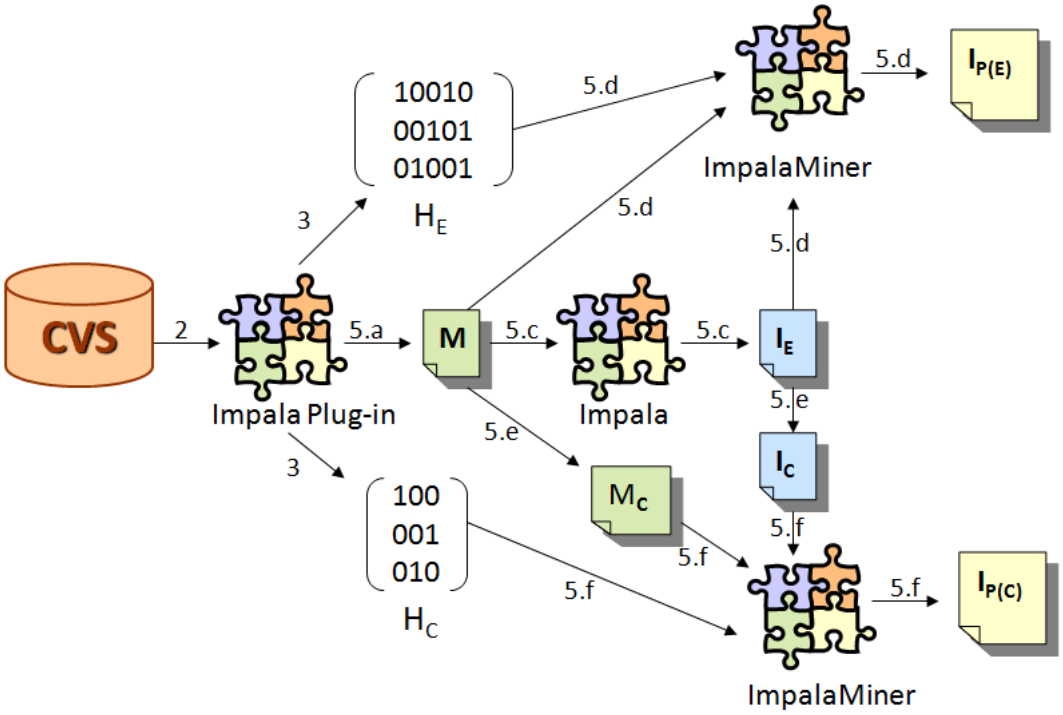


Figura 5.5: Passos da metodologia usada para avaliar a análise probabilística

A precisão e a revocação são calculadas da seguinte forma:

1. Para entidade:

   a. Considerando somente a análise estática:

   $$P = \frac{|I_E \cap I_R|}{|I_E|} \qquad R = \frac{|I_E \cap I_R|}{|I_R|}$$

   b. Considerando a análise probabilística – $I_{P(E)}$, sendo que as entidades com probabilidade igual a zero são excluídas do conjunto $I_{P(E)}$:

   $$P = \frac{|I_{P(E)} \cap I_R|}{|I_{P(E)}|} \qquad R = \frac{|I_{P(E)} \cap I_R|}{|I_R|}$$

2. Para classe:

   a. Considerando somente a análise estática:

   $$P = \frac{|I_C \cap C_R|}{|I_C|} \qquad R = \frac{|I_C \cap C_R|}{|C_R|}$$

   b. Considerando a análise probabilística – $I_{P(C)}$, sendo que as entidades com probabilidade igual a zero são excluídas do conjunto $I_{P(C)}$:

   $$P = \frac{|I_{P(C)} \cap C_R|}{|I_{P(C)}|} \qquad R = \frac{|I_{P(C)} \cap C_R|}{|C_R|}$$

A tabela 5.3 apresenta os períodos de desenvolvimento e de teste para os dois projetos, bem como o número de tarefas de modificação encontras em cada um. O período de treinamento do DesignWizard foi de apenas três meses porque seu desenvolvimento foi intenso nos primeiros meses, enquanto o desenvolvimento do Impala tornou-se mais intenso no quarto mês com a entrada de mais uma pessoa na equipe. Todas as tarefas de modificação encontradas no período de teste foram consideradas e, apesar de parecerem poucas, elas refletem o tamanho de cada software (tabela 5.1) e o tamanho da equipe de desenvolvimento.

Antes de avaliar os resultados, algumas limitações sobre esse estudo de caso devem ser explicadas. Primeiramente, o Impala *Plug-in* é capaz de extrair o histórico do CVS a partir da versão atual do software. Isso significa que classes que já foram excluídas no passado não aparecem no histórico extraído, pois elas não fazem mais parte da versão atual. Como estamos realizando uma análise *post mortem* dos softwares, informações históricas, que poderiam ter sido extraídas se o Impala *Plug-in* tivesse sido utilizado imediatamente antes da

Tabela 5.3: Períodos de desenvolvimento e de teste dos projetos e número de tarefas de modificação analisadas

Nome	**Período de desenvolvimento**	**Período de teste**	**N. de tarefas de modificação**
Impala	09/04/2007-03/12/2007 (8 meses)	08/08/2007-03/12/2007 (4 meses)	12
DesignWizard	01/04/2007-14/11/2007 (7 meses)	03/07/2007-14/11/2007 (4 meses)	15

realização de uma tarefa de modificação, são perdidas por nossa análise. Essa limitação não ocorre quando a solução é utilizada na prática antes da implementação de uma mudança.

Por fim, é preciso estar claro que o DesignWizard e o Impala foram escolhidos para esse estudo de caso porque precisávamos de pessoas com conhecimento de todo o decorrer do desenvolvimento do sistema para localizar as tarefas de modificação passadas. No entanto, os dois softwares caracterizam apenas uma parcela no que diz respeito a processo de desenvolvimento, tamanho do software, cultura e tamanho da equipe. Um software com milhões de linhas de código, desenvolvido por dezenas de pessoas que têm cultura de manter o repositório sempre atualizado e sem erros, terá um histórico de mudanças proporcionalmente maior.

5.3.2 Avaliação dos Resultados

A metodologia descrita na seção 5.3.1 foi aplicada a cada tarefa de mudanças nos dois projetos. A tabela 5.4 apresenta a média, a mediana e o desvio padrão para quatro valores de precisão e revocação: somente análise estática (Impala) em nível de entidade, análise probabilística em nível de entidade (Impala e ImpalaMiner), somente análise estática em nível de classe e análise probabilística em nível de classe. Lembramos que o Impala é sempre aplicado em nível de entidade e que são extraídas apenas as classes do resultado para e comparação em nível de classe. Também, enfatizamos que o conjunto $I_{P(E)} \subseteq I_E$, pois as entidades que tiveram probabilidade igual a zero foram eliminadas de $I_{P(E)}$. O mesmo se aplica para $I_{P(C)}$ em relação a I_C.

Comparando com o estudo anterior, as médias da precisão e da revocação da análise estática em nível de classe foram um pouco menores. Essa diferença existe porque o critério

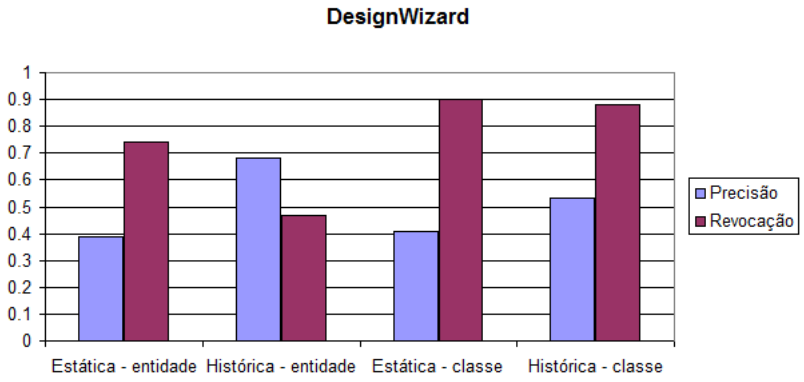
Tabela 5.4: Média, mediana e desvio padrão da precisão e da revocação para os dois projetos avaliados

Projetos	**Estatísticas**	**Entidade**				**Classe**			
		Estática		**Histórica**		**Estática**		**Histórica**	
		P	R	P	R	P	R	P	R
DesignWizard	Média	0,39	0,74	0,68	0,47	0,41	0,90	0,53	0,88
	Mediana	0,14	0,80	0,80	0,46	0,22	1,00	0,44	1,00
	Desv. Padrão	0,40	0,27	0,36	0,28	0,40	0,26	0,39	0,29
Impala	Média	0,17	0,78	0,70	0,60	0,33	0,85	0,53	0,85
	Mediana	0,16	0,90	0,70	0,60	0,25	1,00	0,50	1,00
	Desv. Padrão	0,12	0,26	0,30	0,30	0,25	0,20	0,20	0,20

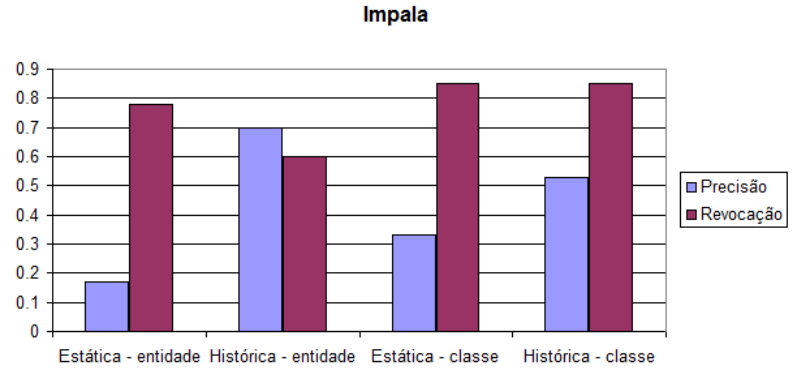
para a seleção do conjunto de mudanças e do conjunto de impactos mudou. Ele ficou mais seletivo, pois deixou de considerar todas as classes que foram validadas até o próximo *commit* da classe para contemplar apenas entidades que estão no mesmo *commit* das mudanças. Também em relação à média, os valores encontrados na análise em nível de entidade foram inferiores aos encontrados na análise em nível de classe. Discutiremos essa diferença na análise subjetiva.

Com relação à mediana, apesar de os valores para o nível de classe terem se mantido altos, os valores em nível de entidade foram menores, principalmente os da análise estática pura. Isso significa que, apesar de a técnica acertar em grande parte das recomendações em nível de classe, quando essas recomendações são detalhadas para o nível das entidades envolvidas, ela acerta menos. Já o desvio padrão se manteve abaixo de 0,40, o que indica que a média é uma boa medida de referência para uma análise mais profunda dos resultados.

Utilizando os valores das médias da precisão e da revocação, construímos os gráficos da figura 5.6 com o intuito de comparar os resultados das análises estática e histórica (probabilística). Percebe-se que, nos dois níveis, a precisão da análise histórica é maior do que da análise estática. Dessa forma, podemos afirmar que a análise histórica gera menos falso-positivos do que a análise estática pura. Em contrapartida, o inverso ocorre com a revocação: ela é menor na análise histórica, o que significa que mais falso-negativos são gerados. No entanto, observando os gráficos, ou calculando o módulo da diferença entre P estática e P histórica e entre R estática e R histórica, concluímos que a redução no número de falso-positivos é maior do que o aumento no número de falso-negativos para todos os casos.



(a) Resultado da análise do DesignWizard



(b) Resultado da análise do Impala

Figura 5.6: Comparação da precisão e revocação obtidas através da aplicação do Impala e do Impala + ImpalaMiner em níveis de entidade e classe dos dois projetos avaliados

A partir desse estudo de caso, podemos afirmar que os resultados da análise histórica foram melhores do que os resultados da análise estática em termos de precisão. Porém, ainda não interpretamos os resultados da análise histórica em relação ao valor da probabilidade encontrada para cada entidade ou classe. Apenas consideramos que uma entidade faria parte do conjunto $I_{P(E)}$ se ela tivesse uma probabilidade de impacto maior do que zero e o equivalente para classe. A seguir, analisamos, de forma subjetiva, as tarefas de modificação e discutiremos sobre os valores encontrados das probabilidades de impacto.

Avaliação Subjetiva das Tarefas de Modificação

Nessa avaliação, analisamos as entidades que pertencem ao conjunto de impactos, mas não pertence ao conjunto de mudanças. Lembramos que o conjunto de mudanças $M \in I$, portanto, se a entidade $e \in M$ então $P(e) = 1$. Portanto, analisamos apenas as entidades $e \in I - M$ cujas probabilidades são calculadas pelo ImpalaMiner.

No âmbito geral, três situações distintas podem ocorrer com uma entidade $e \in I - M$. Ela pode não ser identificada pela análise estática e, por conseqüência, também não ser ponderada pela análise histórica. Outra situação é a entidade ser identificada pela análise estática, mas não ter nenhuma dependência histórica com as mudanças em M, o que acarretará em $P(E) = 0$. Por último, e pode ser identificada pela análise estática e ter dependência histórica, o que produzirá $0 < P(e) < 1$. A seguir, avaliaremos os dois projetos, separadamente, de acordo com essas três situações.

DesignWizard

Das quinze tarefas de modificação do DesignWizard, três não tiveram nenhuma entidade impactada identificada tanto pela análise estática quanto pela análise histórica, cinco tiveram entidades identificadas apenas pela análise estática e sete tiveram entidades identificadas tanto pela análise estática quanto pela histórica.

No caso das tarefas que tiveram entidades impactadas identificadas tanto pela análise estática quando pela análise histórica, a explicação é que todas envolvem adição de entidade. Quando uma entidade está sendo adicionada no *commit* da tarefa de modificação, não existia, anteriormente, nenhuma dependência estática ou histórica entre ela e outras enti-

dades do sistema. Portanto, nos casos de adição, a análise probabilística nada pode afirmar. Houve, também, um caso em que a dependência estática entre a entidade modificada e sua impactada estava sendo criada na tarefa de modificação corrente. Mais especificamente, o método (impactado) passou a realizar uma chamada ao método modificado. Dessa forma, a análise probabilística não é capaz de auxiliar quando as mudanças são do tipo adição.

Para as tarefas de modificação em que os impactos foram identificados pela análise estática, mas não pela análise histórica, grande parte foi conseqüência de não existirem *commits* anteriores nos quais eles aparecessem com pelo menos uma das mudanças. Para esses casos, algumas análises probabilísticas em nível de classe encontraram probabilidades maiores do que zero. Por exemplo, a tarefa #DW-T4, com:

$$
\begin{aligned}
M = \;& \{\texttt{designwizard.design.MethodNode.getParentClass()}\} \\
I - M = \;& \{\texttt{designwizard.design.MethodNode.getClassesUsedBy()}, \\
& \texttt{designwizard.design.MethodNode.extractClasses(Set)}, \\
& \texttt{designwizard.design.FieldNode.getClassesThatUse()}, \\
& \texttt{designwizard.design.ClassNode.getClassesThatUse()}\}
\end{aligned}
$$

Todas as quatro entidades são encontradas pela análise estática, mas suas probabilidades são iguais a zero. No entanto, em nível de classe, encontramos P(`ClassNode`[2])=0,5 e P(`FieldNode`)=0,33. Apesar de serem probabilidades baixas, elas indicam que há algum acoplamento entre as classes `MethodNode`, `ClassNode` e `FieldNode`. O que, de fato, acontece, pois elas são implementações da mesma interface: `Entity`.

A tarefa #DW-T5 está o grupo das que tiveram entidades localizadas somente pela análise estática por causa da limitação da forma de aplicar nosso estudo de caso. Os conjuntos dessa tarefa são:

$$
\begin{aligned}
M = \;& \{\texttt{designwizard.test.impact.EARFilesTest}\} \\
I - M = \;& \{\texttt{designwizard.test.impact.EARFilesTest.main(String[])}, \\
& \texttt{designwizard.test.impact.EARFilesTest.testGetDescription()}\}
\end{aligned}
$$

A classe `EARFilesTest` foi removida do projeto em algum momento entre o *commit* analisado e a data final do desenvolvimento (quando o Impala *Plug-in* foi executado). Por isso, o Impala *Plug-in* não consegue obter as informações históricas dela. Porém, se

[2]As informações de pacote (`designwizard.design`) foram retiradas para simplificar a explicação. Essa simplificação será utilizada, também, nos próximos exemplos.

essa análise tivesse sido executada antes da mudança, esse problema não aconteceria e as probabilidades das entidades poderiam ter sido maiores do que zero.

Das sete tarefas restantes, quatro tiveram apenas parte de seus impactos identificados pelas duas análises. Porém, todas as entidades ponderadas pela análise histórica tiveram probabilidades altas, entre 50 e 100%. Para exemplificar esses quatro casos, descreveremos a tarefa #DW-15:

$M =$ {designwizard.design.Modifier.ISFROMSUPERCLASS,
designwizard.design.test.MethodNodeTest.DESIGNWIZARDJAR,
designwizard.design.Entity.addModifiers(Collection)}

$I - M =$ {designwizard.design.AbstractEntity.<init>(),
designwizard.design.test.MethodNodeTest.testGetPackageName(),
designwizard.design.AbstractEntity.addModifiers(Collection),
designwizard.design.Modifier.extractModifier(String)}

Das quatro entidades impactadas, duas não foram encontradas, sendo que a `AbstractEntity.addModifiers(Collection)` foi adicionada nesse *commit*. Das duas encontradas pela análise de impacto, apenas a `Modifier.extractModifier(String)` tinha histórico de acoplamento com pelo menos uma das mudanças, o que gerou 100% de probabilidade de impacto. A entidade `MethodNodeTest.testGetPackageName()` permaneceu com 0% de probabilidade. Por fim, três tarefas de modificação tiveram todas as suas entidades identificadas pela análise probabilística e com probabilidade de impacto de 100%.

Em resumo, a análise histórica ajudou a melhorar os resultados da análise estática para quase metade das tarefas de modificação. Consideramos esse resultado bastante expressivo, visto que a atividade de análise de impacto é, geralmente, um trabalho minucioso e cansativo para o desenvolvedor ou engenheiro de software.

Impala

Das doze tarefas de modificação do Impala, cinco não tiveram entidades identificadas pela análise estática, duas tiveram entidades identificadas pela análise estática, mas não pela análise histórica e cinco tiveram entidades identificadas pela análise estática e ponderadas pela análise histórica.

As cinco tarefas de modificação que não tiveram entidades identificadas pela análise estática são idênticas. Todas envolvem uma adição de um atributo à classe `TypeOfRefactoring` (conjunto M) e uma alteração no método `DefaultAnalysisStrategy.initCommands()` (conjunto $I - M$). Essa alteração é sempre a adição do uso do novo atributo no método. Como cada atributo ainda não existia antes da modificação, não havia relações de dependências estáticas ou históricas entre ele e o método alterado. Assim, mesmo sendo tarefas semelhantes, nada se pode afirmar sobre elas através da análise probabilística.

Para os dois casos em que as entidades foram encontradas pela análise estática, mas não pela análise histórica, apenas um deles pode ser analisado em nível de classe. A tarefa #I-T12 envolve mudanças e impactos dentro de uma mesma classe, o que, automaticamente, a faz ter 100% de probabilidade de impacto. Portanto, a tarefa #I-T7 será detalhada a seguir:

$M =$ {impactanalyzer.data.ImpactedTree.entity,
impactanalyzer.data.ImpactedTree.<init>(TypeOfRefactoring,String),
impactanalyzer.data.ImpactedTree.extractEntities(java.util.List),
impactanalyzer.data.ImpactedTree.getEntities(),
impactanalyzer.data.ImpactedTree.getEntity(),
impactanalyzer.data.ImpactedTree.entities}

$I - M =$ {impactanalyzer.data.ImpactedTree.equals(Object),
impactanalyzer.data.ImpactedTree.createToString(List,StringBuffer),
impactanalyzer.data.ImpactedTree.toString(),
impactanalyzer.data.ImpactedTree.hashCode(),
impactanalyzer.command.AddInheritanceCommand.execute(DesignIF,String[]),
impactanalyzer.command.AddEntityCommand.execute(DesignIF,String[])}

Todas as entidades impactadas foram localizadas pela análise estática, porém, com probabilidade de impacto igual a zero. Dessa forma, podemos observar se a análise histórica em nível de classe contribui para localizar as entidades impactadas. No entanto, das três classes impactadas (`ImpactedTree` é mudança), apenas uma foi retornada pela análise histórica: P(`AddInheritanceCommand`)=0,1. Em outras palavras, a análise histórica não contribui com nenhuma informação para essa tarefa, tanto em nível de entidade como de classe.

Por fim, das cinco tarefas que tiveram entidades identificadas pela análise probabilística, apenas uma tarefa teve seus impactos com probabilidades menores do que 50% e duas tiveram todas as entidades identificadas pela análise probabilística. A tarefa de modificação #I-T9, por exemplo, tinha 4 entidades no conjunto de mudanças e 8 no conjunto $I - M$. Todas as oito entidades foram identificadas com probabilidade de impacto de 100%.

A partir dos resultados coletados nos dois projetos, DesignWizard e Impala, observamos que a análise probabilística melhora os resultados obtidos pela análise estática e acrescenta conhecimento tanto em nível de entidade, quanto em nível de classe. Os valores encontrados das probabilidades nos dois projetos sugerem a seguinte classificação em relação a falso-positivos: impactos muito prováveis – $[50\%, 100\%]$; impactos pouco prováveis – $[0\%, 50\%)$. Essa classificação está baseada exclusivamente nos valores das probabilidades encontrados para as entidades impactadas dos dois projetos analisados. Para projetos com características diferentes, como tamanho do software e cultura da equipe, a classificação pode ser diferente. Portanto, é necessário realizar um estudo mais completo, que envolva projetos com diferentes características, para que uma classificação mais embasada possa ser proposta.

Além disso, sugerimos que a análise probabilística em nível de classe só deva ser empregada no caso em que a análise probabilística em nível de entidade não fornece muita informação a respeito das entidades possivelmente impactadas. Um cenário típico seria quando todas as entidades do conjunto $I - M$ estiverem com probabilidade de impacto igual a zero. Por fim, concluímos que a análise probabilística agrega informações relevantes para um desenvolvedor ou um engenheiro de software, visto que ela os ajuda a priorizar sua análise. Em outras palavras, ele pode analisar o impacto das entidades de acordo com a ordenação sugerida pela análise probabilística, enfatizando a atenção às primeiras entidades da lista. Dessa forma, respondemos à hipótese deste trabalho positivamente.

5.4 Considerações Finais

Neste capítulo, apresentamos a avaliação da técnica de análise probabilística. Primeiramente, avaliamos apenas a técnica de análise de impacto estática com o intuito de mensurar a quantidade de falso-positivos e falso-negativos gerados. Para isso, definimos duas métricas trazidas da recuperação de informação, precisão e revocação, com o objetivo de mensurar, com clareza, a acurácia dos algoritmos de análise de impacto. A definição dessas métricas foi necessária, pois abordagens anteriores de análise de impacto não diferenciavam, de forma concreta, falso-positivos e falso-negativos. Elas mediam a precisão dos seus algoritmos apenas em termos de falso-positivos, ignorando a geração de falso-negativos. Mais informações a respeito da comparação de nossa abordagem com outras se encontram no capítulo 6.

Antes de empregar as métricas propostas aos resultados obtidos através da aplicação do Impala nos três projetos avaliados, definimos a metodologia usada para criar os conjuntos de mudanças e de impactos reais em nível de classe. Dessa forma, permitimos que essa metodologia seja utilizada para a validação de outras técnicas de análise de impacto. Após coletar os valores de precisão e revocação para cada conjunto de mudanças, calculamos a média, mediana e o desvio padrão dessas medidas para cada configuração dos algoritmos. Assim, obtivemos o comportamento geral dos algoritmos em relação aos falso-positivos e falso-negativos: o aumento da profundidade de busca causa o aumento no número de falso-positivos e a redução no número de falso-negativos; inversamente, a redução da profundidade de busca diminui a quantidade de entidades identificadas que não são realmente impactadas e aumenta o número de entidades impactadas que não são identificadas pelos algoritmos. Como não pudemos afirmar qual das configurações geravam menos falso-positivos e falso-negativos, optamos por utilizar, na segunda etapa da avaliação, os algoritmos originais, que buscam exaustivamente por entidades impactadas.

Na segunda etapa, objetivamos a avaliação da técnica completa de análise probabilística, que compõe a análise estática e a análise histórica juntas. Para isso, escolhemos apenas dois dos três projetos usados na avaliação anterior e definimos uma nova metodologia pra criar as tarefas de mudanças e impactos reais em níveis de entidade e classe. A análise em nível de entidade é de granularidade mais fina, capaz de retornar informações mais detalhadas e valiosas para o usuário. Aplicamos, então, o Impala *Plug-in* aos projetos para coletar as informações históricas, o Impala para realizar a análise estática e, por fim, o ImpalaMiner para atribuir probabilidades de impacto às entidades identificadas pela análise estática através do uso das informações históricas.

As medidas precisão e revocação também são utilizadas nos resultados desse estudo de caso e evidenciam um comportamento semelhante ao encontrado na avaliação anterior: quando o número de falso-positivos decresce, o número de falso-negativos cresce. No entanto, a redução no número de falso-positivos é proporcionalmente maior do que o aumento no número de falso-negativos. Por fim, realizamos uma análise subjetiva dos resultados encontrados nas tarefas de modificação com intuito de investigar a utilidade da probabilidade calculada pela análise histórica. A partir dessa análise, observamos que a análise histórica é útil para mudanças do tipo alteração ou remoção, mas não é capaz de auxiliar quando as mu-

danças são do tipo adição. Propomos a seguinte classificação para os impactos de softwares semelhantes aos analisados: impactos muito prováveis aparecem com o valor de probabilidade dentro do intervalo $[50\%, 100\%]$; impactos poucos prováveis têm probabilidade dentro do intervalo $[0\%, 50\%)$.

Dessa forma, concluímos que a análise probabilística melhora os resultados do Impala e acrescenta informação útil ao desenvolvedor ou engenheiro de software responsável por realizar a atividade de análise de impacto, bem como fornece resultados mais precisos para uma análise de estimativa de custos. Assim, afirmamos que é possível aumentar a precisão da análise de impacto realizada antes da implementação de uma mudança, através do uso de informações históricas do comportamento das mudanças do sistema.

Capítulo 6

Trabalhos Relacionados

Com a crescente demanda por constantes mudanças no software, tanto durante seu desenvolvimento, quanto na fase de manutenção, a análise de impacto de mudanças vem sendo empregada com uma freqüência crescente nos ambientes de desenvolvimento de software. Neste capítulo, discorremos sobre trabalhos semelhantes que propõem técnicas de análise de impacto baseadas no código fonte do programa. Comparamos as métricas utilizadas para analisar a precisão de outras técnicas de análise de impacto com as métricas propostas neste trabalho. Por fim, retomamos a discussão sobre os três trabalhos de evolução de software baseada no histórico de mudanças da seção 2.5.1 para compará-las a nossa solução.

6.1 Análise de Impacto

As primeiras técnicas de análise de impacto criadas eram para programas estruturados. Dessas, uma grande parcela das técnicas de análise de impacto é baseada na análise do código fonte. Dentre as técnicas básicas, a análise de fluxo de dados [Keables, Roberson e Mayrhauser 1996; Harrold e Rothermel 1994], a análise de fluxo de controle [Loyall e Mathisen 1996] e o fatiamento de programas [Horwitz, Reps e Binkley 1996; Korel e Laski 1990] compõem a base da análise de dependência.

A maioria das pesquisas em análise de impacto em software orientado a objetos é, também, baseada no código do software. Dentre elas, destacam-se algumas linhas de pesquisa: a adaptação da análise de programas estruturados [Bohner 2002], o foco em testes de regressão [Ryder e Tip 2001; Ren et al. 2004], a análise de impacto baseada

em modelos UML [Briand, Labiche e O'Sullivan 2003; Briand, Labiche e O'Sullivan 2003], a análise de impacto dinâmica [Law e Rothermel 2003; Law e Rothermel 2003], e a análise algorítmica do código fonte [Li e Offutt 1996; Li e Offutt 1996; Lee 1999; Lee, Offutt e Alexander 2000].

Bohner propõe a adaptação da análise de programas estruturados, já existente e bastante explorada, para ser usada também em sistemas componentizados (*Commercial-Off-The-Shelf – COTS*) [Bohner 2002]. À medida que os sistemas de software tornam-se mais complexos, incorporando aspectos de distribuição e componentização, por exemplo, o volume de informação de cada projeto de software torna-se muito grande e incompreensível a um engenheiro de software. Para isso, Bohner defende a adaptação de técnicas de análise de dependência e análise de rastreabilidade, já exploradas por ele no passado, para controlar as informações do projeto e guiar a análise de impacto. Essas duas análises são chamadas de análise estrutural do sistema, que devem ser associadas a uma análise semântica do sistema, para que informações embutidas nos relacionamentos de dependência do sistema e nos artefatos sejam consideradas na análise de impacto [Bohner 2002].

Ryder e Tip usam a análise de impacto para identificar quais mudanças afetaram determinado conjunto de testes de regressão, que mudaram seu comportamento após a introdução de um conjunto de mudanças [Ryder e Tip 2001]. Quando se usa o conceito de testes de regressão, ao se incorporar uma mudança, é importante re-executar o conjunto de testes do software, para verificar se suas funcionalidades foram alteradas ou se erros foram introduzidos. A técnica proposta consiste em transformar o código fonte das mudanças em um conjunto de mudanças atômicas, que é mapeado para um grafo para posterior verificação de quais mudanças afetaram o comportamento de determinado conjunto de testes.

Posteriormente, Ryder et al constroem uma ferramenta, chamada **Chianti**, para validar a técnica de análise de impacto com foco em testes de regressão [Ren et al. 2004]. Neste trabalho, além da análise de impacto estática, também é empregada a análise de impacto dinâmica. O Chianti recebe como entrada o programa original, o programa modificado e o conjunto de testes de unidade, e fornece os testes afetados e o conjunto de mudanças que afetaram os testes como saída. Além do conjunto de mudanças atômicas, o Chianti usa grafos de chamadas, que podem ser obtidos através de análise estática, ou através da execução dos testes de regressão (análise dinâmica), para realizar a análise de impacto. O Chianti foi

submetido a um estudo de caso, no qual se realizou uma análise *post mortem* do sistema Daikon, escrito em Java. Destaca-se o comparativo dos resultados da análise de impacto usando grafos de chamadas obtidos através da análise estática e da execução de testes de regressão (análise dinâmica), dos quais o último obteve resultados mais precisos.

O Chianti e o Impala assemelham-se por aplicarem análise de impacto através do uso de grafos de chamadas. Contudo, o Chianti só pode ser aplicado após a realização da mudança, enquanto o Impala permite que a análise de impacto seja realizada antes da implementação da mudança, podendo ser utilizada de forma preventiva.

Outra abordagem de análise de impacto, mas que é realizada antes da implementação da mudança, é através do uso de requisitos ou da especificação do sistema, ao invés do código fonte. É o que fazem Briand et al, utilizando a especificação UML do projeto [Briand, Labiche e O'Sullivan 2003]. Esse trabalho tem como objetivos: a classificação automática de mudança entre versões diferentes do modelo UML; a verificação da consistência dos diagramas; a realização da análise de impacto; e a priorização dos resultados obtidos de acordo com a probabilidade dos elementos previstos de sofrerem impacto [Briand, Labiche e O'Sullivan 2003].

Diferente de todos os outros trabalhos vistos, este não utiliza técnicas conhecidas de análise de impacto, como fatiamento de programa, cálculo do fecho transitivo de grafos de chamadas. A técnica desenvolvida é baseada no uso de conjuntos de regras escritas em OCL que, primeiramente, verificam a consistência das especificações (atual e da mudança), em seguida identificam o tipo de mudança para, por fim, realizar a análise de impacto [Briand, Labiche e O'Sullivan 2003]. Foram escritas 120 regras de consistência, uma taxonomia de mudanças com 97 regras e mais 97 regras de análise de impacto. Resultados obtidos, através de um estudo de caso, mostraram que o aumento de elementos impactados cresce linearmente, quando se considera impactos indiretos. Este é um bom indicativo da precisão da técnica, pois a redução de falso-positivos é um dos grandes desafios na análise de impacto. No entanto, ao se considerar apenas a especificação, muitas informações importantes contidas no código, nos testes e em outros artefatos, são ignoradas. Além disso, para aplicar essa técnica, é obrigatório ter uma especificação UML com regras em OCL e manter a conformidade entre a especificação e o código, diferente de nossa técnica, que utiliza apenas o código do software.

Lee e Offutt desenvolveram uma técnica algorítmica de análise de impacto de mudanças para sistemas orientados a objetos [Lee 1999]. Esta técnica recebe como entrada um conjunto de mudanças e a representação do sistema em forma de grafos, adaptados às peculiaridades da orientação a objetos. Os algoritmos são aplicados através da utilização do cálculo de fecho transitivo de grafos de chamadas, análise de fluxo de controle e de fluxo de dados [Li e Offutt 1996; Li e Offutt 1996]. Além da técnica, esse trabalho criou métricas de análise de impacto, classificou os diferentes tipos de relacionamentos de dependência existentes em softwares orientados a objetos e catalogou os diferentes tipos de mudanças que podem ser aplicadas a sistemas orientados a objetos [Lee 1999]. Por fim, uma importante contribuição desse grupo foi a criação dos novos conceitos de grafos de dependência de dados para sistemas orientados a objetos: grafo de dependência de dados intra-método orientado a objetos, grafo de dependência de dados inter-método orientado a objetos, e grafo de dependência do sistema orientado a objetos [Lee, Offutt e Alexander 2000].

Os trabalhos de Lee e Offutt de classificação dos diferentes tipos de dependência e dos tipos de mudanças de software orientados a objetos auxiliaram no desenvolvimento deste trabalho. Por exemplo, a classificação das dependências da seção 2.4.1 foi baseada no estudo apresentado por Lee [Lee 1999]. Contudo, existem diferenças notáveis entre os algoritmos desenvolvidos por eles e os algoritmos do Impala. Os primeiros são de granularidade fina, pois analisam o código em nível de sentença, enquanto os algoritmos do Impala são de granularidade grossa por analisarem o software em nível de entidade. Ambos utilizam grafos de chamadas, porém, o primeiro constrói estruturas de grafos que contém informações adicionais em suas arestas, enquanto o Impala manipula uma representação do software em forma de conjuntos de entidades e relacionamentos.

Law e Rottermel propuseram uma técnica de análise de impacto dinâmica, que reúne a análise baseada em grafos de chamadas, o fatiamento estático e o fatiamento dinâmico do programa [Law e Rothermel 2003; Law e Rothermel 2003]. Nessa mesma linha, outros trabalhos propõem alcançar alta precisão nos resultados da análise de impacto aplicada durante a execução do programa. [Breech, Tegtmeyer e Pollock 2005; Apiwattanapong, Orso e Harrold 2005]. A possibilidade de guiar a análise pelas execuções do programa tem a vantagem de restringi-la e, assim, obter resultados mais precisos do que ao realizar somente a análise estática, pois elimina, por exemplo, todas as possibilidades de instâncias de uma interface,

contabilizando apenas a instância da execução em questão. No entanto, perde-se a completude da análise estática, já que não se pode garantir a execução de todas as possibilidades de comportamento do sistema. Uma desvantagem dessas abordagens para o Impala é que todas elas só podem ser executadas após a mudança do sistema.

Em linhas gerais, as técnicas de análise de impacto de mudanças diferem através das seguintes características: abordagem – estática, dinâmica, online; granularidade da busca; forma de representação do sistema; uso – antes ou depois da mudança; e técnica de análise de dependência – fatiamento de programa, fecho transitivo em grafos de chamadas, fluxo de controle e de dados.

Dentre as variadas técnicas não existe uma que se destaque como melhor opção, mas técnicas que se adéquam melhor a cada situação. Por exemplo, das técnicas empregadas após a mudança, a dinâmica destaca-se pela precisão de seus resultados, podendo ser empregada para garantir a corretude do software. Já as técnicas de análise estática que podem ser utilizadas antes da implementação da mudança, auxiliam no cálculo das estimativas de custo e tempo, podem guiar o desenvolvedor na atividade de modificação e oferecer informações importantes ao engenheiro de software que pretende re-estruturar o código ou, simplesmente, propor uma mudança.

Não realizamos uma análise comparativa de nossa técnica com as demais, devido à falta de disponibilização de ferramentas ou de detalhes dos algoritmos suficientes para que pudéssemos implementarmos. Os algoritmos que poderíamos implementar são complexos e demandariam tempo maior do que o disponível para a realização deste trabalho. Portanto, sugerimos que futuramente seja realizada uma análise comparativa em termos de resultados obtidos e desempenho. Para comparar os resultados das diferentes técnicas, propomos o uso das métricas precisão e revocação definidas neste trabalho. A seguir, discorremos sobre como era mensurada a acurácia dos algoritmos de análise de impacto e argumentamos o porquê de nossas métricas serem mais adequadas.

6.2 Métricas Para Avaliação da Acurácia das Técnicas de Análise de Impacto

Uma técnica de análise de impacto é susceptível a dois tipos de erros: identificar impactos que não ocorrem de fato – falso-positivos; e deixar de identificar impactos que ocorrem – falso-negativos. Não encontramos, na literatura, trabalhos que distinguissem os dois tipos de erros, muito menos que se propusessem a avaliar suas técnicas baseadas neles. Os trabalhos que se propõem a avaliar acurácia calculam, apenas, o número de falso-positivos produzido.

Law et al [Orso et al. 2004] avaliam a acurácia dos dois algoritmos de análise de impacto dinâmica (*CoverageImpact* e *PathImpact*) que propõem através do cálculo dos tamanhos relativos dos conjuntos de impactos computados e dos conjunto de mudanças. A acurácia é medida em termos do número de métodos dentro do conjunto de impactos comparado com o número total de métodos da execução. Eles consideram que o algoritmo A atingiu maior acurácia do que o algoritmo B se os métodos de A constituírem um subconjunto dos métodos de B. Uma informação importante é que eles consideram que seus algoritmos são seguros no sentido de que nenhum método que não está no conjunto de impactos pode ser afetado por uma mudança. Em outras palavras, consideram que seus algoritmos não produzem falso-negativos para as execuções analisadas. Porém, não se pode assegurar de que todas as possíveis execuções são consideradas. Sendo assim, esses algoritmos produzem falso-negativos que não são contabilizados no cálculo da acurácia.

A mesma abordagem é utilizada por Apiwattanapong et al [Apiwattanapong, Orso e Harrold 2005] para comparar seu algoritmo dinâmico (*CollectEA*) com os anteriores de Law et al. Por fim, ela é repetida por Breech et al [Breech, Tegtmeyer e Pollock 2006] para comparar um algoritmo de análise estática com um de análise dinâmica. Um problema adicional deste último é que não se pode assumir que uma técnica de análise estática é segura, porque alguns impactos, como os originados por *very late binding* só podem ser detectados em tempo de execução. Ou seja, para garantir que a análise estática não gera falso-negativos, o código analisado não pode conter uso de reflexão, bem como qualquer *framework*, visto que, atualmente, a grande maioria dos *frameworks* fazem uso de *very late binding*[1].

[1]Com o *very late binding*, uma aplicação pode encontrar um objeto, e carregá-lo e chamar a implementação de sua interface dinamicamente em tempo de execução. É diferente do *late binding*, que permite que a aplicação

Após uma verificação sobre como a acurácia dos resultados obtidos pelas técnicas de análise de impacto era medida, notamos a necessidade de mensurar o número de falso-negativos produzidos, que era ignorado pelas outras abordagens. Por isso, definimos as métricas precisão e revocação que relacionam o conjunto de entidades possivelmente impactadas com o conjunto de entidades impactadas de fato e informam o percentual de falso-positivos e falso-negativos produzidos, respectivamente.

6.3 Mineração de repositórios de software

Esta seção tem o objetivo de retomar a análise comparativa entre as abordagens de mineração de repositórios vista na seção 2.5.1 para compará-las a nossa solução. Para isso, a tabela 6.1 resume uma nova comparação entre as características das abordagens. Com relação à linguagem de programação, nossa solução é dependente da linguagem Java, assim como a terceira abordagem. Em relação à granularidade das informações obtidas, apenas a primeira abordagem difere das demais, pois obtém informações em nível de arquivos (classes). Nossa solução é capaz de realizar a mineração tanto em nível de entidade, como em nível de classe. Essa é uma vantagem em relação as outras abordagens, porque quando não é possível obter informações históricas em nível de entidade, podemos buscar informações históricas em um nível menos detalhado, o de classe.

Tabela 6.1: Comparativo das abordagens de mineração de repositórios

Abordagens	**Ling. de programação**	**Granularidade**	**Integrada a uma IDE**	**Algoritmo**
Predição de mudanças no código através da mineração do histórico de revisões	Independente	Arquivos (classes)	Não	FP-Tree
Minerando histórico de versões para guiar mudanças no software	Independente	Entidades	Eclipse	Apriori
Análise de acoplamento de mudanças com granularidade fina	Java	Entidades	Eclipse	Não informado (*clustering*)
Análise de impacto probabilística	Java	Entidades	Eclipse	ImpalaMiner

Em relação à integração com uma IDE, apenas a primeira abordagem difere das demais

carregue o objeto, o instancie e o utilize em tempo de execução.

por não ter integração. Todas as demais dependem do Eclipse para serem executadas. No caso da nossa solução e da abordagem de Gall et al, essa integração tem benefícios claros, como o uso das bibliotecas do Eclipse para a realização da comparação estrutural entre das classes. Com ela, podemos filtrar as mudanças registradas no repositório para considerar apenas aquelas ligadas, efetivamente, a mudanças na estrutura do software. Diferente da solução de Gall et al [Fluri, Gall e Pinzger 2005], o Impala *Plug-in* consegue eliminar mudanças relacionadas à adição de linhas em branco dentro de um método e modificação no comentário (Javadoc). Essas duas eliminações correspondem a uma melhora significativa, principalmente quando o Javadoc e padrões de formatação são utilizados pela equipe de implementação do software em análise.

Por fim, as quatro abordagens utilizam diferentes algoritmos, que estão estritamente relacionados ao uso final da solução. A abordagem de Ying et al utilizam o algoritmo FP-Tree para gerar grupos de classes que mudam juntas com freqüência para predizer mudanças. Esses grupos são formados por relações conjuntivas entre as classes. Por exemplo, se as classes A, B e C fazem parte de um conjunto de mudanças freqüente, é porque todas as três classes foram validadas juntas com certa freqüência no passado. Zimmermann et al utilizam o algoritmo Apriori, semelhante ao FP-Tree, que encontra regras de associação com a seguinte forma: A, B (4) $\Longrightarrow$ C (3). Essa regra significa que as classes A e B foram validadas juntas quatro vezes e dessas, 3 vezes C também foi validada. As regras de associação encontradas pelo eROSE são utilizadas para sugerir mudanças, que são ordenadas de acordo com os valores do suporte e da confiança da regra. A abordagem de Gall et al tem como objetivo investigar a evolução do software num âmbito geral, analisar quais grupos de classes ou módulos estão mais acoplados, refatorar o código com base nas informações obtidas, etc. Assim, seu trabalho não tem foco no algoritmo de mineração utilizado.

Para nossa solução, desenvolvemos um algoritmo de mineração específico para o problema, baseado no teorema de Bayes, apresentado na seção 4.3. Ele relaciona um conjunto de entidades a serem mudadas com cada entidade potencialmente impactada. Essa relação é feita com a disjunção das entidades no conjunto de mudança. Para o exemplo anterior, a interpretação é: se A ou B mudam, então C tem probabilidade de x% de ser impactada. A vantagem do nosso algoritmo em relação aos demais é que ele combina todas as possíveis regras de associação ou todos os conjuntos freqüentes em um só cálculo, o que as outras

abordagens são incapazes de fazer. Contudo, é importante salientar que essa abordagem é mais adequada ao problema da análise de impacto, no qual queremos ponderar a probabilidade de impacto de uma entidade em relação a um conjunto de mudanças. Portanto, ela não seria, necessariamente, mais adequada aos objetivos dos outros três trabalhos.

6.4 Considerações Finais

Neste capítulo, apresentamos uma série de trabalhos relacionados à análise de impacto de mudanças probabilística em três aspectos. Primeiramente, discorremos sobre as técnicas de análise de impacto de mudanças, principalmente as abordagens que utilizam o código fonte do software e as direcionadas para sistemas orientados a objetos. Em seguida, comparamos as métricas de a avaliação da acurácia dos algoritmos de análise de impacto propostas nesse trabalho, precisão e revocação, com outras medições utilizadas até então. Por fim, realizamos uma análise comparativa entre nossa abordagem para extrair informações históricas de mudanças e três trabalhos semelhantes que exploram dados armazenados em sistemas de controle de versão. O diferencial de nossa abordagem em relação a outras de análise de impacto é que ela coleta informações históricas do software em análise para agregar conhecimento à técnica de análise de impacto estática utilizada.

Capítulo 7

Considerações Finais

Neste trabalho de dissertação, investigamos a hipótese de aumentar a precisão da análise de impacto de mudanças que é realizada antes da implementação da mudança, através do uso de informações do histórico de mudanças do software em análise. Existem duas vertentes que analisam o impacto de mudanças através do uso do código do sistema: a análise estática, que extrai informações do código fonte e, geralmente, é utilizada antes da implementação da mudança; e a dinâmica, que coleta informações das execuções do software, após a implementação da mudança, para analisar os impactos causados por ela. Este trabalho enfocou a análise de impacto que pode ser empregada antes da realização da mudança.

As técnicas de análise de impacto estáticas atuais [Arnold e Bohner 1996; Turver e Malcolm 1994; Bohner 2002; Ren et al. 2004; Lee 1999] são baseadas, exclusivamente, na análise de dependência do código, o que inclui fatiamento de programa, fechamento transitivo em grafos de chamadas, análise de fluxo de controle e de dados. Portanto, elas se limitam a analisar o código do software em seu estado atual, o que acaba causando a geração de muitos falso-positivos.

Com o intuito de reduzir o número de falso-positivos, propomos uma nova técnica de análise de impacto, que, além de aplicar a técnica de fechamento transitivo em grafos de chamadas, pondera os impactos encontrados de acordo com o histórico de mudanças do próprio software analisado. Assim, agregamos conhecimento sobre as dependências históricas entre as entidades do software para utilizá-lo na análise de impacto. Por exemplo, seja um conjunto de mudanças composto pela entidade A. As entidades B e C dependências indiretas de A e estão no conjunto de impactos. Informações históricas revelam que A e B foram

alteradas juntas muitas vezes no passado, o que não ocorre com A e C. Então, a entidade B tem maior probabilidade de ser impactada por uma mudança em A do que C.

A técnica de análise de impacto probabilística que propomos é composta pela técnica de análise de impacto estática para sistemas orientados a objetos e pela análise histórica do software. Primeiramente, aplicamos ao software a técnica de análise estática, que extrai a representação do mesmo para buscar as entidades dependentes das mudanças propostas, o conjunto de impactos. Após a obtenção dos impactos, aplicamos a análise histórica para buscar, no sistema de controle de versão que armazena o software, informações sobre quais entidades mudaram juntas com freqüência no passado. Por fim, atribuímos probabilidades de as entidades do conjunto de impactos serem impactadas pelo conjunto de mudanças a partir das informações históricas coletadas no sistema de controle de versão.

Para avaliar a técnica de análise de impacto proposta, definimos duas métricas que relacionam o conjunto de possíveis impactos com os impactos reais. A precisão informa a percentagem de falso-positivos gerados, que são os possíveis impactos identificados pela técnica, mas que não são alterados de fato. A revocação informa a percentagem de falso-negativos gerados, que são os impactos reais não identificados pela técnica. Primeiramente, aplicamos essas duas métricas à técnica de análise de impacto estática para analisarmos o comportamento (aumento ou redução) dos falso-positivos e falso-negativos em relação à profundidade de busca dos algoritmos do Impala. Desse estudo, concluímos que o aumento de falso-positivos está relacionado com a redução de falso-negativos e vice-versa. Em seguida, aplicamos as métricas à técnica de análise probabilística com o intuito de avaliar se ela melhora os resultados obtidos pela aplicação à técnica de análise estática. Nossa hipótese se confirma, visto que, nos sistemas avaliados, a análise probabilística obteve um ganho na precisão entre 30 e 55 pontos percentuais em relação à análise estática. Como a perda na revocação variou entre 20 e 30 pontos percentuais, o ganho na precisão foi maior do que a perda na revocação. Por fim, avaliamos os resultados subjetivamente para concluir que a análise histórica agrega conhecimento útil para o engenheiro de software aos resultados da análise estática.

7.1 Contribuições

Com a conclusão deste trabalho, podemos destacar as contribuições oriundas do mesmo. Primeiramente contribuímos para o aumento da precisão da análise de impacto realizada antes da implementação da mudança. Com a combinação da técnica de análise de impacto estática com a análise histórica do software, obtivemos melhor precisão do que utilizando apenas a análise estática. Apesar de a técnica probabilística ter gerado mais falso-negativos do que a técnica de análise estática pura, quando as entidades com probabilidades de impacto iguais a zero são eliminadas, o ganho da precisão foi maior do que a perda na revocação.

Mais especificamente, contribuímos com a proposta de uma nova técnica de análise de impacto estática, que extrai a representação do software em termos de entidades e relacionamentos, busca por impactos relacionados às chamadas de métodos e aos usos de atributos e considera questões específicas da orientação a objetos, como a herança e o polimorfismo. Atualmente existem poucas técnicas de análise de impacto estáticas específicas para sistemas orientados a objetos [Lee 1999; Bohner 2002; Ren et al. 2004].

A técnica de análise de impacto estática desenvolvida neste trabalho possui algumas limitações, como não resolver o problema do *very late binding* e buscar, exaustivamente por impactos indiretos sem se basear em algum critério. Porém, os resultados obtidos na avaliação da técnica mostraram-se promissores, visto que a soma dos erros da precisão e da revocação variou entre 10 e 30 pontos percentuais.

Além disso, contribuímos para a pesquisa na área de evolução de software que investiga informações contidas nos sistemas de controle de versão para compreender melhor o comportamento evolutivo do software. Nesse aspecto, construímos uma ferramenta que extrai informações relevantes do CVS e as armazena no banco de dados. Posteriormente, criamos um algoritmo de mineração de dados, baseado no teorema de Bayes, que utiliza essas informações de mudanças para auxiliar futuras modificações no software. Esse algoritmo foi especificamente criado para atender ao problema da análise de impacto. Ele recebe como entrada o conjunto de mudanças, o conjunto de impactos e o histórico do software e fornece a probabilidade de impacto de cada entidade pertencente ao conjunto de impactos. Apesar de seu aspecto específico, ele pode ser generalizado para uso em outros problemas que envolvam a mineração de dados.

Uma outra contribuição para a área da análise de impacto foi a definição das métricas precisão e revocação para medir a acurácia das técnicas de análise de impacto, publicada através do artigo intitulado "On the precision and accuracy of impact analysis techniques" [Hattori et al. 2008]. Cada métrica mensura um dos tipos de erros que uma técnica de análise de impacto pode produzir em seus resultados: falso-positivos e falso-negativos. Com a definição dessas duas métricas e a descrição das metodologias utilizadas para avaliar nossas técnicas, esperamos que próximos trabalhos as empreguem com intuito de realizar uma análise comparativa com os resultados que obtivemos neste trabalho.

Por fim, acreditamos que a técnica de análise de impacto probabilística pode ser empregada com sucesso antes da implementação de uma mudança para facilitar uma série de atividades presentes no ciclo de desenvolvimento e na manutenção de um sistema. Ela pode evitar que uma mudança aparentemente inocente se transforme é uma mudança de grandes proporções, o que chamamos de efeito gelatina. Além disso, pode auxiliar o projetista a escolher entre duas ou mais alternativas de mudanças, apontando qual causa menos impacto. Portanto, seu uso para calcular estimativas de custos de uma modificação é importante, com a ressalva de que os cálculos devem considerar o conhecimento do especialista. A análise de impacto probabilística, além de obter resultados mais precisos, ainda agrega conhecimento sobre a ponderação dos impactos, que pode auxiliar um especialista em uma tomada de decisão.

7.2 Trabalhos Futuros

Com a conclusão deste trabalho, surgem muitas oportunidades para complementá-lo. Algumas melhorias na solução e propostas de novos trabalhos que surgiram durante o desenvolvimento desta são destacadas a seguir.

Criar heurísticas para tratar a busca das dependências indiretas no Impala. Os algoritmos do Impala buscam, exaustivamente pelas entidades que possuem alguma dependência com a entidade que será modificada. Uma entidade pode depender diretamente de outra entidade, ou pode ter dependência indireta. Intuitivamente, as dependências diretas são mais afetadas por uma mudança do que as indiretas e, a depender da mudança, é pouco provável que uma dependência indireta seja impactada.

Por exemplo, é muito provável que a mudança no nome do atributo privado `idade` só cause impacto em seus métodos de acesso: `setIdade()` e `getIdade()`. Dessa forma, a busca por entidades indiretas poderia ser aperfeiçoada através da adição de heurísticas relacionadas ao tipo de mudança.

Complementar a nossa solução com uma solução de análise de impacto dinâmica. Até o momento, todas as abordagens de análise de impacto dinâmica são aplicadas após a realização de uma mudança. Uma forma de melhorar a precisão dos resultados a análise de impacto probabilística é combiná-la com uma nova abordagem de análise de impacto dinâmica que possa ser empregada antes da implementação da mudança. A idéia é utilizar os testes executáveis (por exemplo, teste de unidade) para executar os trechos do código que serão mudados e adicionar ao conjunto de impactos as entidades contidas nos rastros de execução. Apesar de essa abordagem não garantir que todas as possíveis execuções serão testadas, podemos combinar o conjunto de impactos obtido com o conjunto de impactos da técnica de análise probabilística para eliminar mais falso-positivos e melhorar a precisão.

Realizar testes de usabilidade em campo. Este trabalho faz parte do projeto DesignChecker, financiado pela FINEP e pela empresa CPMBraxis. Ele está sendo utilizado na fábrica de software da CPMBraxis para atender parte dos requisitos necessários ao CMMI 5 [SEI 2008]. Portanto, este trabalho continuará em desenvolvimento e pretendemos realizar testes de usabilidade com especialistas que aplicam a análise de impacto nos softwares desenvolvidos na fábrica. Com esses testes, investigaremos na prática e com projetos comerciais, se a ponderação das possíveis entidades impactadas auxilia o especialista. Investigaremos, também, se a solução facilita e agiliza o trabalho dos especialistas, que, antes da ferramenta, era considerado enfadonho pelos mesmos.

Tratar mudanças estruturais de nome de classe, método ou atributo no Impala *Plug-in*. Uma limitação da extração das mudanças estruturais do Impala *Plug-in* é que o comparador de mudanças estrutural não é capaz de identificar mudanças no nome de um método ou um atributo. Ele considera que o método com o nome antigo foi removido e um novo método com o novo

nome foi adicionado. A identificação da mudança de nome não é trivial e há alguns estudos que propõem implementá-la [Xing e Stroulia 2005; Tu e Godfrey 2002]. Integrar uma dessas abordagens ao Impala *Plug-in* seria uma melhoria significativa na extração do histórico de mudanças.

Aplicar análise histórica para descobrir dependências não estáticas. A abordagem atual limita-se a considerar as dependências diretas e indiretas identificadas pela análise estática. A partir dessas dependências estáticas, o ImpalaMiner procura, no histórico de mudanças, evidências de que as entidades dependentes estaticamente também possuem dependências históricas. Podemos estender o Impala *Plug-in* e o ImpalaMiner para serem utilizados, de forma independente em relação à análise estática, para encontrarem dependências históricas não relacionados a dependências estáticas. Essa abordagem assemelha-se à proposta por Ying et al [Ying 2003], mas com a diferença de poder ser realizada em nível de entidade. Assim, impactos não encontrados pela análise atual, como os oriundos do uso de reflexão em Java (*very late binding*) ou de *frameworks*, seriam encontrados com a nova abordagem.

Generalizar o algoritmo ImpalaMiner. Atualmente, estamos investigando a possibilidade de modificar o algoritmo ImpalaMiner com o intuito de criar um novo algoritmo que gere regras de associação generalizadas. Essas regras são generalizadas por conter, além de conjunções entre seus elementos, disjunções. Dessa forma, esse algoritmo geral pode ser empregado em outros problemas que envolvem a mineração de dados.

Bibliografia

[Agrawal e Srikant 1998]AGRAWAL, R.; SRIKANT, R. Fast algorithms for mining association rules. In: *Readings in database systems*. 3. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, (Morgan Kaufmann Series In Data Management Systems). p. 580–592. ISBN 1-55860-523-1.

[Ajila 1995]AJILA, S. Software maintenance: an approach to impact analysis of objects change. *Software Practice Experience*, John Wiley & Sons, Inc., New York, NY, USA, v. 25, n. 10, p. 1155–1181, 1995. ISSN 0038-0644.

[Apiwattanapong 2007]APIWATTANAPONG, T. *Identifying Testing Requirements for Modified Software*. Tese (Doutorado) — Georgia Institute of Technology, Atlanta, Georgia, 2007.

[Apiwattanapong, Orso e Harrold 2005]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 (ICSE '05)*. New York, NY, USA: ACM, 2005. p. 432–441. ISBN 1-59593-963-2.

[Arnold e Bohner 1993]ARNOLD, R.; BOHNER, S. Impact analysis - towards a framework for comparison. In: *Proceedings of the Conference on Software Maintenance (ICSM '93)*. Washington, DC, USA: IEEE Computer Society, 1993. p. 292–301. ISBN 0-8186-4600-4.

[Arnold e Bohner 1996]ARNOLD, R. S.; BOHNER, S. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996. ISBN 0818673842.

[Bayes' theorem 2008]BAYES' theorem. 2008. URL: `http://en.wikipedia.org/wiki/Bayes'_theorem` (Acesso em Fev./2008).

[Bohner 2002]BOHNER, S. A. Extending software change impact analysis into COTS components. In: *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW '02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 175. ISBN 0-7695-1855-9.

[Bohner 2002]BOHNER, S. A. Software change impacts - an evolving perspective. In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 263–2. ISBN 0-7695-1819-2.

[Breech, Tegtmeyer e Pollock 2005]BREECH, B.; TEGTMEYER, M.; POLLOCK, L. A comparison of online and dynamic impact analysis algorithms. In: *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR '05)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 143–152. ISBN 0-7695-2304-8.

[Breech, Tegtmeyer e Pollock 2006]BREECH, B.; TEGTMEYER, M.; POLLOCK, L. Integrating influence mechanisms into impact analysis for increased precision. In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*. Washington, DC, USA: IEEE Computer Society, 2006. p. 55–65. ISBN 0-7695-2354-4.

[Briand, Labiche e O'Sullivan 2003]BRIAND, L. C.; LABICHE, Y.; O'SULLIVAN, L. Impact analysis and change management of UML models. In: *Proceedings of the International Conference on Software Maintenance (ICSM '03)*. Washington, DC, USA: IEEE Computer Society, 2003. p. 256–265. ISBN 0-7695-1905-9.

[Briand, Labiche e O'Sullivan 2003]BRIAND, L. C.; LABICHE, Y.; O'SULLIVAN, L. *Impact Analysis and Change Management of UML Models*. [S.l.], Fev. 2003. Disponível em: <http://squall.sce.carleton.ca/pubs_tech_rep.html#2003 (Acesso em em Dez./2006)>.

[Brunet e Guerrero 2007]BRUNET, J.; GUERRERO, D. *Design Wizard*. 2007. http://www.designwizard.org (Acesso em Jan/2008).

[Cirne et al. 2006]CIRNE, W. et al. Labs of the world, unite!!! *J. Grid Comput.*, v. 4, n. 3, p. 225–246, 2006.

[Fluri e Gall 2006]FLURI, B.; GALL, H. C. Classifying change types for qualifying change couplings. In: *Proceedings of the 14th International Conference on Program Comprehension (ICPC '06)*. Athen, Greece: IEEE Computer Society Press, 2006. p. 35–45.

[Fluri, Gall e Pinzger 2005]FLURI, B.; GALL, H. C.; PINZGER, M. Fine-grained analysis of change couplings. In: *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 66–74. ISBN 0-7695-2292-0.

[Forman, Forman e Forman 2004]FORMAN, I. R.; FORMAN, N.; FORMAN, I. R. *Java Reflection in Action*. Greenwich, CT, USA: Manning Publications Co., 2004. (In Action series). ISBN 1932394184.

[Foundation 2006]FOUNDATION, F. S. *CVS - Concurrent Versions System*. 2006. `http://savannah.nongnu.org/projects/cvs/` (Acesso em Dez./2006).

[Gall, Jazayeri e Krajewski 2003]GALL, H.; JAZAYERI, M.; KRAJEWSKI, J. CVS release history data for detecting logical couplings. In: *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003. p. 13. ISBN 0-7695-1903-2.

[Han, Pei e Yin 2000]HAN, J.; PEI, J.; YIN, Y. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 29, n. 2, p. 1–12, 2000. ISSN 0163-5808.

[Harrold e Rothermel 1994]HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 1994. p. 154–163. ISBN 0-89791-691-3.

[Hattori et al. 2008]HATTORI, L. et al. On the precision and accuracy of impact analysis techniques. In: *Proceedings of the 7th IEEE International Conference on Computer and Information Science*. Washington, DC, USA: IEEE Computer Society, 2008. Aceito para publicação.

[Horwitz, Reps e Binkley 1996]HORWITZ, S.; REPS, T.; BINKLEY, D. Interprocedural slicing using dependence graphs. In: BOHNER, S. (Ed.). *Software Change Impact Analysis*. [S.l.]: IEEE Computer Society Press, 1996. p. 137–171.

[Keables, Roberson e Mayrhauser 1996]KEABLES, J.; ROBERSON, K.; MAYRHAUSER, A. von. Data flow analysis and its applications to software maintenance. In: BOHNER, S. (Ed.). *Software Change Impact Analysis*. [S.l.]: IEEE Computer Society Press, 1996. p. 184–196.

[Korel e Laski 1990]KOREL, B.; LASKI, J. Dynamic slicing of computer programs. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 13, n. 3, p. 187–195, 1990. ISSN 0164-1212.

[Law e Rothermel 2003]LAW, J.; ROTHERMEL, G. Incremental dynamic impact analysis for evolving software systems. In: *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*. Washington, DC, USA: IEEE Computer Society, 2003. p. 430. ISBN 0-7695-2007-3.

[Law e Rothermel 2003]LAW, J.; ROTHERMEL, G. Whole program path-based dynamic impact analysis. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003. p. 308–318. ISBN 0-7695-1877-X.

[Lee, Offutt e Alexander 2000]LEE, M.; OFFUTT, A. J.; ALEXANDER, R. T. Algorithmic analysis of the impacts of changes to object-oriented software. In: *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*. Washington, DC, USA: IEEE Computer Society, 2000. p. 61. ISBN 0-7695-0774-3.

[Lee 1999]LEE, M. L. *Change Impact Analysis of Object-Oriented Software*. Tese (Doutorado) — George Mason University, Fairfax, Virginia, 1999.

[Lehman 1969]LEHMAN, M. *The Programming Process*. [S.l.], Dez. 1969.

[Lehman 1974]LEHMAN, M. *Programs, Cities, Students, Limits of Growth?* 1974. 211-229 p. Imperial College of Science and Technology Inaugural Lecture Series.

[Lehman 1978]LEHMAN, M. Laws of program evolution-rules and tools for programming management. In: *Proc. Infotech State of the Art Conference*. [S.l.: s.n.], 1978. p. 1V1–1V25.

[Lehman 1980]LEHMAN, M. Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, v. 68, n. 9, p. 1060– 1076, September 1980. Special Issue on Software Evolution.

[Lehman 1980]LEHMAN, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, v. 1, p. 213–221, 1980.

[Lehman e Ramil 2001]LEHMAN, M. M.; RAMIL, J. F. An approach to a theory of software evolution. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*. New York, NY, USA: ACM Press, 2001. p. 70–74. ISBN 1-58113-508-4.

[Lehman e Ramil 2001]LEHMAN, M. M.; RAMIL, J. F. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, J. C. Baltzer AG, Science Publishers, Red Bank, NJ, USA, v. 11, n. 1, p. 15–44, 2001. ISSN 1022-7091.

[Li e Offutt 1996]LI, L.; OFFUTT, A. J. *Algorithmic Analysis of the Impact of Changes on Object-Oriented Software*. [S.l.], Fev. 1996.

[Li e Offutt 1996]LI, L.; OFFUTT, A. J. Algorithmic analysis of the impact of changes to object-oriented software. In: *Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96)*. Washington, DC, USA: IEEE Computer Society, 1996. p. 171–184. ISBN 0-8186-7677-9.

[Loyall e Mathisen 1996]LOYALL, J.; MATHISEN, S. Using dependency analysis to support the software maintenance process. In: BOHNER, S. (Ed.). *Software Change Impact Analysis*. [S.l.]: IEEE Computer Society Press, 1996. p. 127–136.

[Ltd. 2006]LTD., D. *DeltaXML - Managing Change in an XML Environment*. 2006. `http://www.deltaxml.com/dxml/` (Acesso em Dez./2006).

[Madhavji, Fernandez-Ramil e Perry 2006]MADHAVJI, N. H.; FERNANDEZ-RAMIL, J.; PERRY, D. *Software Evolution and Feedback: Theory and Practice*. [S.l.]: John Wiley & Sons, 2006. ISBN 0470871806.

[Meyer 1997]MEYER, B. *Object-oriented software construction*. 2. ed. [S.l.: s.n.], 1997.

[Meyer 1970]MEYER, P. Conditional probability and independence. In: *Introductory Probability and Statistical Applications*. 2. ed. Reading, MA: Addison-Wesley Publishing Company, 1970. cap. 3, p. 39–41.

[Mockus e Votta 2000]MOCKUS, A.; VOTTA, L. G. Identifying reasons for software changes using historic databases. In: *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000. p. 120. ISBN 0-7695-0753-0.

[Moreton 1996]MORETON, R. A process model for software maintenance. In: BOHNER, S. (Ed.). *Software Change Impact Analysis*. [S.l.]: IEEE Computer Society Press, 1996. p. 29–33.

[Myers 1986]MYERS, E. W. An O(ND) difference algorithm and its variations. *Algorithmica*, v. 1, n. 2, p. 251–266, 1986.

[Orso et al. 2004]ORSO, A. et al. An empirical comparison of dynamic impact analysis algorithms. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. Washington, DC, USA: IEEE Computer Society, 2004. p. 491–500. ISBN 0-7695-2163-0.

[Pfleeger e Bohner 1990]PFLEEGER, S.; BOHNER, S. A framework for software maintenance metrics. In: *Proceedings of Conference on Software Maintenance*. [S.l.: s.n.], 1990. p. 320–327. ISBN 0-8186-2091-9.

[Pfleeger 1998]PFLEEGER, S. L. The nature of system change. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 15, n. 3, p. 87–90, 1998. ISSN 0740-7459.

[Pilato 2004]PILATO, M. *Version Control With Subversion*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004. ISBN 0596004486.

[Ren et al. 2004]REN, X. et al. Chianti: a tool for change impact analysis of java programs. In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*. [S.l.: s.n.], 2004. p. 432–448.

[Rijsbergen 1979]RIJSBERGEN, C. J. V. *Information Retrieval*. 2. ed. Dept. of Computer Science, University of Glasgow, 1979. Disponível em: <http://www.dcs.gla.ac.uk/Keith/Preface.html>.

[Ryder e Tip 2001]RYDER, B. G.; TIP, F. Change impact analysis for object-oriented programs. In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*. New York, NY, USA: ACM Press, 2001. p. 46–53. ISBN 1-58113-413-4.

[SEI 2008]SEI. *Capability Maturity Model Integration*. 2008. `http://www.sei.smu.edu/cmmi/` (Acesso em Jan/2008).

[Tan, Steinbach e Kumar 2005]TAN, P.-N.; STEINBACH, M.; KUMAR, V. *Introduction to Data Mining*. 1. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321321367.

[Tu e Godfrey 2002]TU, Q.; GODFREY, M. W. An integrated approach for studying architectural evolution. In: *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 127. ISBN 0-7695-1495-2.

[Turver e Malcolm 1994]TURVER, R. J.; MALCOLM, M. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, v. 6, n. 1, p. 35–52, 1994.

[Wilde e Huitt 1996]WILDE, N.; HUITT, R. Maintenance support for object-oriented programs. In: BOHNER, S. (Ed.). *Software Change Impact Analysis*. [S.l.]: IEEE Computer Society Press, 1996. p. 118–124.

[Xing e Stroulia 2005]XING, Z.; STROULIA, E. UMLDiff: an algorithm for object-oriented design differencing. In: *Proceedings of the 20th IEEE/ACM international Con-*

ference on Automated software engineering (ASE '05). New York, NY, USA: ACM Press, 2005. p. 54–65. ISBN 1-59593-993-4.

[Ying 2003]YING, A. T. T. *Predicting source code changes by mining revision history*. Dissertação (Mestrado) — University of British Columbia, Vancouver, November 2003.

[Ying, Ng e Chu-Carroll 2004]YING, A. T. T.; NG, R.; CHU-CARROLL, M. C. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 30, n. 9, p. 574–586, 2004. ISSN 0098-5589. Member-Gail C. Murphy.

[Zimmermann e Weißgerber 2004]ZIMMERMANN, T.; WEIßGERBER, P. Preprocessing CVS data for fine-grained analysis. In: *Proceedings of the First International Workshop on Mining Software Repositories*. [S.l.: s.n.], 2004. p. 2–6.

[Zimmermann et al. 2004]ZIMMERMANN, T. et al. Mining version histories to guide software changes. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. Washington, DC, USA: IEEE Computer Society, 2004. p. 563–572. ISBN 0-7695-2163-0.