Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Avaliar Refatoramentos Baseada no Impacto da Mudança

Melina Mongiovi Cunha Lima Sabino

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I 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

Rohit Gheyi
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S116a Sabino, Melina Mongiovi Cunha Lima.
Uma abordagem para avaliar refatoramentos baseada no impacto da mudança. -- 2013.
109 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

"Orientação: Prof. Dr. Rohit Gheyi".
Referências.

1. Engenharia de Software. 2. Refatoramentos. 3. Análise de Impacto. 4. Testes de Programas. I. Gheyi, Rohit. II . Título.

CDU 004.41(043)

"UMA ABORDAGEM PARA AVALIAR REFATORAMENTOS BASEADA NO IMPACTO DA MUDANÇA"

MELINA MONGIOVI CUNHA LIMA SABINO

DISSERTAÇÃO APROVADA EM 11/03/2013

ROHIT GHEYI, Dr., UFCG
Orientador(a)

ADALBERTO CAJUEIRO DE FARIAS, Dr., UFCG
Examinador(a)

ROBERTA DE SOUZA COELHO, Drª, UFRN
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Refatoramentos são transformações que melhoram a estrutura interna do programa preservando seu comportamento observável. Na prática, desenvolvedores utilizam testes de regressão e ferramentas para garantir que o refatoramento preservou o comportamento do programa. Entretanto, ferramentas de refatoramentos possuem bugs. Além disso, a coleção de testes pode ser modificada pela transformação aplicada manualmente ou pela ferramenta. Se a transformação não for aplicada corretamente, esta pode modificar a coleção de testes incapacitando-a de detectar a mudança comportamental. Por fim, a coleção pode não ser adequada para testar a transformação, pois os casos de testes podem não exercitar as entidades impactadas pela transformação. Este problema se torna maior se a coleção de testes for grande, tornando a execução de toda a coleção de testes custosa. Nós propomos uma abordagem para avaliar preservação de comportamento em refatoramentos baseada em geração automática de testes e análise de impacto da mudança. A abordagem analisa o impacto da mudança e gera automaticamente testes apenas para os métodos impactados pela transformação. Implementamos uma ferramenta chamada SafeRefactorImpact para avaliar a preservação de comportamento. Esta utiliza Safira, ferramenta que implementamos para realizar a análise de impacto. Avaliamos SafeRefactorImpact em um conjunto de 10 transformações aplicadas a um sistema real, e em uma técnica para testar implementações de refatoramentos. Além disso, comparamos SafeRefactorImpact com SafeRefactor, uma ferramenta que também avalia preservação de comportamento em refatoramentos mas não utiliza análise de impacto. Nós comparamos com relação às mudanças comportamentais identificadas, quantidade de métodos identificados para geração de testes, tempo total da análise da transformação, quantidade de casos de testes gerados e cobertura da mudança dos testes gerados. O SafeRefactorImpact conseguiu identificar mudanças comportamentais não identificadas pelo SafeRefactor, reduziu em torno de 60% o tempo para testar implementações de refatoramentos, mostrou-se menos sensível ao tempo limite passado para o gerador automático de testes, além da análise de impacto permitir a diminuição de algumas limitações do gerador automático de testes enfrentadas pelo SafeRefactor.

Abstract

Refactorings are transformations that improve the internal structure of the program while preserving its observable behavior. In practice, developers use regression tests and refactoring tools to ensure that the refactoring has preserved the behavior of the program. However, refactoring tools may have bugs. In addition, the test suite may be modified by the transformation applied manually or using a refactoring tool. If the transformation is applied incorrectly, it can change the test suite by disabling it to detect the behavioral change. Finally, the test suite may be inappropriate to test the transformation because the test cases may not exercise the change. This problem can get worse if the test suite is large. This way, it is time consuming executing the entire test suite. We propose an approach for evaluating whether a transformation is behavior preserving based on change impact analysis. This approach performs a change impact analysis and it automatically generates tests only to the methods impacted by the transformation. We implemented a tool called SafeRefactorImpact to evaluate behavior preservation. It uses Safira, a tool that we implemented to identify the methods impacted by a change in the program. We evaluate SafeRefactorImpact in a set of 10 transformations applied to a real system and to a technique to test refactoring implementations. Moreover, we compared the SafeRefactorImpact with SafeRefactor, a tool that also evaluates whether a transformation preserves the program behavior. The difference is that SafeRefactor does not use the impact analysis. The SafeRefactorImpact managed to identify behavioral changes not identified by SafeRefactor; it reduced in 60% the total time to analyze the refactoring implementations; it showed to be less sensitive to the time limit to generate the tests; and the impact analysis allows the reduction of some limitations of the automatic tests generator faced by the SafeRefactor.

Agradecimentos

Agradeço primeiramente a Deus, que me deu o privilégio da vida e a benção da saúde;

Ao meu pai, Giuseppe Mongiovi, o grande incentivador da minha história, por todo amor, dedicação e companheirismo. Mesmo em memória, pude sentir sua constante presença, me guiando e me ajudando como sempre fez toda sua vida;

À minha mãe, Graça Mongiovi, pelo grande carinho, amor, preocupação e apoio, não medindo esforços para que eu conquistasse mais esta etapa da minha vida. Obrigada por estar sempre presente, vibrando com minha felicidade e me dando forças nos momentos difíceis;

Agradeço ao meu esposo Gusthavo e à minha filha Camilla, pelo amor, compreensão das minhas ausências e apoio em todos os momentos. Aos meus irmãos Finuzza e Angelo por sempre me encorajarem e torcerem por mim;

Meus sinceros agradecimentos ao meu amigo e orientador Rohit, que me ensinou os primeiros passos da pesquisa, sempre com muita paciência, dedicação, empenho, disponibilidade, e vontade de ensinar e orientar. Sem ele esse trabalho não seria possível. Agradeço também pelas oportunidades concedidas e preocupação com minha formação;

Agradeço a Gustavo Soares, grande amigo que acompanhou toda minha caminhada para realização deste trabalho. Obrigada pelo constante incentivo e ajuda durante o mestrado;

Aos professores Tiago Massoni, Paulo Borba, Dalton Serey, Patrícia Machado, Adalberto Cajueiro, Roberta Coelho e Augusto Sampaio pelas sugestões e contribuições neste trabalho;

Agradeço a todos os amigos do SPG, que contribuíram de alguma nesta pesquisa. Agradeço especialmente a Larissa Braz, Catuxe Varjão e Laerte Xavier pelas boas conversas, trocas de ideias, e momentos de descontração;

Aos professores e funcionários da COPIN e do DSC;

Ao CNPq pelo apoio e suporte financeiro fornecidos a este trabalho.

Conteúdo

1 Introdução **1**

1.1 Problema 2

1.2 Solução 3

1.3 Avaliação 5

1.4 Contribuições 5

1.5 Organização 6

2 Fundamentação Teórica **7**

2.1 Testes 7

2.1.1 Definição 7

2.1.2 Teste de Regressão 8

2.1.3 Critério de Adequação de Teste 10

2.1.4 Geração de Casos de Testes 14

2.1.5 Teste de Implementação de Refatoramentos 16

2.2 Análise de Impacto da Mudança 19

2.2.1 Definição 19

2.2.2 Abordagens: Estática e Dinâmica 20

2.2.3 Estado da Arte 24

2.3 Refatoramento 29

2.3.1 Definição 30

2.3.2 Problema 33

2.3.3 Estado da Arte 36

3 Safira: Um analisador de impacto da mudança **39**

3.1 Exemplo Motivante 40

3.2 Abordagem 42

3.2.1 Visão Geral 42

3.2.2 Subtransformações 42

3.2.3 Leis da Análise de Impacto 44

3.2.4 Exemplo 47

3.2.5 Especificação formal 48

3.3 Cobertura da Mudança 50

3.3.1 Nível de Métodos 51

3.3.2 Nível de *Statements* 52

3.4 Safira 52

3.4.1 Arquitetura 54

3.5 Abordagens Similares 55

3.5.1 Chianti 56

3.5.2 FaultTracer 57

3.5.3 Comparação 58

3.6 Limitações 64

3.6.1 Análise de Fluxo de Dados 64

3.6.2 Análise de Classes de Bibliotecas 65

4 SafeRefactorImpact: Uma Abordagem para Avaliar Refatoramentos Baseada no Impacto da Mudança **68**

4.1 Abordagem 71

4.1.1 Visão Geral 71

4.1.2 Primeira Fase: Análise de Impacto da Mudança 72

4.1.3 Segunda Fase: Geração de Testes 73

4.1.4 Terceira Fase: Execução dos Testes 73

4.1.5 Quarta Fase: Avaliação dos Resultados 74

4.1.6 Exemplo 74

4.2 SafeRefactorImpact 77

4.2.1 Arquitetura 78
4.3 Limitações 80

5 Avaliação **81**
5.1 JHotDraw 81
5.1.1 Definição 81
5.1.2 Planejamento 85
5.1.3 Configurações Experimentais 87
5.1.4 Resultados 88
5.1.5 Discussão 93
5.1.6 Conclusões 100
5.2 Teste de Implementação de Refatoramentos 101
5.2.1 Definição 101
5.2.2 Planejamento 103
5.2.3 Configurações Experimentais 105
5.2.4 Resultados 106
5.2.5 Discussão 107
5.2.6 Conclusões 110
5.3 Ameaças a Validade 110
5.3.1 Validade Interna 110
5.3.2 Validade Externa 111
5.3.3 Validade de Construção 112

6 Conclusões **113**
6.1 Trabalhos Relacionados 115
6.1.1 Análise de Impacto 116
6.1.2 Refatoramento 117
6.1.3 Testes 120
6.2 Trabalhos Futuros 121

A Especificação em Alloy das Leis da Análise de Impacto **130**

Lista de Figuras

2.1 Análise de cobertura dos testes realizado pela ferramenta EclEmma. 13
2.2 Grafo de dependência dos métodos do programa 2.3 21
2.3 Adicionar método afeta diversas partes do programa. 27
2.4 Refatoramento *PullUpField*: move atributos de uma classe para sua super classe. 33
2.5 *PullUpField* aplicado pelo Eclipse que não preserva o comportamento. . . . 34

3.1 Transformação que move método para classe filha e muda o comportamento do programa 41
3.2 Análise de Impacto da Mudança 43
3.3 Resultado da análise de Safira na transformação ilustrada da Figura 3.1 . . . 53
3.4 Arquitetura de Safira 55
3.5 Renomear método resulta em sombreamento de método importado 63
3.6 Transformação aplicada no programa altera valor de atributo através de chamada de método e muda o comportamento do programa. 65
3.7 Transformação aplicada no programa remove um método que sobrescreve método de biblioteca e muda comportamento do programa. 67

4.1 Renomear método introduz mudança comportamental por habilitar sobrescrita. 69
4.2 Abordagem para detectar mudanças comportamentais baseada em geração automática de testes e análise de impacto da mudança. 72
4.3 Pull Up Method aplicado pelo Eclipse 3.4.2 que não preserva comportamento 75
4.4 Arquitetura do SafeRefactorImpact 79

5.1 Técnica de testar implementação de refatoramentos. 102

Lista de Tabelas

2.1 Métodos afetados por cada método do programa 2.3, considerando todas as possíveis execuções do programa 22

3.1 Subtransformações utilizadas na análise de impacto realizada por Safira . . 44
3.2 Mudanças atômicas consideradas nas análises de Chianti e FaultTracer . . . 57
3.3 Tabela comparativa das abordagens de Chianti, FaultTracer e Safira 60

5.1 Características das versões do JHotDraw avaliadas no experimento. 86
5.2 Design fatorial completo utilizado no experimento do JHotDraw. Cada abordagem é executada uma vez utilizando cada tempo limite. 88
5.3 Métricas acurácia, precisão e revocação calculadas para cada transformação avaliada do JHotDraw. 89
5.4 Quantidade de métodos identificados pelo SafeRefactorImpact e SafeRefactor. 90
5.5 Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 1s. 91
5.6 Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 5s. 92
5.7 Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 60s. 93
5.8 Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 120s. 94
5.9 Redução da quantidade de casos de testes gerados nas transformações das versões 650 e 700 e na média de todas as transformações. 97
5.10 Redução do tempo total para avaliar as transformações das versões 344 e 650 e a média de redução de todas as transformações. 98

5.11 Aumento da cobertura da mudança dos testes gerados pelo SafeRefactorImpact em relação aos gerados pelo SafeRefactor, nas transformações das versões 650 e 700, e na média de todas as transformações. 99
5.12 Resumo das implementações de refatoramentos avaliadas no Eclipse e JRRT; Escopo = Pacotes (P), Classes (C), Atributos (A) e Métodos (M) 104
5.13 Design fatorial fracionário utilizado no experimento de teste de implementação de refatoramentos. 105
5.14 Resultados do experimento nas implementações de refatoramentos do Eclipse. 107
5.15 Resultados do experimento nas implementações de refatoramentos do JRRT. 108
5.16 Falhas não detectadas pelo SafeRefactor utilizando um tempo limite de 0,2s. 109

Lista de Códigos Fonte

2.1 Teste gerado pelo Randoop para API Collection da JDK 16
2.2 Um subconjunto do metamodelo de Java especificado em Alloy. 18
2.3 Métodos de um programa 21
2.4 Coleção de testes do programa 5.4 23
2.5 Programa Original 27
2.6 Programa Modificado 27
2.7 Programa com *bad smell*: método longo 31
2.8 Médodo $imprimeDivida()$ do Programa 2.7 refatorado 32
2.9 Programa Original 34
2.10 Programa Modificado 34
3.1 Programa Original 41
3.2 Programa Modificado 41
3.3 Parte da Especificação em Alloy das Leis de Análise de Impacto 49
3.4 Asserção 50
3.5 Coleção de testes do programa ilustrado na Figura 3.1 50
3.6 Exemplo de como utilizar Safira para analisar uma transformação 53
3.7 Classe de Safira que Representa uma Entidade de um Programa 54
3.8 Programa Original 63
3.9 Programa Modificado 63
3.10 Coleção de Testes 63
3.11 Programa Original 65
3.12 Programa Modificado 65
3.13 Programa Original 67
3.14 Programa Modificado 67

4.1 Programa Original. 69
4.2 Programa Modificado. 69
4.3 Coleção de Testes do Programa Ilustrado na Figura 4.1 70
4.4 Programa Original 75
4.5 Programa Modificado 75
4.6 Teste gerado pelo Randoop 77
4.7 Exemplo de como utilizar SafeRefactorImpact para analisar uma transformação 78
Epecificação em Alloy das Leis da Análise de Impacto 130

Capítulo 1

Introdução

Com o crescente aumento da dependência de usuários por sistemas de software e a progressiva dependência por programas de qualidade, é cada vez maior a demanda por uma contínua evolução do sistema [3]. A necessidade de uma mudança pode surgir para diversos propósitos, como por exemplo, adição de novos requisitos, substituição de uma tecnologia obsoleta ou correção de novos defeitos. Porém, a estrutura originalmente definida pode não acomodar as mudanças propostas para o sistema. Por isso, é necessário reestruturar o programa, para tornar mais fácil a manutenção e compreensão do software pelos desenvolvedores. Esse é o tipo de evolução que se denomina perfectiva.

Na evolução perfectiva o objetivo é modificar o software para melhorar a sua manutenabilidade. Um tipo de evolução perfectiva é refatoramento, que tem o objetivo de melhorar a estrutura interna do código preservando seu comportamento. O termo refatoramento foi originalmente proposto por Opdyke [51] em sua tese de doutorado. Ele propôs um conjunto de refatoramentos e precondições para garantir que, ao se aplicar o refatoramento, seja mantido o comportamento do programa. Entretanto, ele não provou formalmente as precondições. Posteriormente, Tokuda and Batory [74] mostraram que as precondições propostas por Opdyke não garantiam preservação de comportamento para certos tipos de transformação. Mais tarde, refatoramentos foi popularizado na prática por Fowler [20].

1.1 Problema

Refatoramentos podem ser aplicados manualmente ou com ajuda de ferramentas como Eclipse e NetBeans que implementam vários tipos de refatoramentos. Refatoramentos aplicados manualmente são tendenciosos a erros e custosos pois, principalmente em programas grandes, é difícil checar se o mesmo preserva o comportamento. Além disso, em programas orientados a objetos, uma mudança local pode impactar várias outras partes do código.

As ferramentas de refatoração implementam os refatoramentos utilizando um conjunto de precondições que precisam ser checadas antes do refatoramento ser aplicado. Essas implementações são definidas com base na experiência dos desenvolvedores. Algumas abordagens para definir precondições formalmente e implementar refatoramentos foram propostas para analisar alguns aspectos da linguagem, como por exemplo, acessibilidade, tipos, associação de nomes, fluxo de dados e fluxo de controle. Schafer et al. [64; 61] propuseram uma ferramenta, o JastAdd Refactoring Tools (JRRT), que formaliza algumas precondições de 17 refatoramentos para Java e os implementa. O objetivo foi aumentar a corretude das implementações do Eclipse. Steimann e Thies [70] definiram um conjunto de restrições relacionadas a visibilidade que alguns refatoramentos devem satisfazer para preservar o comportamento. Entretanto, as especificações não foram provadas formalmente e funcionam apenas para um conjunto restrito de transformações. Borba et al. [7] provaram formalmente as precondições de algumas transformações para um conjunto restrito de construções da linguagem. No entanto, essa abordagem funciona para um subconjunto de Java e não considera todas as construções da linguagem. O ideal seria definir formalmente todas as condições necessárias para um refatoramento preservar comportamento com relação a uma semântica formal considerando todas as construções da linguagem. Entretanto, esta atividade foi considerada um desafio por Schafer et al. [62]. Logo, as ferramentas de refatoramentos podem aplicar transformações que modificam o comportamento do programa [68; 66].

Como já vimos, refatoramentos aplicados manualmente ou com ajuda de ferramentas não são seguros e precisam ser avaliados quanto a preservação de comportamento. Na prática, desenvolvedores utilizam testes de regressão para avaliar o programa após aplicar o refatoramento. No entanto, essa também pode não ser uma forma segura, já que os mesmos

podem ser modificados pela transformação. Por exemplo, um refatoramento que renomeia um método do programa pode modificar a coleção de testes. Os casos de testes que exercitam esse método serão modificados para chamar o método com o novo nome. Soares et al. [66] encontraram alguns *bugs* em implementações do refatoramento *rename method* [66]. Se utilizadas, estas implementações podem também modificar o comportamento da coleção de testes.

Com o objetivo de tentar melhorar este cenário, Soares et al. [68] propuseram o SafeRefactor, uma ferramenta que analisa preservação de comportamento em refatoramentos. A partir do programa original e do programa refatorado, o SafeRefactor gera automaticamente uma coleção de testes para todos os métodos comuns às duas versões do programa. Desta forma, é utilizada uma única coleção de testes para avaliar o comportamento do programa original e do modificado. Se os resultados dos testes forem diferentes, o SafeRefactor conclui que o comportamento do programa foi modificado. Entretanto, alguns casos de testes gerados pelo SafeRefactor podem não ter sido impactados pela mudança. Ao avaliar transformações aplicadas a sistemas grandes, esta estratégia pode se tornar inviável já que o gerador automático de testes poderá considerar um conjunto de métodos grande. Possivelmente, o SafeRefactor vai avaliar a transformação com uma coleção que exercitará pouco a mudança. Esses problemas também podem ocorrer ao avaliar a transformação com a coleção de testes do programa, pois os testes podem não exercitar a mudança. Foi o que mostrou um estudo realizado por Rachatasumrit e Kim [55], em três grandes projetos *open source*. Foi avaliado que apenas 22% dos refatoramentos aplicados nos programas são cobertos pelos testes de regressão.

1.2 Solução

Neste trabalho, nós propomos uma abordagem para avaliar se dois programas possuem o mesmo comportamento observável baseada em geração automática de testes e análise de impacto da mudança. O programa original e o programa modificado são analisados e, caso seja identificada uma mudança comportamental, é reportado um caso de teste. A abordagem consiste de 4 passos sequenciais: análise de impacto da mudança para identificar os métodos impactados comuns às duas versões do programa, geração automática de testes apenas para

os métodos impactados, execução dos testes nas duas versões do programa e análise dos resultados. Se algum caso de teste falhar apenas no programa refatorado, indica que os programas possuem comportamentos diferentes. Caso contrário, temos uma confiança maior que a transformação preserva o comportamento.

Para identificar os métodos impactados pela mudança, nós propomos também um analisador de impacto. A abordagem analisa duas versões de um programa (o programa original e o programa modificado) e identifica os métodos impactados que podem ter mudado de comportamento devido à transformação. Primeiramente, a transformação é caracterizada decompondo sua diferença em um conjunto de subtransformações. Em seguida, o impacto de cada subtransformação no programa é calculado. Para isso, definimos leis que identificam o conjunto de métodos impactados por cada subtransformação considerada em nossa análise de impacto. O conjunto de métodos diretamente impactados é definido pela união dos conjuntos impactados por cada subtransformação. Além disso, também consideramos como impactados os métodos que exercitam direta ou indiretamente algum método do conjunto de métodos diretamente impactados em qualquer nível de indireção. Nós implementamos a abordagem para analisar transformações em Java, em uma ferramenta chamada Safira.

Com o objetivo de avaliarmos o quanto uma coleção de testes cobre as entidades impactadas, definimos uma métrica de cobertura da mudança a nível de *statements* e métodos. A partir de duas versões de um programa e sua coleção de testes, é medido o quanto da mudança está sendo coberta. Para cada transformação avaliada pelo SafeRefactorImpact, além de indicar se houve preservação de comportamento, o mesmo indica o quanto a coleção de testes usada na avaliação cobre a mudança. Os cálculos e medições são feitos de forma automática em um módulo do SafeRefactorImpact. Por fim, fizemos um estudo comparativo de Safira com duas ferramentas similares do estado da arte: Chianti [56] e FaultTracer [80]. Nesse estudo, identificamos vantagens e desvantagens de cada abordagem com relação ao custo e corretude das análises, utilizando alguns critérios de comparação. Por exemplo, se a análise da abordagem é estática ou dinâmica ou se depende de uma coleção de testes.

O SafeRefactorImpact é uma extensão do SafeRefactor [68]. O grande diferencial é a análise de impacto da mudança realizada pelo SafeRefactorImpact. O mesmo gera testes apenas para o que foi identificado por Safira, enquanto o SafeRefactor gera testes para todos os métodos em comum do programa. O SafeRefactorImpact utiliza Safira para realizar a

análise de impacto da mudança e o Randoop [53], um gerador automático de testes de unidade, para gerar os testes. Os testes são gerados dentro de um tempo limite informado ao Randoop. Todos os passos da ferramenta são feitos de forma automática.

1.3 Avaliação

Avaliamos nossa abordagem em dois experimentos. No primeiro experimento avaliamos transformações aplicadas em 10 versões do JHotDraw, um framework para desenvolvimento de editores gráficos. No segundo experimento, avaliamos programas gerados automaticamente e refatorados por ferramentas de refatoração (Eclipse e JRRT). Para este conjunto, utilizamos a técnica de testar implementação de refatoramentos proposta por Soares et al. [66]. Nos dois conjuntos, nós avaliamos o SafeRefactorImpact e o SafeRefactor para analisar o benefício da análise de impacto realizada pelo SafeRefactorImpact.

No primeiro conjunto, vimos que na análise de programas grandes, a análise de impacto do SafeRefactorImpact reduziu em média, 93% a quantidade de métodos a serem testados e assim, focando a geração de testes apenas nos métodos impactados, foi possível encontrar mais mudanças comportamentais que o SafeRefactor. Em 3 versões do JHotDraw, o SafeRefactorImpact identificou mudanças comportamentais que o SafeRefactor não identificou no experimento. Em outras 2 versões, o SafeRefactor precisou de um tempo limite maior que o SafeRefactorImpact para identificar a mudança.

No segundo conjunto, vimos que utilizar o SafeRefactorImpact torna a técnica de testar implementação de refatoramentos mais rápida. A análise de impacto do SafeRefactorImpact permite que seja utilizado um tempo limite de geração de testes menor. A técnica utilizando o SafeRefactorImpact reduziu o tempo total para avaliar as implementações de refatoramentos do Eclipse e JRRT, em 25,8h e 31h, respectivamente. Esses valores correspondem a uma redução de 60%.

1.4 Contribuições

Em resumo, as principais contribuições deste trabalho são:

- Uma abordagem para avaliar refatoramentos baseada em geração de testes e análise de

impacto da mudança (uma extensão da abordagem do SafeRefactor);

- Uma ferramenta SafeRefactorImpact que implementa a abordagem;
- Safira, um analisador de impactado da mudança;
- Avaliação do SafeRefactorImpact em 10 implementações de refatoramentos do Eclipse e 8 do JRRT e 10 transformações aplicadas a sistemas reais.

1.5 Organização

No capítulo seguinte, apresentamos a fundamentação teórica necessária para o entendimento do nosso trabalho (Capítulo 2). Em seguida, no Capítulo 3 descrevemos a abordagem proposta para analisar o impacto de uma mudança no código. No Capítulo 4, apresentamos a abordagem para avaliar preservação de comportamento em refatoramentos, baseada em análise de impacto da mudança. Descrevemos a avaliação da nossa abordagem no Capítulo 5. Por fim, apresentamos as considerações finais, trabalhos futuros e trabalhos relacionados no Capítulo 6.

Capítulo 2

Fundamentação Teórica

Neste capítulo, abordamos conceitos relevantes que foram utilizados como base para realização deste trabalho. Inicialmente apresentamos conceitos de testes de software, com ênfase em cobertura de testes e automação de testes. Em seguida, falamos de análise de impacto da mudança, onde focamos em sistemas orientados a objetos. Por fim, apresentamos conceitos fundamentais de refatoramento, bem como alguns problemas e abordagens propostas.

2.1 Testes

Nesta seção, vamos mostrar alguns fundamentos teóricos na área de testes. Vamos iniciar a seção apresentando alguns conceitos básicos de testes. Em seguida, falamos de testes de regressão e critério de adequação de testes. Por fim, falamos de geração de casos de testes e mostramos algumas abordagens que testam implementações de refatoramentos.

2.1.1 Definição

O software deve ser previsível e consistente, não oferecendo nenhuma surpresa para o usuário. Teste de software é o processo de execução de um programa com o objetivo de avaliar se ele funciona corretamente de acordo com o que foi especificado [50]. O objetivo principal de uma atividade de testes é revelar a maior quantidade possível de falhas com o mínimo de esforço. Com base na definição de Binder [5], vamos esclarecer alguns conceitos importantes a atividade de teste de software:

- **Falha**: é o comportamento operacional do software diferente do esperado pelo usuário.
- **Falta**: é definida como a ausência de código ou a presença de código incorreto em um programa que pode provocar a falha.
- **Erro**: é uma ação humana que resulta em uma falta no sistema.

A execução dos testes acontece em diferentes níveis ao longo do desenvolvimento do software. De acordo com Rocha et al. [58], os principais níveis de software são:

- **Teste de Unidade**: tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. Nesse tipo de teste, o universo alvo são os métodos dos objetos ou mesmo pequenos trechos de código.
- **Teste de Integração**: visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.
- **Teste de Sistema**: avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software. Ele avalia se o produto satisfaz seus requisitos.
- **Teste de Regressão**: teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Essa estratégia de teste será descrita com mais detalhes na próxima seção.

2.1.2 Teste de Regressão

Testes de regressão é uma atividade de teste que é realizada para aumentar a confiança de que as mudanças feitas no programa não prejudicaram o seu comportamento. Essa atividade consiste em executar a cada nova versão do software ou a cada ciclo, todos os testes que já foram executados nas versões ou ciclos de teste anteriores do sistema. Como o software evolui, o conjunto de testes tende a crescer, tornando a execução de toda a coleção de testes

custosa. Muitas abordagens já estudaram formas de melhorar esse problema. De acordo com Yoo e Harman [79], as três principais linhas são:

- **Minimização de Casos de Teste**: foca em diminuir a coleção de testes removendo os testes redundantes, desnecessários ou obsoletos.

- **Seleção de Casos de Teste**: seleciona um subconjunto de casos de testes que serão utilizados para testar uma mudança feita no programa. Em atividades de testes de regressão, a seleção de casos de testes identifica os testes relevantes para testar as mudanças feitas entre a versão atual e a versão anterior do sistema que está sendo testado. Várias abordagens foram propostas para selecionar casos de testes de regressão. Os detalhes do processo de seleção dos testes diferem de acordo com a forma como uma técnica específica define, procura e identifica alterações no programa em teste. Ryder and Tip [60] propuseram uma abordagem para selecionar os testes de regressão que foram afetados por um conjunto de mudanças feitas no programa. Para identificar as mudanças, é realizada uma análise de impacto utilizando análise estática do código e construção de grafos de chamadas. Agrawal et al. [1] introduziram uma família de técnicas de seleção de casos de testes baseadas em diferentes abordagens de fatiamento de programas. Chen et al. [10] propuseram uma ferramenta, TestTube, que utiliza uma técnica baseada em modificação, para selecionar casos de testes. TestTube particiona o programa testado em entidades de programas e monitora a execução dos testes para estabelecer conexões entre casos de testes e as entidades executadas.

- **Priorização de Casos de Teste**: determina uma ordem de execução dos testes, a partir da priorização dos testes mais propensos a detectar faltas no programa. Elbaum et al. [18], realizaram um estudo empírico para comparar algumas técnicas de priorização de casos de testes. Os resultados mostraram que todas as técnicas consideradas no estudo melhoram a detecção de faltas da coleção. As técnicas comparadas no estudo incluem estratégias de cobertura de nós ou de métodos, indexação de faltas e diferenças sintáticas entre programas. Técnicas de baixa granularidade ou seja, que analisam código a nível de *statements*, superaram as técnicas de alta granularidade, que analisam o código a nível de métodos, com uma diferença relativamente pequena. Embora as técnicas de baixa granularidade sejam um pouco melhores para detectar faltas, elas são

mais custosas e mais intrusivas. Por fim, eles concluem que a melhor técnica para ser utilizada pode variar dependendo do programa a ser testado.

2.1.3 Critério de Adequação de Teste

Atividades de teste de software são imprescindíveis no processo de desenvolvimento de um sistema e cumprem papel fundamental no controle da qualidade do sofware. É amplamente reconhecida na engenharia de software a importância de testes para avaliar o sistema. Mas, como saber se uma coleção de testes está adequada para testar o programa? Quando finalizar a atividade de testes e entregar o produto? Todo o código está sendo testado? Um critério de adequação de testes pode ajudar a responder essas perguntas.

Uma questão importante no gerenciamento de uma atividade de teste de software é garantir que os objetivos dos testes sejam conhecidos e definidos em termos do que pode ser medido. Um critério de teste de software é o critério que define o que constitui um teste adequado, ou seja, define quais propriedades de um programa precisam ser exercitadas para a coleção de testes ser considerada completa [23]. A maioria dos critérios de teste propostos na literatura especificam explicitamente requisitos específicos do teste. Eles são regras objetivas aplicadas pelos gerentes de projeto para esta finalidade [82]. Por exemplo, a cobertura de nós é uma exigência de teste em que todos os nós do programa devem ser exercitados. Então, se este é o critério de teste utilizado, o objetivo do teste deve ser satisfazer esta exigência.

Existem outros critérios de testes, como por exemplo, adequação de mutação, que utiliza teste de mutação para avaliar o quão boa está uma coleção de testes [14]. Nesta atividade, faltas artificiais são inseridas no programa para avaliar se os testes são capazes de detectá-las. O critério é satisfeito se os testes conseguirem detectar todas as faltas. Neste trabalho, daremos ênfase para o critério cobertura. A seguir falamos um pouco sobre cobertura de testes e cobertura da mudança e mostramos algumas ferramentas que calculam a cobertura.

Cobertura de Teste

Cobertura de testes é uma métrica utilizada em teste de software, que indica o grau em que o programa está sendo testado. Ela pode ser expressa por cobertura de critérios estruturais, funcionais e baseados em defeitos. A cobertura de critérios estruturais baseada em código-

fonte mede a quantidade de código executado pelos testes em comparação com a quantidade de código não executado. As principais estratégias de medir a cobertura do código são:

- Cobertura de *Statements*: indica a porcentagem de *statements* executados pelos testes.
- Cobertura de Ramos: indica a porcentagem de transferências de controle, por exemplo, um *if - then - else*, exercitadas pelos testes.
- Cobertura de Caminhos: indica a porcentagem de caminhos de execução, a partir da entrada até a saída do programa, executados durante a execução dos testes.

Cobertura da Mudança

A métrica cobertura da mudança mede a porcentagem de entidades modificadas ou impactadas por uma mudança no código exercitadas pelos testes. Ren et al. [56] propuseram um analisador de impacto, Chianti, que decompõe uma transformação feita no código em um conjunto de mudanças atômicas. A abordagem de Chianti identifica as mudanças atômicas que são exercitadas pelos testes. Posteriormente, Wloka et al. [78] introduziram um novo conceito ao JUnit: barra amarela na ferramenta JUnitMX. A barra amarela indica que todos os testes estão passando mas a mudança realizada no código não está sendo totalmente coberta pelos testes. A ferramenta utiliza o Chianti para realizar a análise de impacto da mudança. Ela calcula a cobertura da mudança em termos de mudanças atômicas exercitadas pelos testes.

Rachatasumrit e Kim [55] realizaram um estudo do impacto da aplicação de refatoramentos nos testes de regressão. No estudo eles examinam se os refatoramentos são testados pelos coleção de testes de regressão, a quantidade de testes que exercitam um refatoramento e os testes que falharam devido a um refatoramento. Eles utilizam o FaultTracer [80], um analisador de impacto da mudança que estende o Chianti. Neste trabalho eles utilizam a métrica cobertura da mudança e a define como a porcentagem de métodos e atributos, que fazem parte de mudanças atômicas identificadas por FaultTracer em um refatoramento, exercitados pelos testes. A cobertura da mudança é utilizada para examinar se um refatoramento está sendo bem testado.

Ferramentas de Cobertura

Como já vimos, medir a cobertura de critérios estruturais no programa ajuda ao desenvolvedor avaliar a qualidade da sua coleção de testes. No entanto, não é fácil realizar esta tarefa manualmente, principalmente em grandes sistemas. Por isso, existem várias ferramentas que medem, de forma automática, a cobertura de testes. A seguir, falaremos sobre algumas dessas ferramentas para Java:

- **EMMA**

  EMMA[1] é um conjunto de ferramentas *open-source* que medem e informam a cobertura dos testes no código Java. Ela mede a cobertura a nível de classes, métodos, linhas de código e blocos básicos. EMMA informa até quando uma linha de código foi apenas parcialmente coberta. Três tipos de relatórios contendo informações sobre a cobertura dos testes podem ser gerados: HTML, XML e TXT. Esses relatórios podem destacar entidades que obtiveram uma cobertura menor que um valor passado pelo usuário. Além disso, relatórios obtidos por diferentes instrumentações ou execuções de testes podem ser acoplados em um só relatório. EMMA pode ser executada via linha de comando ou utilizando o ANT para rodar *scripts* com suas tarefas.

  O relatório HTML é mais fácil de ser visualizado pelo usuário que deseja ver as informações da cobertura de seus testes. Entretanto, para os dados serem analisados por um programa a fim de realizar cálculos ou outro tipo de análise sobre eles, é mais adequado o formato XML, por sua facilidade em ser manipulado utilizando algumas APIs já existentes.

- **EclEmma**

  Baseando-se na ferramenta EMMA, foi criado o EclEmma [2], um *plugin* para o Eclipse que utiliza EMMA para medir a cobertura. Depois de instalado o *plugin*, aparece no menu de execução dos testes a opção de rodar os testes com o EclEmma. A Figura 2.1 ilustra uma análise da ferramenta. As linhas marcadas em verde foram completamente cobertas pelos testes, as linhas em amarelo foram cobertas parcialmente e as linhas em

[1] http://emma.sourceforge.net/

[2] http://www.eclemma.org/

vermelho não foram cobertas. Além disso, a ferramenta também gera relatórios nos formatos HTML, XML e CSV.

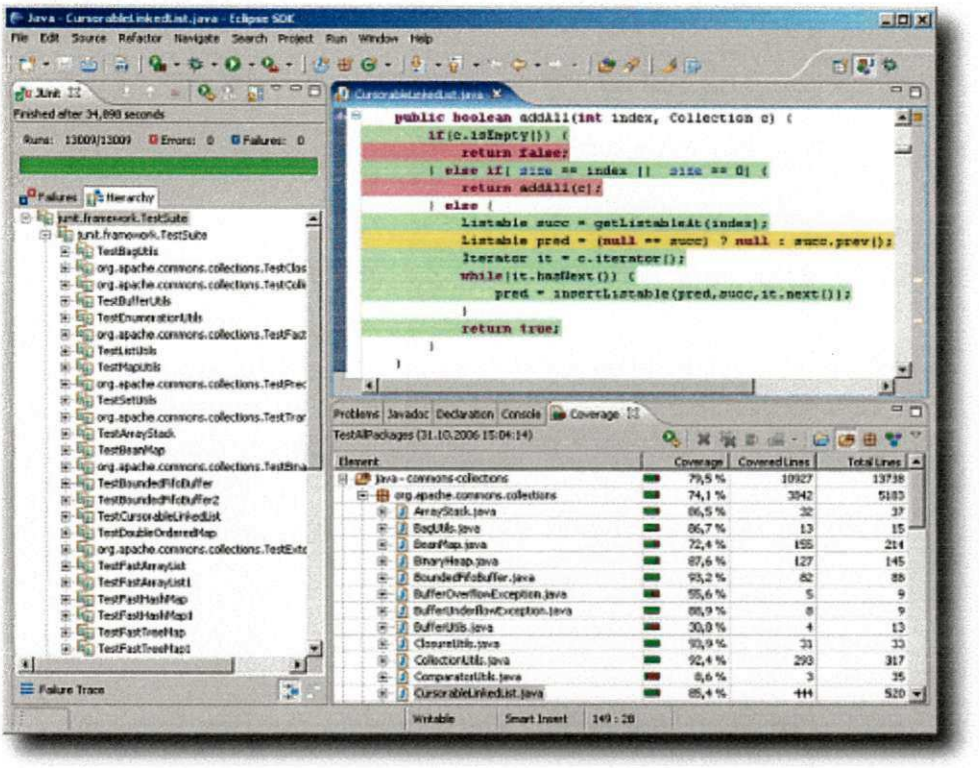


Figura 2.1: Análise de cobertura dos testes realizado pela ferramenta EclEmma.

- **Cobertura**

  Cobertura [3] é uma ferramenta gratuita para programas Java, que calcula a porcentagem de código exercitada pelos testes. Ela pode ser usada para identificar partes de um programa que não estão sendo cobertas pelos testes. A ferramenta calcula a porcentagem de linhas e ramos cobertos por cada classe, pacote ou de todo o programa. Ela pode ser executada via linha de comando ou pelo ANT. Cobertura reporta os resultados em relatórios HTML ou XML. Os relatórios apresentam os dados em vários níveis do programa, desde todo o programa até uma linha de código. Outra funcionalidade interessante da ferramenta é apresentar a complexidade McCabe de cada classe e a média de complexidade dos pacotes de um projeto.

[3] http://cobertura.sourceforge.net/

2.1.4 Geração de Casos de Testes

À medida que o software se torna maior e mais complexo, atividades de testes manuais são cansativas, demoradas e principalmente, susceptíveis a erros. Além disso, os testes manuais são executados poucas vezes e dificilmente alcançam uma boa cobertura do programa. Com isso, dificilmente os testes manuais serão suficientes para realizar uma atividade de testes de qualidade. Ao contrário dos testes manuais, os testes automatizados podem ser executados sempre que desejado, sem necessitar o esforço humano dos testes manuais. A automação de testes tem o objetivo de aplicar técnicas e ferramentas para diminuir o esforço humano nas atividades de testes [34]. Esse tipo de atividade também pode incluir a geração automática de testes.

As principais estratégias de geração de testes são: geração sistemática e geração aleatória [25] de casos de teste. Dentro da geração sistemática, destacam-se algumas técnicas: geração exaustiva de casos de testes [46], encadeamento (*chaining*) [19] e execução simbólica [77]. Outras técnicas foram propostas que utilizam algoritmos evolucionários [37] e desenvolvimento guiado por contratos [43]. A seguir, apresentamos três geradores automáticos de testes de unidade: AutoTest, TestFul e Randoop. Daremos um enfoque maior neste último, pois utilizamos neste trabalho.

AutoTest

O AutoTest [48] implementa um método chamado Desenvolvimento Guiado por Contratos [43]. Esse método de desenvolvimento é baseado em um novo mecanismo que extrai casos de testes de falhas produzidas por execuções de programadores. Ele explora ações que os desenvolvedores realizam como parte do seu processo de escrever código. Desta forma, o método adquire o conhecimento dos desenvolvedores sobre a semântica e estrutura do código. A abordagem é baseada em contratos de código que atuam como oráculo dos casos de testes. Os casos de testes são automaticamente criados enquanto os programadores implementam seu programa. A ferramenta possui uma versão implementada para Java: JAutoTest.

TestFul

TestFul [37] é um *framework* para geração automática de testes de unidade para classes Java. Ele implementa uma abordagem que reconhece e reutiliza configurações de estados úteis para exercitar diferentes funcionalidades, combinado com uma orientação capaz de atingir elementos estruturais não cobertos e bons desempenhos globais. Além disso, usa um algoritmo evolucionário para identificar e selecionar os melhores testes. Como função de aptidão, foi utilizada a cobertura de nós e de ramos. Quanto maior a cobertura do teste, maior a probabilidade do teste ser o melhor. A ferramenta busca pelos menores testes que possuem a maior cobertura.

Randoop

Randoop [53] é um gerador automático de testes em Java. Ele gera aleatoriamente testes de unidade, baseando-se no *feedback* do comportamento do programa. Em uma visão geral, a técnica gera sequências de invocações de métodos e construtores das classes que estão sendo testadas. Ele executa as sequências que foram criadas e utiliza os resultados da execução para criar asserções que capturam o comportamento do programa. As asserções são utilizadas para criar os casos de testes.

Randoop gera dois tipos de testes de unidade: testes reveladores de erros que falham quando executados, indicando um potencial erro em uma ou mais classes do programa, e testes de regressão, que passam quando executados e podem ser usados para aumentar a coleção de testes de regressão do sistema.

Os testes reveladores de erros mostram um uso específico de uma classe testada, que gera a violação de um contrato em tempo de execução. Um contrato é uma propriedade de uma classe ou método, que espera-se ser preservada. A violação de um contrato sugere erros no código. Por exemplo, o teste mostrado no Código 2.1 representa a violação do contrato *reflexivity of equality*. Esse teste foi gerado pelo Randoop e encontrou um *bug* na API `Collection` da JDK. O teste mostra que as classes de `Collection` podem criar um objeto que não é igual a ele mesmo. Uma `TreeSet` é uma coleção ordenada. De acordo com a API da Sun, uma chamada ao seu construtor

passando uma lista (linha 6), deve lançar uma exceção, já que o elemento lista não é comparável. No entanto, o construtor aceita a lista como parâmetro. Por isso, esse contrato foi violado e a asserção da linha 9 falha em tempo de execução.

Código Fonte 2.1: Teste gerado pelo Randoop para API Collection da JDK

```
1 public static void teste() {
2     LinkedList list = new LinkedList();
3     Object o1 = new Object();
4     list.addFirst(o1);
5     TreeSet t1 = new TreeSet(list);
6     Set s1 = Collections.synchronizedSet(t1);
7     Assert.assertTrue(s1.equals(s1));
8 }
```

Atualmente, Randoop considera os seguintes contratos:

- *Equals to null*: `o.equals(null)` tem que retornar `false`
- *Reflexivity of equality*: `o.equals(o)` tem que retornar `true`
- *Symmetry of equality*: `o1.equals(o2)` implica em `o2.equals(o1)`
- *Equals-hashcode*: Se `o1.equals(o2)==true` então `o1.hashCode() == o2.hashCode()`
- *No null pointer exceptions*: Nenhuma `NullPointerException` é lançada se não for usada alguma entrada com valor `null`

Os testes de regressão gerados pelo Randoop são úteis para serem usados no futuro, depois de uma mudança no código. Se o teste passar após sua geração e vir a falhar depois de uma mudança, então o comportamento do programa pode ter sido modificado pelas alterações realizadas no código.

2.1.5 Teste de Implementação de Refatoramentos

Testar ferramentas de refatoração necessita de entradas complexas, como programas. A saída do teste pode ser o programa refatorado ou a rejeição da transformação quando alguma condição do refatoramento é violada. Criar manualmente casos de testes para ferramentas de refatoração é custoso e difícil, já que desenvolvedores precisam criar programas e oráculos

para avaliar se a transformação preserva o comportamento. Esse processo pode resultar em testes com um baixo nível de cobertura e potencial em detectar faltas. A seguir falaremos de algumas abordagens propostas para testar ferramentas de refatoração utilizando os geradores de programas: ASTGen [13] e JDolly [66]. Detalharemos um pouco mais a técnica que utiliza o JDolly pois vai ser utilizada na avaliação deste trabalho.

ASTGen

Daniel et al. [13] propuseram uma técnica para testar ferramentas de refatoramentos. A técnica é baseada em geração de programas e oráculos para avaliar se um refatoramento está correto. Para cada tipo de refatoramento avaliado, são gerados programas e para cada programa o refatoramento é aplicado pela ferramenta e, utilizando os oráculos propostos, o refatoramento é avaliado. O gerador de programas, ASTGen, permite desenvolvedores escreverem geradores imperativos cuja execução produz milhares de programas, gerados exaustivamente, com propriedades estruturais relevantes para um tipo de refatoramento específico. O ASTGen utiliza a linguagem Java para escrever os geradores.

Para avaliar os refatoramentos o ASTGen implementa 6 oráculos.Entretanto, os oráculos não avaliam a semântica do programa, pois eles não checam se a transformação preserva o comportamento. Os oráculos avaliam a transformação apenas a nível de sintaxe.

JDolly

Soares et al. [66] propuseram outra técnica para testar ferramentas de refatoramentos baseada no JDolly, um gerador automático de programas, e no SafeRefactor [68] para avaliar se a transformação preservou o comportamento. O JDolly gera os programas e para cada programa gerado, a ferramenta aplica o refatoramento e o SafeRefactor avalia se o refatoramento está correto quanto a preservação de comportamento.

JDolly gera automaticamente e exaustivamente programas de acordo com uma especificação do programa, que inclui escopo de elementos Java (pacotes, classes, atributos e métodos) e restrições adicionais. JDolly utiliza Alloy [32], uma linguagem de especificação, como uma infraestrutura formal para gerar programas. Ele contém um subconjunto do metamodelo Java especificado em Alloy. O escopo e as restrições dos programas a serem gerados, são especificados em Alloy pelo usuário. O JDolly utiliza o Alloy Analyzer para

encontrar soluções, que ele traduz em programas, considerando as restrições especificadas pelo usuário. A seguir, daremos uma visão geral de Alloy e de como ela é utilizada pelo JDolly.

Alloy

Alloy é baseada em uma lógica que combina quantificadores de lógica de primeira ordem com operadores do cálculo relacional. Um modelo ou especificação em Alloy é uma sequência de *parágrafos* de dois tipos: assinaturas e restrições. Cada *assinatura* denota um conjunto de objetos associados a outros objetos por relações declaradas nas assinaturas. Uma assinatura introduz um tipo e uma coleção de *relações*, juntamente com seus tipos e outras restrições de seus valores incluídos.

O Código 2.2 mostra parte do modelo de Alloy usado pelo JDolly para especificar os conceitos de classes, métodos e atributos de Java. Uma classe Java pode declarar um conjunto de atributos e métodos, e pode estender outra classe. O qualificador *set* nas relações *fields* e *methods* não impõe nenhuma restrição na multiplicidade (>=0). O qualificador *lone* denota uma relação parcial (0 ou 1). Em Alloy, uma assinatura pode estender outra assinatura. Neste caso, a assinatura filha (*subsignature*) é um subconjunto da assinatura pai. No exemplo, *Class* é subassinatura de *Type*.

Código Fonte 2.2: Um subconjunto do metamodelo de Java especificado em Alloy.

```
1 sig Type { ... }
2 sig Class extends Type {
3   extend: lone Class,
4   fields: set Field,
5   methods: set Method, ...
6 }
7 sig Field { ... }
8 sig Method { ... }
```

A palavra-chave *all* representa o quantificador universal, e *in* denota o operador associação do conjunto no fragmento anterior. Os operadores ^ e *!* representam o fechamento transitivo e operador de negação, respectivamente. O quantificador existencial é representado por *some* e o *no* quando aplicado a uma expressão, indica que a expressão é

vazia. O operador (.) é uma definição generalizada do operador de junção relacional. Por exemplo, a expressão *c.extend* retorna a superclasse de *c*.

Em Alloy, as restrições de um modelo são organizadas em parágrafos: fatos (*fact*), predicados (*pred*), funções (*fun*) e asserções (*assert*). Fatos são blocos cujas fórmulas sempre serão satisfeitas. Por exemplo, no metamodelo de Java existe um fato que especifica a restrição de que uma classe não pode estender-se. Funções são expressões chamadas com zero ou mais parâmetros e a declaração de uma expressão de retorno. Predicados são fórmulas chamadas com zero ou mais parâmetros. Por fim, asserções são implicações a serem verificadas. Elas se destinam a serem seguidas a partir dos fatos do modelo.

2.2 Análise de Impacto da Mudança

Nesta seção, vamos mostrar alguns fundamentos teóricos na área de análise de impacto da mudança, utilizados na realização deste trabalho. Vamos iniciar a seção apresentando alguns conceitos básicos de análise de impacto. Em seguida, descrevemos os tipos de análises de código: estática e dinâmica, e discutimos vantagens e desvantagens de cada uma. Falamos também sobre as principais estratégias da análise de impacto. Por fim, falamos da análise de impacto para programas orientados a objetos. Este último é o foco deste trabalho.

2.2.1 Definição

Várias definições são encontradas na literatura para análise de impacto. De acordo com Bohner e Arnold [6] análise de impacto é a avaliação do efeito de uma mudança. Turver and Munro [76] definem análise de impacto como a avaliação de uma mudança no código-fonte de um módulo, sobre os outros módulos do sistema. Queille et al. [54] tem como definição de análise de impacto uma técnica aproximada que deve focar na minimização do custo de detectar efeitos secundários indesejados. Lindvall [45] define análise de impacto como a identificação do conjunto de entidades do software que precisam ser modificadas para implementar um novo requisito em um sistema já existente. Já Jönsson e högskola [33] afirmam que análise de impacto é uma ferramenta para controlar mudanças e assim, evitar

deterioração do sistema.

Várias outras definições podem ser encontradas na literatura. Cada uma depende do contexto que está sendo empregada a análise de impacto. Por exemplo, algumas abordagens analisam o código antes da mudança ter sido implementada. Outras avaliam o impacto da mudança após ela ter sido realizada. Além disso, a análise de impacto desejada pode ser à nível de entidades do código ou de módulos do sistema. No contexto deste trabalho, consideramos análise de impacto da mudança como uma técnica para identificar partes do programa que possam ter mudado de comportamento após uma mudança. Os resultados desta análise podem ser usados para orientar engenheiros de testes para determinar como alocar seus esforços de testes de regressão no sistema modificado ou para guiar a implementação ou geração de novos testes.

2.2.2 Abordagens: Estática e Dinâmica

Atualmente, existem dois tipos de análise de impacto: estática e dinâmica. A análise estática analisa o código fonte, em tempo de compilação, para identificar os possíveis impactos. Já a análise dinâmica, analisa rastros de execuções no programa, para identificar as entidades impactadas.

Análise Estática

Na análise estática, o programa não é executado e a análise é feita no código do programa. Ela pode ser realizada antes da mudança ter sido feita para prever o impacto que a mudança pode causar no código. Para isso, é necessário informar detalhes da mudança proposta. Ela também pode ser feita após a mudança. Neste caso, é necessário avaliar o código antes e depois da mudança. A análise estática explora todos os caminhos de execução possíveis. É considerada uma análise conservadora, pois vai identificar todas as partes do programa que possam ter sido impactadas [33].

Código Fonte 2.3: Métodos de um programa

```
1  class A {
2    metodo1(x) {
3      if (x == 10)
4        metodo2()
5      else
6        metodo3()
7    }
8    metodo2() {
9      metodo4()
10   }
11   metodo3() {
12     metodo5()
13   }
14   metodo4() {}
15   metodo5() {}
16 }
```

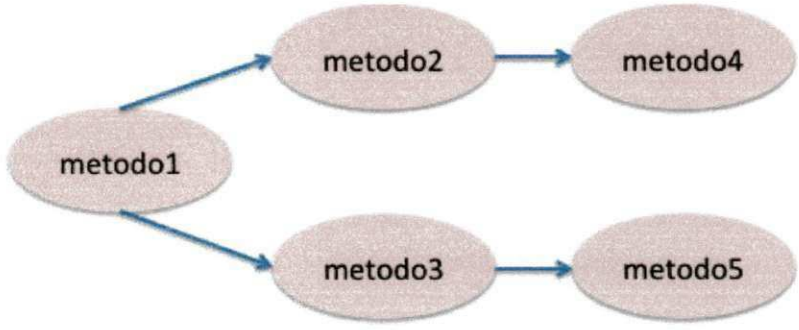


Figura 2.2: Grafo de dependência dos métodos do programa 2.3

Observe o Código 2.3. Ele mostra um programa que contém uma classe com 5 métodos. O método *metodo*1 chama o método *metodo*2 se o seu parâmetro x for igual a 10, caso contrário, chama *metodo*3. O *metodo*2 chama *metodo*4 e o *metodo*3 chama *metodo*5. Um método depende de quem ele chama. O grafo de dependência desse programa está ilustrado na Figura 2.2. A partir desse grafo, montamos a Tabela 3.3. Um método modificado vai afetar todos os métodos que dependem dele. Por exemplo, *metodo*4 pode afetar *metodo*1 e *metodo*2, pois esses métodos dependem do *metodo*4 (exercitam direta ou indiretamente).

Método Modificado	Métodos Afetados
metodo1	-
metodo2	metodo1
metodo3	metodo1
metodo4	metodo1, metodo2
metodo5	metodo1, metodo3

Tabela 2.1: Métodos afetados por cada método do programa 2.3, considerando todas as possíveis execuções do programa

Essa tabela pode ser o resultado de uma análise estática realizada nesse programa. Veja que para qualquer execução desse programa, os possíveis métodos afetados por um método modificado estão na tabela.

Em resumo, as vantagens e desvantagens da análise de impacto estática:

- **Vantagens**: análise mais conservadora, é capaz de identificar todos os possíveis impactos.
- **Desvantagens**: pode retornar um conjunto impactado muito grande, pois ela pode identificar entidades impactadas quando na verdade não são. Dependendo do tipo de análise, ela pode ser muito custosa em programas grandes e com um alto nível de acoplamento.

Análise Dinâmica

A análise dinâmica é realizada a partir de execuções no programa. Essa análise coleta os rastros das execuções para analisar o impacto. Seu resultado é preciso, mas depende das entradas fornecidas, pois o resultado pode não ser generalizado para todas as execuções possíveis de um programa. Não existe garantia que a coleção de testes do programa, que está sendo executada, é característica de todas as possíveis execuções. Por isso, a análise de impacto dinâmica não promete identificar todos os possíveis impactos.

Observe novamente o programa 5.4. A Tabela 3.3 foi construída a partir de uma análise estática no código. Note que ela não depende de valores de entrada, pois como vimos, a

análise estática generaliza os resultados para qualquer execução. Na análise dinâmica, precisamos de testes para executar o programa. Suponha que os testes desse programa sejam os que estão apresentados no Código 2.4. Se o *metodo*5 for modificado, a análise de impacto dinâmica consegue identificar os métodos afetados: *metodo*1 e *metodo*3, pois as chamadas das linhas 4 a 6, saem de *metodo*1, passam por *metodo*3 e chegam no *metodo*5. No entanto, se o *metodo*4 for modificado, não será possível identificar os métodos afetados por ele: *metodo*1 e *metodo*2, pois não existe nos testes alguma execução partindo desses métodos que chame *metodo*4. Neste último exemplo, a análise não foi bem sucedida porque o programa não possui casos de testes que cubram todos os possíveis caminhos do programa. Na prática, principalmente em programas grandes, é difícil ter uma coleção de testes que tenha todas as execuções de métodos possíveis.

Código Fonte 2.4: Coleção de testes do programa 5.4

```
1  class Teste {
2    teste1() {
3      A a = new A();
4      a.metodo1(-1);
5      a.metodo1(30);
6      a.metodo1(-20);
7    }
8    teste2() {
9      A a = new A();
10     a.metodo3();
11   }
12   teste3() {
13     A a = new A();
14     a.metodo4();
15   }
16 }
```

Em resumo, as vantagens e desvantagens da análise de impacto dinâmica:

- **Vantagens**: prever o impacto em relação a execuções de programas específicos ou perfis operacionais, que podem ser úteis para muitas tarefas de manutenção. Análise precisa. Dificilmente identifica uma entidade impactada que não seja realmente impactada.

- **Desvantagens**: sacrifica a corretude da avaliação do impacto. Pode deixar de identificar entidades impactadas. Depende da cobertura da mudança das execuções.

2.2.3 Estado da Arte

Um dos primeiros trabalhos de análise de impacto que consta na literatura estuda as conexões entre módulos do sistema [26]. A técnica baseia-se na ideia de que para cada par de módulos de um sistema, existe uma probabilidade de que uma mudança em um módulo implique em uma mudança no outro módulo. O objetivo do trabalho é modelar propagação de mudanças entre os artefatos do sistema, incluindo os requisitos. Posteriormente, foram surgindo as outras estratégias de análise de impacto. A seguir, descrevemos algumas estratégias existentes de análise de impacto e em seguida, falaremos especificamente de análise de impacto para programas orientados a objetos, que é o foco do deste trabalho.

- **Análise de Rastreabilidade**

  A análise de rastreabilidade é a análise de relações que foram identificadas durante o desenvolvimento entre todos os tipos de artefatos. Ela é adequada para a análise de relações entre os requisitos, componentes de arquitetura, documentação e outros artefatos do programa [41; 24]. Por exemplo, a rastreabilidade feita entre requisitos e componentes do sistema, permite realizar predições no efeito de mudanças de requisitos. A partir de uma mudança, é possível identificar os componentes que foram afetados. Da mesma forma, também devem ser avaliados os requisitos que possuem alguma ligação com o requisito afetado.

- **Análise de Dependência**

  A análise de dependência [40; 44; 36] fornece uma avaliação detalhada de baixo nível, das dependências entre entidades do programa, como por exemplo variáveis ou funções. Essas relações são extraídas do código-fonte e a análise não pode ser utilizada para outros artefatos do sistema. Ela considera dependência de dados e dependência de controle. Na dependência de dados, são avaliadas as relações entre instruções do programa que utilizam dados em comum do sistema. Por exemplo, quando uma instrução fornece um valor direta ou indiretamente usado por outra instrução do programa,

é identificado um relacionamento entre essas instruções. A análise de dependência de controle identifica os relacionamentos entre instruções do programa que controlam a execução do sistema.

- **Grafos de Chamadas**

  Algumas técnicas de análise de impacto utilizam grafos de chamadas para realizar a análise do código [60; 56; 71; 4; 80]. Alterações em métodos de um programa podem afetar outras entidades do código que chamam o método direta ou indiretamente. Analisar o comportamento de chamadas de um sistema pode ajudar a analisar o impacto de uma mudança em um método do programa. Então, um grafo de chamadas armazena chamadas de métodos que são extraídas a partir de uma análise do código fonte. Esse grafo permite que seja estimada a propagação de uma mudança.

- **Fatiamento de Programas**

  Fatiamento de programas [73; 31] é uma técnica de análise de código que baseia-se em computar um conjunto de instruções de um programa (fatia), que possam afetar valores em algum ponto de interesse, referido como critério de corte. O fatiamento de código pode ser estático ou dinâmico.

- **Modelos Probabilísticos**

  Modelos probabilísticos [75; 81] permitem modelar a propagação de mudanças baseado na exploração de modelos e teoremas matemáticos. Ele permite computar a probabilidade de uma entidade ser impactada pela mudança.

Lehnert [42] realizou um estudo sobre as estratégias de análise de impacto existentes e fez um quadro comparativo entre várias abordagens de diferentes estratégias. A revisão também identificou uma séria de questões em aberto para futuras investigações. Primeiramente, foi revelado que faltam validações empíricas das abordagens propostas. Em segundo lugar, faltam abordagens que abrangem todos os estágios do processo de desenvolvimento do software. Por fim, não existe uma classificação uniforme, entre as abordagens, dos tipos de mudanças e dependências.

Análise de Impacto em Programas Orientados a Objetos

As primeiras técnicas de análise de impacto foram propostas para programas estruturados. A maioria das técnicas de análise de impacto para esses programas são baseadas na análise de relações de dependência entre controle, dados e componentes de um programa. As principais linhas de pesquisa são análise de fluxo de dados [27] e fatiamento de programas [29; 35].

Em programas orientados a objetos, características como o polimorfismo, a herança e o encapsulamento contribuem diretamente para tornar a tarefa da análise de impacto mais complexa [44]. Além disso, uma transformação local pode impactar diversas partes do programa, devido a dependências e inter-relacionamentos entre as classes. Por exemplo, adicionar um método pode ter um impacto em algumas subclasses da hierarquia. Considere a mudança ilustrada na Figura 2.3. É adicionado o método $foo()$ na classe B. Veja que esse método adicionado sobrescreve o método $A.foo()$. Então, antes da mudança as classes B e C usavam o método $A.foo()$. Inclusive, o método $C.bar()$ chamava $A.foo()$ retornando 1. Após a adição do método $B.foo()$, ele passa a ser chamado por $C.bar()$ que vai mudar de comportamento e retornar 2. Observe que essa mudança pontual causou um impacto em várias classes da hierarquia. Além disso, outras partes do programa também foram impactadas. A classe D é afetada, pois o método $D.m()$ chama o método $C.foo()$ que é impactado pela mudança.

Algumas abordagens foram propostas para identificar o impacto de uma mudança em códigos de linguagens orientadas a objetos. Lee et al. [40] desenvolveram uma técnica algorítmica para calcular o impacto de uma mudança no programa. O software é analisado e as informações são salvas em grafos de fluxo de dados e fluxo de controle. A partir desses grafos, são aplicados algoritmos para recuperar as suas informações e calcular o fecho transitivo do impacto das mudanças propostas. Os resultados do cálculo são apresentados, tanto a nível de classe, quanto a nível de membros da classe. Briand et al. [9] propuseram a análise de impacto baseada em modelos UML do tipo diagrama de classes, sequência ou estados. A abordagem avalia os elementos do modelo que são diretamente ou indiretamente afetados pela mudança utilizando regras de impacto formalmente definidas na análise.

Kung et al. [36] propuseram um método para identificar classes impactadas devido a mudanças estruturais em classes de biblioteca de linguagens orientadas a objetos. O método é baseado em uma abordagem de engenharia reversa que extrai da biblioteca informações

Figura 2.3: Adicionar método afeta diversas partes do programa.

Código Fonte 2.5: Programa Original

```
1  class A {
2    int foo() {
3      return 1;
4    }
5  }
6  class B extends A {}
7
8
9
10
11 class C extends B {
12   int bar() {
13     foo();
14   }
15 }
16 class D {
17   int m() {
18     new C().bar();
19   }
20 }
```

Código Fonte 2.6: Programa Modificado

```
1  class A {
2    int foo() {
3      return 1;
4    }
5  }
6  class B extends A {
7    int foo() {
8      return 2;
9    }
10 }
11 class C extends B {
12   int bar() {
13     foo();
14   }
15 }
16 class D {
17   int m() {
18     new C().bar();
19   }
20 }
```

de classes e seus relacionamentos. Essas informações são representadas em grafos utilizados para automaticamente identificar as mudanças e seus efeitos. Li e Offut [44] realizaram um estudo para examinar como mudanças feitas em programas orientados a objetos podem afetar classes do programa. Eles propuseram um algoritmo que calcula o fechamento transitivo do grafo de dependência do programa. O cálculo do efeito da mudança no programa inclui a identificação de efeitos dentro de uma classe, a identificação dos efeitos entre classes relacionadas e a identificação dos efeitos causados por relações de herança entre classes.

Law et al. [38] apresentaram uma nova técnica de análise de impacto, PathImpact, que é baseada em particionamentos estáticos e dinâmicos e algoritmos recursivos de grafos de chamadas. Lulu Huang e Yeong-Tae Song [30] propuseram uma abordagem de análise de impacto que define o impacto de uma mudança a partir da análise de rastros de execução que identificam dependências entre elementos do programa. A abordagem identifica atributos e métodos impactados pela mudança.

Lile Hattori [28] propôs 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. Ela combina 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.

Ryder e Tip [60] propuseram uma abordagem para identificar os testes de regressão que foram afetados por um conjunto de mudanças feitas no programa e as mudanças que afetaram cada teste. Além disso, identifica as mudanças que podem ser incorporadas ao programa com segurança. A abordagem decompõe uma transformação em transformações atômicas previamente definidas. Então, são construídos grafos de chamadas para as mudanças e o impacto de cada mudança atômica é calculado. Posteriormente, Ren et al. [56] implementaram Chianti, a ferramenta para analisar o impacto da mudança de códigos Java, baseada na técnica anterior. Stoerzer et al. [71] propuseram um classificador de mudanças de acordo com a probabilidade de cada uma ter causado a falha de um teste. O Chianti é utilizado para fazer a análise de impacto. As mudanças são classificadas em: vermelho, verde ou amarelo. As vermelhas são mais prováveis de terem causado a falha de um teste enquanto as verdes são menos prováveis. Badri et al. [4] apresentam uma nova técnica de análise de impacto da

mudança baseado em um modelo de grafos de controle de chamadas. A técnica é estática e examina no nível de chamada de métodos, capturando o controle relacionado com chamadas entre os componentes do programa.

Crisp, uma extensão do Chianti foi proposta por Chesley et al. [11]. A partir de uma modificação no programa, a ferramenta utiliza o Chianti para realizar a análise de impacto da mudança. Se algum teste falhar no código modificado, o desenvolvedor utiliza Crisp para ajudar na atividade de depuração do código. Crisp cria um programa intermediário juntando o programa original e o conjunto de mudanças que afetaram um determinado teste. Ou seja, permite que o desenvolvedor faça uma depuração mais rápida, focando apenas nas mudanças que afetaram um teste que falhou. Wloka et al. [78] introduziram um novo conceito ao JUnit: barra amarela na ferramenta JUnitMX. A barra amarela indica que todos os testes estão passando mas a mudança realizada no código não está sendo totalmente coberta pelos testes. Ou seja, com a ferramenta é possível identificar se os testes estão exercitando todas as entidades impactadas por uma mudança. A ferramenta utiliza o Chianti para realizar a análise de impacto de mudança.

Zhang et al. [80] reimplementaram o Chianti adicionando algumas melhorias: adicionaram uma nova mudança atômica e refinaram as dependências entre as mudanças atômicas acrescentando novas regras. Eles propuseram o FautTracer que além realizar a análise de impacto, ordena as mudanças que causaram a falha de um determinado teste utilizando técnicas de localização de faltas baseadas em espectro. Os autores comparam a análise de impacto do Chianti com a do FaultTracer e dizem melhorar a seleção dos testes afetados e a detecção das mudanças atômicas que afetam cada teste.

2.3 Refatoramento

A medida que o sofware se torna mais complexo devido às mudanças realizadas durante seu ciclo de vida, como por exemplo, adição de novas funcionalidades ou correção de defeitos, é necessário reestruturá-lo para melhorar o entendimento do sistema e tornar mais fácil sua manutenção. Reestruturar é transformar uma forma de representação para outra no mesmo nível de abstração, preservando o comportamento externo do sistema [12]. Refatoramento é basicamente uma variação de reestruturação para sistemas orientados a objetos.

2.3.1 Definição

O termo refatoramento foi originalmente introduzido por Opdyke em sua tese de doutorado [51]. Fowler [20] define refatoramento como o processo de modificar um sistema de software, de modo que melhore sua estrutura interna, preservando seu comportamento observável. Sua essência está em uma série de pequenas transformações que preservam comportamento. Cada transformação (chamada de refatoração) é uma mudança pequena, mas uma sequência de pequenas transformações produz uma reestruturação significativa.

Segundo Mens e Tourwé [47], o processo de refatorar um programa consiste nas seguintes atividades:

1. Identificar a parte do programa que precisa ser refatorada;
2. Determinar os refatoramentos que vão ser aplicados para as partes identificadas;
3. Garantir que o refatoramento preserva o comportamento do programa;
4. Aplicar o refatoramento;
5. Avaliar o efeito do refatoramento na qualidade do software;
6. Manter a consistência entre o código do programa refatorado e outros artefatos do sistema;

Para identificar estruturas do código que sugerem possibilidade de refatoração, Fowler [20], introduziu o termo *bad smells*. Como o próprio nome diz, um "mau cheiro", indica que alguma parte do código não está boa e precisa ser melhorada. Após algum *bad smell* ter sido identificado, deve-se escolher o refatoramento adequado para melhorar essa parte do código. Alguns exemplos de *bad smells* são: código duplicado, método longo, classe grande e longa lista de parâmetros. A seguir, na Seção 2.3.1 mostramos exemplos de aplicações de refatoramentos em códigos com *bad smells*.

Exemplos

Nessa seção, exibimos dois exemplos de refatoramentos de programa. Primeiramente, identificamos o *bad smell* no código e a parte do programa que deve ser refatorada. Em seguida, descrevemos a escolha e aplicação do refatoramento e mostramos o código refatorado.

- Exemplo *Extract Method*

  Para este exemplo, observe o Código 2.7. Ele contém o método *imprimeDivida*() que imprime a dívida de um cliente. Primeiramente, ele imprime o cabeçalho com dados do cliente. Depois, ele calcula a dívida, somando o valor de cada pedido do cliente. Por fim, ele imprime os pedidos e o valor total da dívida. Podemos ver um *bad smell* nesse método: ele precisa de vários comentários para ser entendido. É adequado aplicar o refatoramento *Extract Method*, que extrai trechos do método e cria novos métodos para os trechos extraídos. Dessa forma, melhora as chances de reutilização do código e faz com que os métodos que o chamam fiquem mais fáceis de entender. O Código 2.8 mostra o método *imprimeDivida*() refatorado. Foram extraídos três métodos: *imprimeCabecalho*(), *calculaDivida*() e *imprimeDetalhes*(*doubledivida*). Após o refatoramento, o método *imprimeDivida*() está mais fácil de entender.

Código Fonte 2.7: Programa com *bad smell*: método longo

```
1  void imprimeDivida() {
2    List<Pedido> pedidos = cliente.getPedidos();
3    double divida = 0.0;
4    // imprime cabeçalho
5    System.out.println ("*** Dívidas ****");
6    System.out.println ("Nome:  "+cliente.nome);
7    System.out.println ("Endereço: "+cliente.endereco);
8    // calcula dívida
9    for (Pedido pedido: pedidos) {
10     divida += pedido.valor ();
11   }
12   // imprime detalhes
13   System.out.println ("Pedidos");
14   imprimePedidos(pedidos);
15   System.out.println ("Valor da dívida : " + divida);
16 }
```

Código Fonte 2.8: Médodo *imprimeDivida*() do Programa 2.7 refatorado

```
1  void imprimeDivida () {
2    imprimeCabecalho();
3    double divida = calculaDivida();
4    imprimeDetalhes(divida);
5  }
6  void imprimeCabecalho() {
7    System.out.println ("*** Dívidas ****");
8    System.out.println ("Nome:  "+cliente.nome);
9    System.out.println ("Endereço: "+cliente.endereco);
10 }
11 double calculaDivida() {
12   List<Pedido> pedidos = cliente.getPedidos();
13   double divida = 0.0;
14   for (Pedido pedido: pedidos) {
15     divida += pedido.valor ();
16   }
17   return divida;
18 }
19 void imprimeDetalhes(double divida) {
20   System.out.println ("Pedidos");
21   imprimePedidos(pedidos);
22   System.out.println ("Valor da dívida : " + divida);
23 }
```

- Exemplo *Pull Up Field*

  Veja neste exemplo, os diagramas da Figura 2.4. O diagrama da esquerda representa um programa com três classes: A, B e C, onde B e C são subclasses de A. Podemos ver um pequeno *bad smell*: código repetido. As classes B e C contém o mesmo atributo f. Como elas têm a superclasse A em comum, esse atributo pode ser passado para superclasse, evitando essa repetição de código. O diagrama da direita representa o programa após o refatoramento *pull up field* ter sido aplicado. Esse tipo de refatoramento move um atributo de uma classe para sua superclasse.

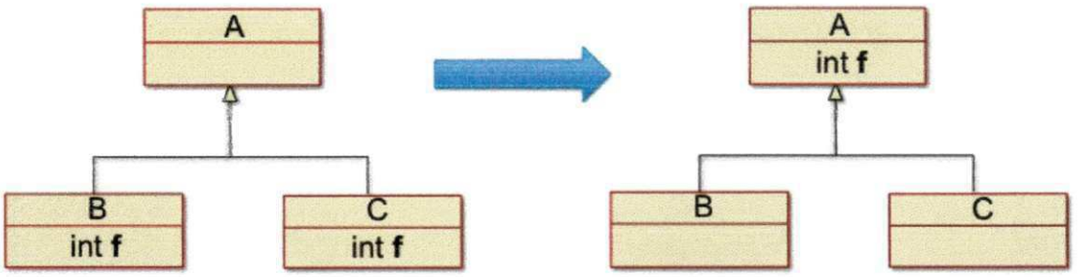


Figura 2.4: Refatoramento *PullUpField*: move atributos de uma classe para sua super classe.

2.3.2 Problema

Para cada refatoramento, existe um conjunto de precondições que garante a preservação do comportamento do programa. Por exemplo, para renomear um método no programa, antes precisa ser verificado se já existe outro método com a mesma assinatura do novo nome desejado para evitar um erro de compilação. Fowler [20] propôs aplicar refatoramentos através de pequenos passos intercalados por compilação e testes para garantir a preservação do comportamento do programa. Para ajudar os desenvolvedores nessa atividade, Don Roberts [57] introduziu a primeira ferramenta de refatoramento. Ela automatiza o processo de checar as pre-condições e aplicar a transformação. Atualmente, a maioria dos Ambientes de Desenvolvimento Integrados (IDE) utilizados no desenvolvimento de sistemas, como Eclipse [16], Netbeans [49], JBuilder [72], possuem um módulo para automatizar refatoramentos. A seguir, elencamos alguns problemas em atividades de refatoramentos:

- **Problema 1: Ferramentas podem aplicar refatoramentos que não preservam o comportamento**

  Uma maneira de garantir que uma transformação preserva o comportamento é ter um conjunto de precondições, provadas formalmente com base na semântica da linguagem. Entretanto, provar refatoramentos com respeito a uma semântica formal foi proposto como um desafio [62]. Como isso, as ferramentas de refatoramentos implementam as transformações utilizando o senso comum sem um embasamento formal. Logo, as ferramentas podem aplicar transformações que não preservam o comportamento do programa. Por exemplo, considere a transformação da Figura 2.5, aplicada pelo Eclipse [16], que não preserva o comportamento. A transformação move o atri-

buto k da classe C para a classe B. O método $teste()$ tem uma chamada $super.k$. No programa original, o atributo k da superclasse mais próxima de C é $A.k$, então ele vai ser chamado pelo método $teste()$, retornado 10. Já no programa modificado, o k chamado pelo método $teste()$ vai ser $B.k$, que está definido na superclasse mais próxima de C, retornado 20. Logo, é um *bug* da ferramenta.

Código Fonte 2.9: Programa Original

```
1  public class A {
2      public int k = 10;
3  }
4
5  public class B extends A {
6  }
7
8  public class C extends B {
9      public int k = 20;
10     public int teste() {
11         return super.k;
12     }
13 }
```

Código Fonte 2.10: Programa Modificado

```
1  public class A {
2      public int k = 10;
3  }
4
5  public class B extends A {
6      public int k = 20;
7  }
8
9  public class C extends B {
10     public int teste() {
11         return super.k;
12     }
13 }
```

Figura 2.5: *PullUpField* aplicado pelo Eclipse que não preserva o comportamento.

Renomear entidades como classes, métodos e atributos de um programa são tipos de refatoramentos bastante utilizados pelos desenvolvedores. Renomear manualmente um nome acessível por todo programa é bastante complicado e sujeito a erros. Por isso, desenvolvedores utilizam ferramentas de refatoramentos para ajudar nesta tarefa. Entretanto, para pesquisas complexas por um nome no programa, o conjunto de condições podem não ser suficiente. Além disso, incrementos na linguagem requerem novas condições. Ferramentas de refatoramento podem renomear entidades cujos nomes não correspondem à mesma declaração antes e após o refatoramento, ou seja, permite transformações de renomeação que não preservam o comportamento.

Em linguagens de programação orientadas a objetos, como Java, são utilizados modificadores de acesso, como *public* e *private*, para esconder ou tornar pública alguma informação. Ao aplicarem-se alguns tipos de refatoramentos, podem ser necessárias

algumas alterações em modificadores de acesso para que a transformação preserve o comportamento. Quando a transformação causa acessibilidade insuficiente, o compilador relata o problema. Entretanto, quando a acessibilidade é excessiva e causa problemas como, por exemplo, *overloading*, levando a mudança no comportamento, o problema é mais difícil de ser identificado. Várias IDEs que suportam refatoramentos, como Eclipse e NetBeans, falham ao aplicar transformações que necessitam uma mudança de acessibilidade, pois não sendo feita a mudança necessária, a transformação não preserva o comportamento.

- **Problema 2: Testes de regressão podem não ser adequados para testar refatoramentos**

  Na prática, desenvolvedores utilizam testes de regressão para garantir que o refatoramento preservou o comportamento do programa. Entretanto, esses testes podem não ser adequados pelos seguintes motivos:

  - **Testes podem ser modificados pelo refatoramento**

    Ao aplicar um refatoramento, a transformação pode afetar alguns métodos do programa. Por exemplo, considere a aplicação de um refatoramento que renomeia um método do programa. Se este método é chamado por algum teste, o refatoramento vai modificar o teste para chamar o método com o novo nome. Nessa situação, se a transformação não estiver correta, vai inserir problemas na coleção de testes, incapacitando-a de detectar a mudança comportamental.

  - **Coleção de testes pode ser grande e consumir muito tempo na execução dos testes**

    Em programas grandes, a coleção de testes de regressão pode ser grande também. Desta forma, executar todos os testes a cada refatoramento pode ser custoso. Elbaum et al. [17] afirmam que um colaborador industrial reportou que para um de seus produtos de aproximadamente 20.000 linhas de código, a coleção de testes requer sete semanas para ser completamente executada.

  - **Os testes podem não exercitar o refatoramento**

    Os testes de regressão são construídos para testar o programa a cada mudança ou *release* do projeto. Como eles não focam no refatoramento aplicado, é possível

que este não seja exercitado pela coleção de testes. Em um estudo realizado por Rachatasumrit e Kim [55], em três grandes projetos *open source*, mostrou que refatoramentos não estão sendo bem testados. Apenas 22% dos métodos e atributos refatorados são cobertos pelos testes de regressão. Eles concluem que os projetos tem testes de regressão insuficientes para avaliar a corretude dos refatoramentos.

2.3.3 Estado da Arte

Opdyke [51] propôs um conjunto de precondições para garantir que alguns tipos de refatoramentos preservassem o comportamento. Entretanto, ele não provou formalmente as precondições. Posteriormente, Tokuda and Batory [74] mostraram que as precondições propostas por Opdyke não garantiam preservação de comportamento. Provar refatoramentos com respeito a uma semântica formal foi proposto como um desafio [62].

Outras técnicas foram propostas para melhorar atividades de refatoramentos. Em sua tese de doutorado, Gheyi [59] propôs um conjunto de transformações que preservam o comportamento, em Alloy. Além disso, codificou um sistema de tipos e especificou regras de boa formação e uma semântica estendida para Alloy [32] em PVS [52], que engloba uma linguagem de especificação formal e um provador de teoremas [22]. Borba et al. [7] propuseram um conjunto de refatoramentos para um subconjunto de Java com a semântica de cópia (ROOL). Eles provaram os refatoramentos com relação a uma semântica formal. Silva et al. [65] definiram e provaram formalmente um conjunto de leis para que algumas transformações preservassem o comportamento, em uma linguagem OO sequencial, com respeito a semântica rCOS.

Dig e Johnson [15] analisaram mudanças em três frameworks e uma biblioteca largamente utilizados. Os resultados da avaliação do trabalho mostraram que refatoramentos foram a causa de mais de 80% das mudanças feitas nas APIs que provocam incompatibilidade com os clientes. Fuhrer et al. [21] propuseram um refatoramento que ajuda os programadores com a adoção de uma versão genérica de uma biblioteca de classes existente. Eles implementaram esse refatoramento no Eclipse [16] e avaliaram o trabalho com a migração de um número de aplicações Java de tamanho moderado, do framework *Java Collections* para *generic Collection* de Java 1.5.

Schafer et al. [63] definiram uma nova abordagem para renomear entidades em um programa que utiliza a técnica de criar nomes simbólicos para garantir que os nomes renomeados correspondem a entidade desejada. Posteriormente, Schafer et al. [61] propuseram uma ferramenta JastAdd Refactoring Tools (JRRT) que formaliza algumas precondições de dezessete refatoramentos, incluindo a implementação do *Rename*, para Java e implementou na ferramenta. Os programas em Java são traduzidos para outra linguagem para tornar mais fácil avaliar se houve preservação de comportamento. O objetivo foi aumentar a corretude das implementações. Steimann e Thies [70] definiram um conjunto de restrições relacionadas a visibilidade que alguns refatoramentos devem satisfazer para preservar o comportamento. Entretanto, eles não formalizaram todas as condições necessárias para preservar o comportamento. Essas implementações foram propostas para melhorar a corretude de alguns refatoramentos. Entretanto, as precondições não foram provadas formalmente e funcionam apenas para um conjunto restrito de transformações.

Como discutido anteriormente, definir formalmente todas as precondições necessárias para cada tipo de refatoramento não é uma tarefa fácil. Com isso, as ferramentas de refatoramentos podem aplicar transformações que não preservam o comportamento. Soares et al. [68] propuseram o SafeRefactor, uma ferramenta que analisa uma transformação qualquer e gera testes para detectar mudanças comportamentais. A análise do SafeRefactor identifica todos os métodos em comum nas duas versões do programa e gera uma coleção de testes para eles usando o Randoop [53].

Como ferramentas de refatoramentos podem conter bugs, elas precisam ser testadas. Entretanto, testar essas ferramentas não é fácil, já que exigem entradas complexas (programas) e um oráculo para avaliar se a transformação preserva o comportamento. Uma técnica para testar ferramentas de refatoramentos foi proposta por Daniel et al. [13]. A técnica é baseada em geração de programas e oráculos para avaliar se um refatoramento está correto. Para cada tipo de refatoramento avaliado, são gerados programas e para cada programa o refatoramento é aplicado pela ferramenta e, utilizando os oráculos propostos, o refatoramento é avaliado. O gerador de programas, ASTGen, permite desenvolvedores escreverem geradores imperativos cuja execução produz milhares de programas, gerados exaustivamente, com propriedades estruturais relevantes para um tipo de refatoramento específico. O ASTGen utiliza a linguagem Java para escrever os geradores. Eles implementaram 6 oráculos para avaliar os

refatoramentos. Esses oráculos não avaliam a semântica do programa, pois eles não checam se a transformação preserva o comportamento. Os oráculos avaliam a transformação apenas à nível de sintaxe.

Soares et al. [69] introduziram outra técnica para testar ferramentas de refatoramentos. A técnica é baseada no JDolly 2.1.5 e no SafeRefactor para avaliar o comportamento. O JDolly gera automaticamente e exaustivamente programas de acordo com uma especificação para o refatoramento a ser testado, que inclui escopo de elementos Java (pacotes, classes, atributos e métodos) e restrições adicionais. Para cada programa gerado, a ferramenta aplica o refatoramento e o SafeRefactor avalia se a transformação preservou o comportamento do programa.

Capítulo 3

Safira: Um analisador de impacto da mudança

Análise de impacto da mudança é uma atividade que identifica partes no código que possam ter mudado de comportamento após uma mudança feita no programa. Os resultados desta análise podem ser usados para orientar engenheiros de testes para determinar como alocar seus esforços de testes de regressão no sistema modificado ou para guiar a implementação ou geração de novos casos de testes.

Neste capítulo apresentamos Safira, um analisador de impacto de mudança para programas Java. Safira realiza uma análise estática do código para identificar mudanças feitas no programa e relacionamentos entre as entidades do sistema. Como estamos lidando com programas orientados a objetos, consideramos que entidades de um sistema são classes, métodos e atributos. Utilizamos as características inerentes desse tipo de programas para determinar a relação entre as entidades.

Safira analisa duas versões de um programa e identifica os métodos que podem ter mudado de comportamento devido à transformação. A abordagem de Safira consiste em 3 passos sequenciais: decompor a transformação em subtransformações, identificar o conjunto de métodos impactados, ou seja, os métodos impactados por cada subtransformação e identificar os métodos que exercitam direta ou indiretamente o conjunto impactado em qualquer nível de indireção.

Iniciamos este capítulo com um exemplo de uma transformação que muda o comportamento de alguns métodos do programa (Seção 3.1). Na Seção 3.2 descrevemos em detalhes

cada passo da nossa abordagem. Em seguida, na Seção 3.3, mostramos como é feita a cobertura da mudança, utilizada como critério de adequação de testes neste trabalho. Descrevemos Safira, a ferramenta que implementamos a abordagem proposta, na Seção 3.4. Por fim, na Seção 3.5, descrevemos algumas abordagens similares e realizamos um estudo comparativo entre elas e Safira.

3.1 Exemplo Motivante

Nesta seção mostramos uma transformação que muda o comportamento de alguns métodos do programa. Considere o programa original e o programa modificado pela transformação ilustrados na Figura 3.1. A transformação move o método m da classe B para classe C. Ela não preserva o comportamento do programa, pois o valor retornado pelo método $teste$ foi alterado de 1, no programa original, para 2 no programa modificado. O método m possui uma expressão de invocação de método utilizando o $super$ para acessar o método da superclasse. A mudança comportamental ocorre porque no programa original, como m está declarado em B e sua superclasse é A, então ele chama $A.k()$. Já no programa modificado, m está declarado em C e como sua superclasse direta é B, ele agora chama $B.k()$.

Se a transformação não envolver o programa inteiro, alguns métodos que não foram modificados podem não ser impactados e, assim, não mudar de comportamento. Observe que os métodos $A.k()$ e $B.k()$ nunca vão mudar de comportamento. O método $C.m()$ muda de comportamento, pois ele foi modificado. No programa original ele é herdado da classe B e chama A.k(). No programa modificado ele está implementado e chama $B.k()$. Já o método $C.teste()$ não foi modificado. Entretanto, ele exercita o método m que foi modificado. Logo, ele é considerado impactado e pode mudar de comportamento. Neste exemplo ele mudou.

Em um programa pequeno como este, é fácil identificar os métodos que podem mudar de comportamento devido à transformação. No entanto, em programas grandes não é trivial fazer esta análise. Por estar lidando com programas orientados a objetos, a complexidade da análise é ainda maior. Algumas características desse tipo de sistema, como polimorfismo, herança e encapsulamento tornam a análise mais complicada. O objetivo deste capítulo é apresentar uma abordagem para analisar uma transformação em programas orientados a objetos e identificar os métodos do programa que possam ter mudado de comportamento.

Código Fonte 3.1: Programa Original

```
1  public class A {
2    public int k() {
3      return 1;
4    }
5  }
6  public class B extends A {
7    public int k() {
8      return 2;
9    }
10   public int m() {
11     return super.k();
12   }
13 }
14 public class C extends B {
15   public int teste() {
16     return m();
17   }
18 }
```

Código Fonte 3.2: Programa Modificado

```
1  public class A {
2    public int k() {
3      return 1;
4    }
5  }
6  public class B extends A {
7    public int k() {
8      return 2;
9    }
10 }
11 public class C extends B {
12   public int m() {
13     return super.k();
14   }
15   public int teste() {
16     return m();
17   }
18 }
```

Figura 3.1: Transformação que move método para classe filha e muda o comportamento do programa

3.2 Abordagem

Nesta seção descrevemos como funciona a abordagem de análise de impacto proposta neste trabalho. Iniciamos a seção com uma visão geral da abordagem (Seção 3.2.1). Em seguida, explicamos com detalhes as subtransformações (Seção 3.2.2) e leis da análise de impacto (Seção 3.2.3). Na Seção 3.2.4, mostramos um exemplo de como aplicar a técnica para analisar uma transformação. Por fim, na Seção 3.2.5, descrevemos a especificação formal utilizada para formalizar as leis da análise de impacto.

3.2.1 Visão Geral

Nesta seção, descrevemos a visão geral de nossa abordagem. Atualmente, ela está implementada em Java. Apesar disso, a técnica pode ser similarmente aplicada para outras linguagens orientadas a objetos.

A Figura 3.2 ilustra de maneira geral o processo da nossa abordagem de análise de impacto. Ela recebe como entrada o programa original e o programa modificado pela transformação e retorna os métodos impactados pela mudança. O primeiro passo da análise de impacto é caracterizar a transformação decompondo sua diferença em um conjunto de subtransformações. Para cada subtransformação, são identificados os métodos diretamente impactados no programa utilizando um conjunto de leis da análise de impacto. (Passo 2). O conjunto de métodos diretamente impactados será a união dos conjuntos impactados por cada subtransformação (Passo 3). Além disso, também identificamos os métodos indiretamente impactados, ou seja, que exercitam direta ou indiretamente algum método do conjunto de métodos diretamente impactados em qualquer nível de indireção (Passo 4). Por fim, no Passo 5, o conjunto de métodos impactados pela transformação é a união do conjunto de métodos direta e indiretamente impactados.

3.2.2 Subtransformações

O primeiro passo da nossa abordagem é decompor a transformação em subtransformações para analisar o impacto de cada uma separadamente no programa. Por exemplo, se um método for adicionado ao programa, então nós consideramos essa adição como uma subtransformação e identificamos os métodos que foram impactados pela adição do método. As

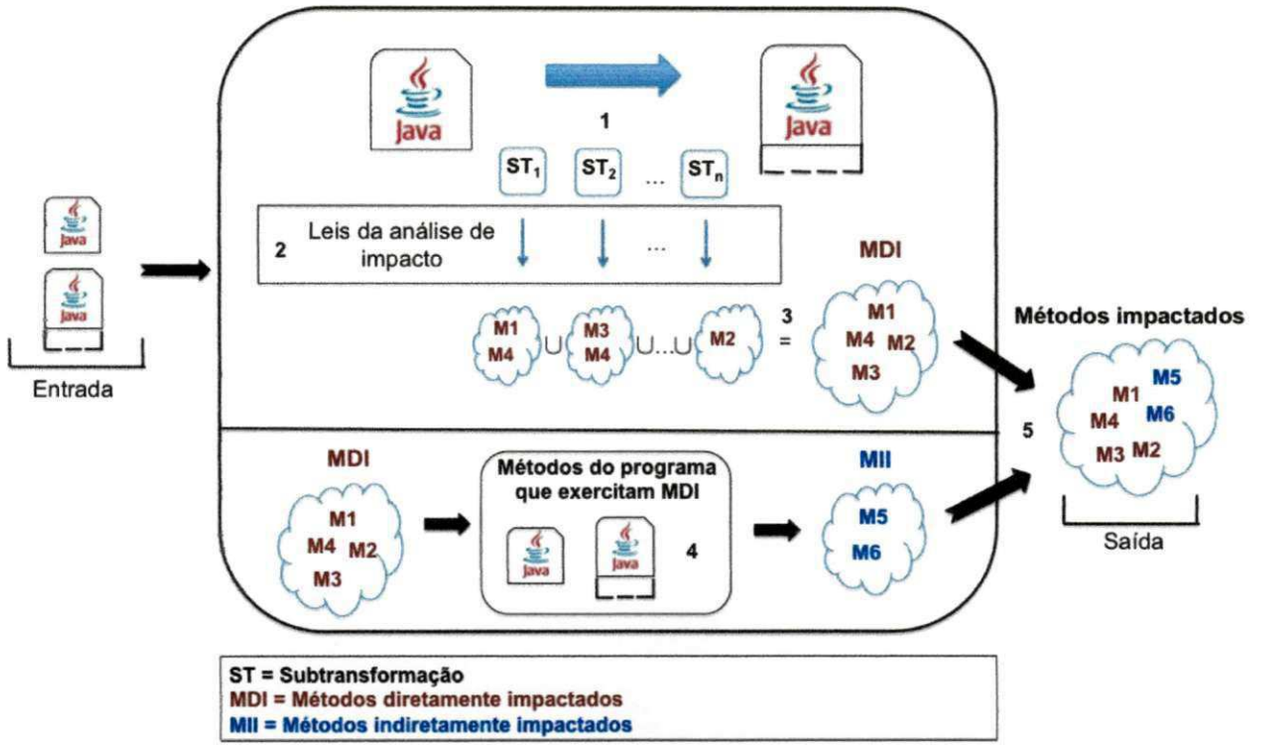


Figura 3.2: Análise de Impacto da Mudança

subtransformações consideradas por Safira são: AM (adicionar método), DM (remover método), CMB (mudar corpo de método), CMM (mudar modificador de método), AF (adicionar atributo), DF (remover atributo), CFM (mudar modificador de atributo), CFI (mudar inicializador de atributo) e CSFI (mudar inicializador estático de atributo). As subtransformações estão listadas na Tabela 3.1. A seguir explicaremos com mais detalhes cada subtransformação.

As subtransformações adicionar e remover métodos e atributos são autoexplicativas. A subtransformação CMB representa qualquer mudança nas instruções do corpo de um método. Por exemplo, modificar, acrescentar ou remover alguma instrução modifica o corpo do método. As mudanças são analisadas a nível de *bytecodes*. As subtransformações CMM e CMF representam mudanças de adicionar, modificar ou remover modificadores de métodos e atributos, respectivamente. Por exemplo, são mudanças em modificadores tornar um atributo estático ou aumentar a visibilidade de um método. Por fim, as subtransformações CFI e CSFI consistem em adicionar ou remover inicializadores de atributos ou modificar o valor

de inicialização de atributos de instância e de classe (estático), respectivamente.

Subtransformações
AM – Adicionar método
DM – Deletar método
CMB – Mudar corpo de método
CMM – Mudar modificador de método
AF – Adicionar atributo
DF – Deletar atributo
CFM – Mudar modificador de atributo
CFI – Mudar inicializador de atributo
CSFI – Mudar inicializador estático atributo

Tabela 3.1: Subtransformações utilizadas na análise de impacto realizada por Safira

3.2.3 Leis da Análise de Impacto

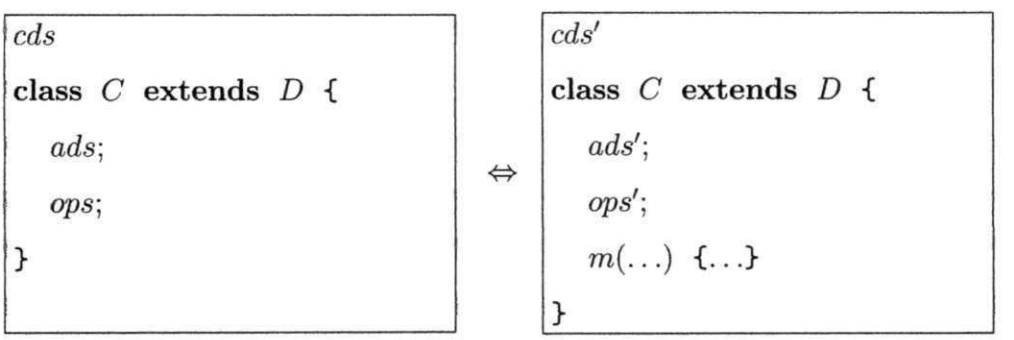
Após a transformação ser decomposta em subtransformações, o passo seguinte é calcular o impacto de cada uma no programa. Para isso, definimos leis que identificam o conjunto de métodos impactados por cada subtransformação considerada em nossa análise de impacto. Utilizamos uma notação formal para especificar o conjunto impactado definido por cada lei. A seguir descrevemos a formalização de algumas subtransformações usando as leis. Cada lei declara dois *templates* de um programa. As meta-variáveis cds, ads e ops definem um conjunto de classes, atributos e métodos, respectivamente. Por fim, a lei define o conjunto de métodos impactados ao ser aplicada no programa.

A Lei 1 adiciona um método m em uma classe C quando aplicada da esquerda para a direita e remove o método quando aplicada da direita para esquerda. Se a classe C não pertencer a uma hierarquia, o conjunto impactado será formado apenas por $C.m$. Entretanto, se existir algum ancestral de C na hierarquia que implemente o método m, os métodos que antes herdavam m do ancestral de C e após a mudança passou a herdar o m de C podem ter seu comportamento modificado. Por isso, também consideramos como impactados todos os métodos m que herdam de C. A especificação formal está descrita na Lei 1. O operador $<^*$ define um relacionamento direto ou indireto de herança.

A Lei 2 modifica o corpo de um método m de uma classe C. O método modificado $C.m$ será impactado. Além disso, se a classe possuir subclasses na hierarquia, também serão

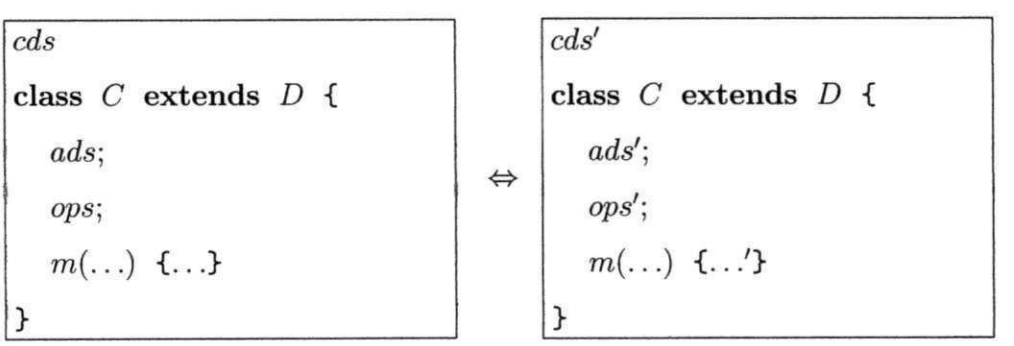
impactados os métodos herdados desse método. Por exemplo, seja B subclasse de C. Se B não implementar o método m e herdá-lo de C, então esse método herdado também será impactado. A lei definida para a subtransformação CMM especifica o conjunto de métodos impactados de maneira similar a Lei 2.

Lei 1 ⟨adicionar/remover método⟩



(↔) Seja F o descendente mais próximo de C que declara um método m. O conjunto de métodos impactados é {n:Metodo | ∃ E:Classe | (F <* E ∧ E ≤* C) ∧ (n ∈ metodos(cds') ∪ ops') ∧ (n = $E.m$))}.

Lei 2 ⟨mudar corpo método⟩

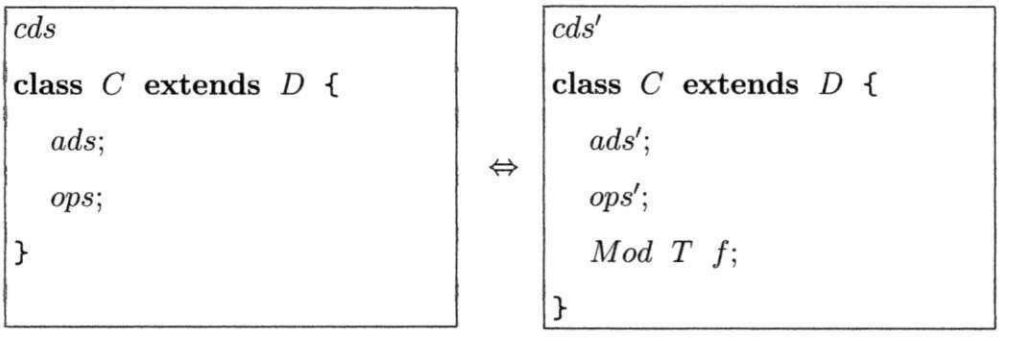


(↔) Seja F o descendente mais próximo de C que declara um método m. O conjunto de métodos impactados é {n:Metodo | ∃ E:Classe | (F <* E ∧ E ≤* C) ∧ (n ∈ metodos(cds') ∪ ops') ∧ (n = $E.m$))}.

A Lei 3 adiciona um atributo f do tipo T em uma classe C quando aplicada da esquerda para a direita e remove o atributo quando aplicada da direita para esquerda. Se a classe C não

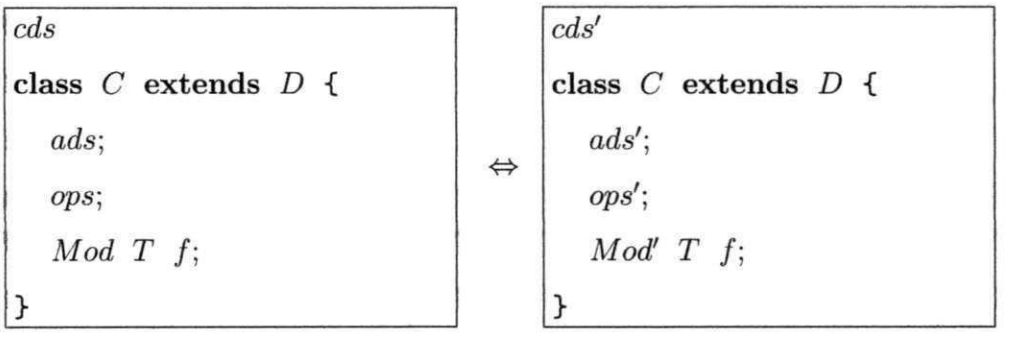
pertencer a uma hierarquia, o conjunto impactado será formado pelos métodos que chamam o atributo $C.f$. Se a classe pertencer a uma hierarquia, também serão impactados os métodos que chamam algum atributo f herdado de C.

Lei 3 ⟨adicionar/remover atributo⟩



(↔) Seja F o descendente mais próximo de C que declara um atributo f. O conjunto de métodos impactados é {m:Metodo | ∃ E:Classe | (F <* E ∧ E ≤* C) ∧ (m ∈ metodos(cds') ∪ ops') ∧ ($E.f$ ⊆ comandos(m))}.

Lei 4 ⟨mudar modificador de atributo⟩



(↔) Seja F o descendente mais próximo de C que declara um atributo f. O conjunto de métodos impactados é {m:Metodo | ∃ E:Classe | (F <* E ∧ E ≤* C) ∧ (m ∈ metodos(cds') ∪ ops') ∧ ($E.f$ ⊆ comandos(m))}.

A Lei 4 muda o modificador (Mod) de um atributo f de uma classe C. Consideramos os modificadores de acesso (*private*, *public*, *protected* e *default*) e outros modificadores como *final*, *static* e *abstract*. Os métodos que chamam o atributo $C.f$ serão impactados. Além

disso, também serão impactados todos os métodos que chamam algum atributo f herdado de C. As leis definidas para as subtransformações CFI e CSFI especificam o conjunto de métodos impactados de maneira similar a Lei 4.

Quando um método ou atributo é removido, o impacto é analisado apenas no programa original, pois no programa modificado a entidade não existe mais. De maneira similar, quando é adicionado, analisa-se apenas o programa modificado. Se algum método ou atributo for modificado, é analisado o impacto nas duas versões do programa. O conjunto de todas as entidades impactadas será a união dos conjuntos de métodos impactados por cada subtransformação. Além disso, a nossa análise também inclui todos os métodos e construtores que exercitam este conjunto impactado direta ou indiretamente em qualquer nível de indireção.

3.2.4 Exemplo

Nesta seção utilizamos a abordagem proposta para identificar os métodos impactados pela transformação mostrada na Figura 3.1. O primeiro passo é decompor a transformação em subtransformações. Identificamos duas subtransformações: DM (remover o método m da classe B) e AM (adicionar o método m na classe C). O segundo passo é identificar os métodos impactados por cada subtransformação utilizando as leis de análise de impacto. Para a subtransformação DM ($B.m$), serão impactados o método removido ($B.m$) e o método herdado ($C.m$). Para a subtransformação AM ($C.m$) será impactado apenas o método adicionado ($C.m$).

O conjunto diretamente impactado será a união do conjunto impactado por cada subtransformação. Neste exemplo, o conjunto de métodos diretamente impactados será composto pelos métodos $B.m$ e $C.m$. Também são considerados impactados os métodos que exercitam direta ou indiretamente pelo menos um método diretamente impactado. Então, identificamos o método $C.teste$ que exercita os métodos $B.m$ e $C.m$. Logo, os métodos impactados pela transformação serão: $B.m$, $C.m$ e $C.teste$.

3.2.5 Especificação formal

Nós especificamos o conjunto impactado por cada lei da análise de impacto utilizando uma notação formal de lógica de primeira ordem (Seção 3.2.3) e codificamos em Alloy[1], uma linguagem de especificação formal, baseando-se em uma teoria de Java em Alloy. O objetivo de codificar as leis em Alloy foi evitar ambiguidades na especificação e checar se ela está consistente.

O Código 3.3 mostra parte da especificação em Alloy da lei que representa a adição de um método em uma classe (Lei 1). A função `addedMethods[source,target: Program]` retorna todos os métodos adicionados pela transformação do programa original para o programa modificado. A função `law1` retorna todos os métodos impactados pela subtransformação *adicionar método*. Os métodos impactados pela adição de um método serão: o método adicionado e os métodos herdados do método adicionado. Também incluímos no conjunto impactado os métodos que exercitam diretamente o método adicionado (`exerciseMethod[impactedMethods: set Method]`). O predicado `contains[m: Method, methodsSet: set Method]` verifica se existe um método m no conjunto de métodos $methodsSet$.

Analisamos a especificação em Alloy, utilizando a ferramenta Alloy Analyzer, que permite checar se a especificação está consistente. Essa checagem é realizada por asserções especificadas em blocos `assert`. A ferramenta procura para um dado escopo algum contraexemplo que não satisfaça a asserção. Se não for encontrado nenhum contraexemplo significa que ela está consistente para o escopo avaliado. Fizemos algumas asserções para termos mais confiança de que a especificação está correta. O Código 3.4 mostra a asserção `testLaw1` de que todo método adicionado é impactado. Essa asserção é checada pelo comando `check testLaw1 for 6`. Nós escolhemos o escopo de tamanho 6 para checar todas as asserções. Para escopos maiores, as checagens ficam muito custosas, pois o Alloy gera todas as soluções possíveis dentro do escopo determinado. De maneira similar, especificamos asserções para checar as outras leis. A especificação completa encontra-se no Apêndice A.

[1] http://alloy.mit.edu/alloy/

Código Fonte 3.3: Parte da Especificação em Alloy das Leis de Análise de Impacto

```
1  fun impactedMethods [source, target: Program]  : set Method {
2    {m: Method | contains[m, law1[source, target]] ||
3      contains[m, law2[source, target]] ||
4      contains[m, law3[source, target]] ||
5      contains[m, law4[source, target]] ||
6      contains[m, law5[source, target]] ||
7      contains[m, law6[source, target]]
8    }
9  }
10
11 fun law1[source, target: Program]  : set Method {
12   {m: Method | contains[m, addedMethods[source, target]]  ||
13     contains[m, inheritedMethods[addedMethods[source, target]]] ||
14     contains[m, exerciseMethod[addedMethods[source, target]]]
15   }
16 }
17
18 fun addedMethods [source, target: Program] : set Method {
19   {m: Method | not (contains[m, source.methods]) &&
20     contains[m, target.methods] }
21 }
22
23 pred contains[m: Method, methodsSet: set Method] {
24   some m1: Method | some c, c1: Class |
25     m1 in methodsSet &&
26     m1.id = m.id && m1.param = m.param &&
27     m in c.methods && m1 in c1.methods && c.id = c1.id
28 }
29
30 fun exerciseMethod[impactedMethods: set Method]  : set Method {
31   {m2: Method | some mi: MethodInvocation, m: Method |
32     contains[m, impactedMethods] &&
33     m.id = mi.id &&  mi = m2.b}
34
35 }
```

Código Fonte 3.4: Asserção

```
1 assert testLaw1 {
2   all m: Method, p1,p2: Program |
3     (not contains[m, p1.methods] &&
4       contains[m, p2.methods] ) =>
5     contains[m, impactedMethods[p1,p2]]
6 }
7
8 check testLaw1 for 6
```

3.3 Cobertura da Mudança

Um critério de teste de software é o critério que define o que constitui um teste adequado, ou seja, define quais propriedades de um programa precisam ser exercitadas para a coleção de testes ser considerada completa [23; 82]. Nós definimos neste trabalho a métrica cobertura da mudança como critério de adequação dos testes. A seguir nós descrevemos como a calculamos. Ela pode ser medida a nível de métodos (Seção 3.3.1) ou a nível de *statements* (Seção 3.3.2), assumindo que os métodos impactados identificados por nossa abordagem representam a mudança feita no programa.

Código Fonte 3.5: Coleção de testes do programa ilustrado na Figura 3.1

```
1  @Test
2  public void teste1() {
3    A a = new A();
4    int f = a.k();
5    assertEquals(f,1);
6  }
7  public void teste2() {
8    A a = new B();
9    int f = a.k();
10   assertEquals(f,2);
11 }
12 public void teste3() {
13   A a = new C();
14   int f = a.k();
15   assertEquals(f,2);
```

```
16 }
17 public void teste4() {
18   A a = new C();
19   int f = a.m();
20   assertEquals(f,1);
21 }
22 public void teste5() {
23   A a = new C();
24   int f = a.teste();
25   assertEquals(f,1);
26 }
```

3.3.1 Nível de Métodos

A cobertura da mudança a nível de métodos representa a porcentagem de métodos impactados pela mudança que são exercitados pela coleção de testes. Seja I o conjunto de métodos impactados e E o conjunto de métodos impactados exercitados pelos testes, onde $E \subset I$. A cobertura da mudança a nível de métodos C, é definida como $C = \frac{\#E}{\#I}$.

Considere o programa da Figura 3.1 e sua coleção de testes (Código 3.5). Os testes 1, 2 e 3 exercitam os métodos $A.k$, $B.k$ e $C.k$, respectivamente. O teste 4 exercita o método $C.m$ e o teste 5 exercita o método $C.teste$. Utilizando a análise de impacto proposta neste trabalho, identificamos 3 métodos impactados: $B.m$, $C.m$ e $C.teste$. Vamos calcular a cobertura da mudança utilizando a coleção de testes mostrada no Código 3.5 e a mesma coleção com a remoção de alguns métodos de testes:

- A coleção de testes exercita os 3 métodos impactados. O método $B.m$ é exercitado pelo método $C.teste$ quando os testes são executados no programa original. Então, a cobertura da mudança será: $C = \frac{3}{3} = 100\%$.
- Removendo os testes 4 e 5 da coleção de testes, nenhum método impactado vai ser exercitado pelos testes. Então, a cobertura da mudança será: $C = 0\%$

3.3.2 Nível de *Statements*

A cobertura da mudança medida a nível de *statements* representa a porcentagem de instruções de métodos impactados que são executadas pelos testes. Seja *IS* o conjunto de *statements* dos métodos impactados e *ES* o conjunto de *statements* dos métodos impactados executados pelos testes, onde $ES \subset IS$. A cobertura da mudança a nível de *statements CS*, é definida como $CS = \frac{\#ES}{\#IS}$.

3.4 Safira

Nesta seção nós apresentamos Safira, um analisador de impacto da mudança que identifica métodos que possam ter mudado de comportamento após uma transformação. Ela pode ser utilizada através de uma interface de linha de comando, que recebe como entrada duas versões de um programa Java (o original e o modificado por uma transformação) e reporta os métodos impactados pela transformação. Além disso, ela reporta as subtransformações (métodos e atributos removidos, adicionados ou modificados). Para executar Safira, basta utilizar o seguinte comando:

$$java - jar\ safira.jar\ p\_original\ p\_modificado$$

Onde, $p\_original$ e $p\_modificado$ são os caminhos para o programa original e o programa modificado, respectivamente.

O código 3.6 mostra como utilizar Safira no ambiente de desenvolvimento. Com a execução desse código colocando os caminhos para os programas da Figura 3.1, Safira analisa a transformação e mostra os resultados, que estão exibidos na Figura 3.3. Ao chamar o construtor da classe $ClassManipulator$ passando os caminhos para as duas versões do programa, a análise de impacto é feita automaticamente. O método $ClassManipulator.print()$ imprime os métodos impactados e as subtransformações.

Código Fonte 3.6: Exemplo de como utilizar Safira para analisar uma transformação

```
1 public static void main(String[] args) {
2   String original = "caminho do programa original";
3   String modificado = "caminho do programa modificado";
4
5   ClassManipulator c = new ClassManipulator(original, modificado);
6   c.print();
7 }
```

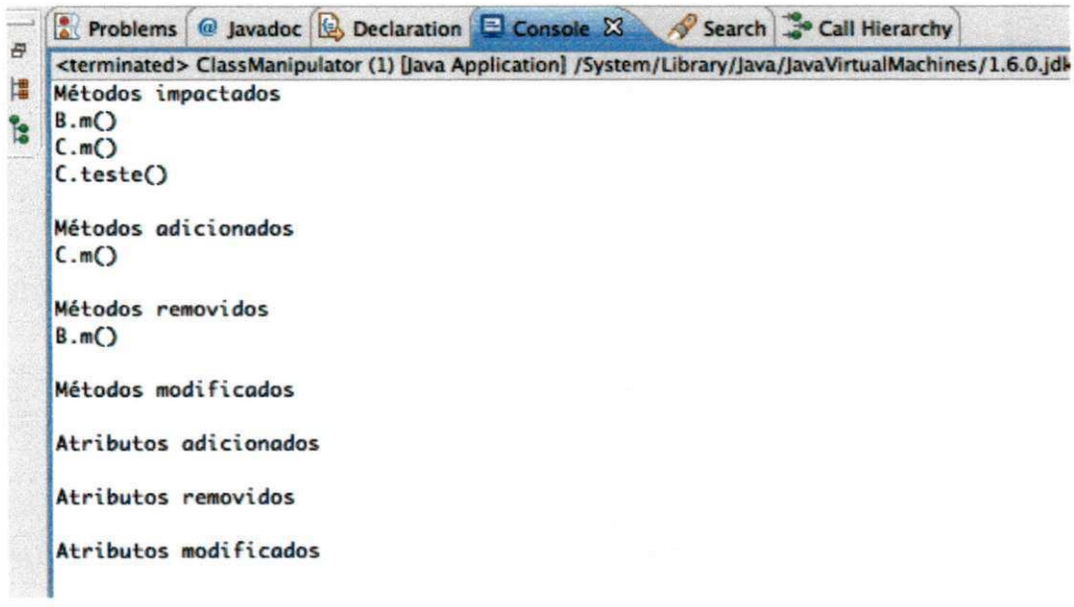


Figura 3.3: Resultado da análise de Safira na transformação ilustrada da Figura 3.1

Se o usuário precisar utilizar dados da análise de impacto para realizar alguma outra análise, é possível acessar alguns atributos de $ClassManipulator$, como por exemplo, a lista de métodos impactados, de métodos em comum impactados e dos métodos e atributos adicionados, removidos ou modificados. Essas listas contém objetos do tipo $Method$ ou $Field$ que são subtipos do objeto $Entity$ ilustrado na Código 3.7. O atributo $fullname$ representa o nome completo da entidade, incluindo pacotes e classe. O $simpleName$ é apenas o nome da entidade. Já o $classFullName$ é o $fullName$ da classe que a entidade pertence. Os atributos $visibility$ e $modifier$ representam a visibilidade e outro modificador, se houver, da entidade. Por fim, o atributo $inherited$ informa se a entidade é herdada ($inherited = true$) ou implementada na classe. Além desses atributos, cada subtipo de $Entity$ contém alguns

atributos específicos. Por exemplo, a classe *Method* contém atributos que representam parâmetros do método, tipo de retorno e instruções do corpo do método.

Código Fonte 3.7: Classe de Safira que Representa uma Entidade de um Programa

```
1 public class Entity {
2   String fullName;
3   String simpleName;
4   String classFullName;
5   String visibility;
6   boolean inherited;
7   String modifier;
8 }
```

3.4.1 Arquitetura

Safira foi implementada na linguagem Java. Para acessar e manipular os elementos do projeto, como classes, métodos e atributos, foi utilizada a api do ASM[2], um framework para manipular e analisar Java Bytecode. Ela permite realizar toda a análise estática do código. O ASM implementa o padrão *visitor* para facilitar o desenvolvedor obter as informações de entidades do código.

A ferramenta contém três módulos principais: *UI*, *Extractor* e *Manipulator*. O módulo *UI* é responsável pela comunicação da ferramenta com o usuário. Como foi mostrado, a interface é de linha de comando mas é extensível para implementação de uma interface gráfica. O módulo *Extractor* tem dois componentes: *ClassExtractor* e *ProgramBuilder*. O primeiro recebe o programa, extrai as classes e passa para o *ASM* que retorna informações das classes recebidas. O segundo utiliza essas informações para montar os programas lógicos armazenando as informações nas classes que implementam *Entity*. O módulo *Manipulator* recebe as informações dos programas do módulo *Extractor*. A primeira etapa é identificar a transformação realizada no programa para decompor em subtransformações, que é feita pelo *ChangeDecomposer*. Em seguida, o componente *ChangeImpactAnalyzer* recebe as subtransformações e identifica os métodos impactados pela mudança. A Figura 3.4 ilustra a arquitetura de Safira.

[2] http://asm.ow2.org/

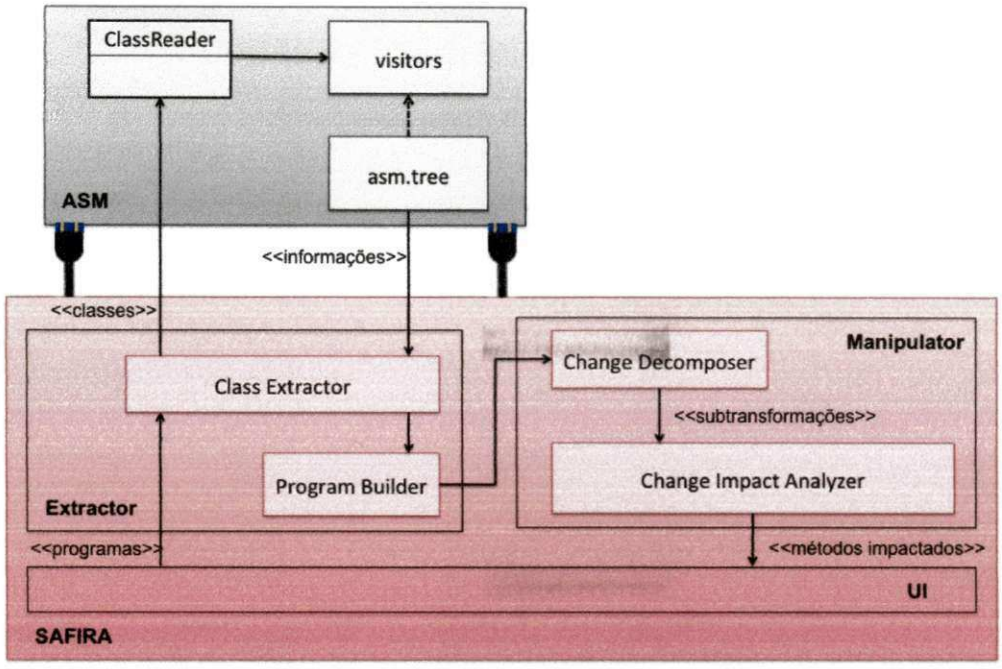


Figura 3.4: Arquitetura de Safira

3.5 Abordagens Similares

Outros analisadores de impacto de mudança para programas orientados a objetos foram propostos na literatura. A maioria se baseia na análise do código do software. Algumas linhas de pesquisa se destacam: análise de impacto dinâmica [38], análise de impacto baseada em testes [56; 80], análise baseada em modelos UML [9] e análise algorítmica do código [44; 28]. A análise de impacto pode ser feita antes da transformação do código para prever o impacto de uma mudança proposta. Nesse tipo de análise, é necessário ter informações sobre a mudança desejada. A análise também pode ser feita após a transformação no código, para identificar partes do programa que possam ter mudado de comportamento. Além disso, algumas análises dependem de alguns parâmetros para serem realizadas, como por exemplo, uma coleção de testes ou informações adicionais da transformação.

Nosso objetivo, em criar um analisador de impacto da mudança, foi que ele precisasse apenas de duas versões de um programa e retornasse os métodos que pudessem ter mudado de comportamento. Inicialmente, pensamos em utilizar uma ferramenta já proposta, Chianti [56], para realizar a análise de impacto. Entretanto, além da ferramenta não estar

disponível, sua análise depende de uma coleção de testes, o que não é desejado para o objetivo deste trabalho. Ainda assim, nossa abordagem é baseada em algumas características de Chianti, como por exemplo, decompor a transformação em transformações menores.

Nesta seção faremos um estudo comparativo das abordagens de análise de impacto da mudança de Safira, Chianti [56] e FaultTracer [80][3]. Escolhemos utilizar a abordagem de Chianti neste estudo devido à similaridade de sua abordagem com a de Safira, discutida anteriormente. FaultTracer reimplementa a abordagem de Chianti com a adição de algumas melhorias, por isso, também o utilizamos para este estudo. Falaremos com alguns detalhes sobre as abordagens de Chianti e FaultTracer nas Seções 3.5.1 e 3.5.2, respectivamente. Por fim, mostraremos nosso estudo comparativo entre as três abordagens na Seção 3.5.3.

3.5.1 Chianti

Chianti é um analisador de impacto para programas Java. Ele analisa duas versões de um programa e a coleção de testes de regressão. Seu objetivo é identificar os testes de regressão que foram afetados por um conjunto de mudanças feitas no programa e as mudanças que afetaram cada teste para facilitar a depuração do código. Além disso, Chianti identifica as mudanças que podem ser incorporadas ao programa com segurança.

A partir de duas versões de um programa, Chianti decompõe sua diferença em um conjunto de mudanças atômicas, cuja granularidade de cada uma é a nível de classe, método ou atributo. São consideradas 16 mudanças atômicas na análise de Chianti (Tabela 3.2). Uma ordem parcial entre as mudanças é determinada. Essa ordem parcial captura as dependências entre as mudanças que precisam ser respeitadas de modo que possa ser criado um programa sintaticamente válido. Regras foram formalizadas para modelar as dependências entre as mudanças atômicas.

Chianti constrói grafos de chamadas executando os testes no programa original. Um teste é considerado afetado, se houver alguma mudança do tipo modificar método, deletar método vazio ou *lookup change* no grafo de chamadas. Para cada teste afetado, são construídos grafos de chamadas a partir da execução dos testes no programa modificado. As mudanças que estiverem nos grafos e suas dependências são as que afetaram o teste analisado.

[3] As ferramentas Chianti e FaultTracer não estão disponíveis para download.

Mudanças atômicas	Chianti	FaultTracer
AEM – Adicionar método vazio	✔	✔
DEM – Deletar método vazio	✔	✔
CM – Modificar método	✔	✔
AF – Adicionar atributo	✔	✔
DF – Deletar atributo	✔	✔
LC – *Lookup change* de método	✔	✔
CFI – Modificar inicializador de atributo	✔	✔
CSFI – Modificar inicializador de atributo estático	✔	✔
AEC – Adicionar classe vazia	✔	
DEC – Deletar classe vazia	✔	
AI – Adicionar inicializador de instância vazio	✔	
DI – Deletar inicializador de instância vazio	✔	
CI – Modificar inicializador de instância	✔	
ASI – Adicionar inicializador estático vazio	✔	
DSI – Deletar inicializador estático vazio	✔	
CSI – Modificar inicializador estático	✔	
LC_f – *Lookup change* de atributo		✔

Tabela 3.2: Mudanças atômicas consideradas nas análises de Chianti e FaultTracer

3.5.2 FaultTracer

FaultTracer também é um analisador de impacto para programas Java. Ele reimplementa a abordagem de análise de impacto do Chianti com alguns diferenciais. Seu objetivo é melhorar a análise de Chianti com respeito a seleção dos testes afetados pela mudança e a depuração das mudanças que afetaram cada teste. FaultTracer herda 8 mudanças atômicas de Chianti. A Tabela 3.2 mostra as mudanças atômicas consideradas na análise de Chianti e FaultTracer. Observe que ele considera uma mudança atômica, LCf (*lookup change field*), não considerada por Chianti. A mudança LCf representa mudanças de atributos herdados na hierarquia. Com essa mudança FaultTracer consegue detectar testes afetados, que possam ter mudado de comportamento, que Chianti não detecta. Embora as mudanças atômicas adicionar e remover classe vazia e mudanças em inicializadores de instância e estáticos não sejam consideradas na análise do FaultTracer, os autores argumentam que é possível decompor qualquer mudança de granularidade maior nesses 9 tipos básicos de mudanças e por isso, não considerou as demais mudanças que Chianti considera.

Outro diferencial da análise do FaultTracer é a adição de algumas regras para modelar

melhor as dependências entre as mudanças atômicas. Chianti não modela de maneira precisa as dependências entre as mudanças atômicas quando envolve sobrescrita de método e herança de atributo. FaultTracer realizou um refinamento das regras para derivar as dependências entre as mudanças mais precisamente nesses cenários.

3.5.3 Comparação

Nesta seção faremos um estudo comparativo entre as abordagens de Chianti, FaultTracer e Safira. O nosso objetivo neste estudo é identificar vantagens e desvantagens de cada abordagem utilizando alguns critérios de comparação. A Tabela 3.3 resume os principais critérios de comparação. Os critérios foram escolhidos a partir de um estudo das principais diferenças entre as abordagens e da identificação de pontos que possam mostrar e diferenciar características relevantes de cada uma. Nas próximas seções discutiremos com mais detalhes cada critério utilizado neste estudo.

Noção da análise

Neste trabalho nos referiremos a noção da análise por mundo aberto ou mundo fechado. No nosso contexto, a noção de mundo fechado representa o impacto definido apenas em termos de uma coleção de testes. Ou seja, mesmo que o programa mude de comportamento, só haverá algum impacto se algum teste for afetado pela mudança. Na noção de mundo aberto, o impacto é definido em termos de todo o programa. Qualquer método do programa que possivelmente mude de comportamento é considerado impactado.

Este critério é importante para ressaltar o que cada abordagem considera ser um impacto. Nós temos a noção de mundo aberto. A análise de impacto de Safira considera qualquer possível mudança comportamental em todo o programa após uma transformação. Já as abordagens de Chianti e FaultTracer avaliam se a transformação pode mudar o comportamento de algum teste, ou seja, se algum teste foi impactado pela mudança. Se não foi, então não houve impacto. Essa é a noção de mundo fechado. Utilizando essa noção, mesmo um método tendo seu comportamento alterado após a transformação, pode não ser considerado impactado se ele não for exercitado por algum teste.

Dependência

Como já discutido anteriormente, algumas abordagens de análise de impacto dependem de alguns parâmetros ou informações adicionais para serem realizadas. Neste estudo comparativo vamos discutir a dependência das abordagens em relação a uma coleção de testes. É importante ressaltar este critério porque nem sempre o usuário vai ter uma coleção de testes no programa que ele deseja analisar ou a coleção pode não ter uma boa cobertura do programa.

As abordagens de Chianti e FaultTracer dependem de uma coleção de testes. Essa dependência está relacionada com a noção de mundo aberto e mundo fechado discutida na seção anterior. Para explicar melhor essa dependência e suas consequências na análise de impacto, voltaremos a utilizar o programa ilustrado na Figura 3.1 e sua coleção de testes 3.5. Como já foi mostrado na Seção 3.3, os testes 4 e 5 exercitam a mudança. Utilizando as abordagens de Chianti e FaultTracer esses métodos são identificados como afetados. Já os testes 1, 2 e 3 não exercitam a mudança e consequentemente não mudam de comportamento. Então, se os testes 4 e 5 fossem removidos ou não existissem na coleção de testes impossibilitaria essas abordagens de detectar a mudança comportamental no programa, levando os programadores a acharem que a transformação está correta. Com isso, as abordagens de Chianti e FaultTracer dependem que os testes cubram bem a mudança, para que a análise de impacto seja útil para ajudar os programadores. Se a coleção de testes for vazia não é feita análise de impacto.

Safira não depende de coleção de testes. Ela recebe duas versões de um programa e identifica os métodos que foram impactados pela mudança. Inclusive, os métodos de casos de testes, se houverem. Neste caso, ela também pode ser usada para priorizar casos de testes de regressão. Esse foi um dos motivos pelo qual nós não reimplementamos as abordagens de Chianti e FautTracer. Eles consideram o impacto apenas nos testes. Nós precisávamos de uma análise em todo o programa. Uma estratégia alternativa que poderia ter sido adotada, era gerar testes automáticos para o programa e utilizar uma dessas abordagens para realizar a análise de impacto. O problema seria o grande consumo de tempo, já que teria a geração de testes além da análise de impacto utilizada para priorizar os testes.

	Chianti	FaultTracer	Safira
Noção da análise (mundo aberto/fechado)	Mundo fechado	Mundo fechado	Mundo aberto
Dependência	Coleção de testes	Coleção de testes	Nenhuma
Tipo de análise	Estática e Dinâmica	Estática e Dinâmica	Estática
Considera transitividade na análise	Não	Não	Sim
Mudanças que afetam um teste	CM, DEM, LC	CM, CF, LC, LC_t	-

Tabela 3.3: Tabela comparativa das abordagens de Chianti, FaultTracer e Safira

Tipo da análise

Existem, atualmente, dois tipos de abordagens de análise de impacto: estática e dinâmica. A análise estática avalia o código fonte para identificar os possíveis impactos. Já, a análise dinâmica avalia rastros de execuções do software para identificar os métodos que chamam entidades que foram modificadas. Uma das vantagens da análise estática é ela ser conservadora, ou seja, poder identificar todas as possíveis entidades que mudaram de comportamento. Por outro lado, a análise dinâmica obtém resultados mais precisos, já que analisa execuções possíveis do programa. O tipo de análise adotada na abordagem pode influenciar fatores como tempo e complexidade da análise. Então, é isso que vamos discutir a seguir.

A análise feita por Chianti e FaultTracer é estática e dinâmica. A parte estática das abordagens é a decomposição da transformação em mudanças atômicas e a construção dos grafos de dependências entre as mudanças. A parte dinâmica é a construção dos grafos de chamadas que é feita a partir da execução dos testes no programa. Como toda a coleção de testes é executada para identificar os testes afetados, o consumo de tempo pode ser bem maior do que uma análise estática de código. Além disso, a eficácia da análise dinâmica está relacionada com a qualidade dos testes. Se os testes não cobrirem bem a mudança, a análise não vai ser muito útil. Por outro lado, a identificação dos testes afetados é mais precisa, já que é considerado apenas o que realmente for exercitado na execução.

A abordagem de Safira é estática. Com isso, o consumo de tempo pode ser menor que uma abordagem dinâmica se a coleção de testes executada for grande. Além disso, ela identifica os possíveis métodos impactados. Entretanto, a análise estática identifica um maior número de falso-positivos (detectar uma entidade impactada quando na verdade não é) que a abordagem dinâmica. Por fim, a análise estática é mais complexa, pois precisa analisar todas

as dependências entre as entidades, levando em consideração as propriedades dos sistemas orientados a objetos, como herança e polimorfismo.

Transitividade

Transitividade é uma propriedade comumente utilizada em abordagens de análise de impacto, pois uma mudança pontual pode impactar várias entidades do programa a medida do nível de acoplamento do código. Considere um método modificado m. Se houverem no programa os métodos k e z, tal que, z chama k e k chama m, a mudança que foi feita em m pode mudar o comportamento dos métodos k e z, que exercitam m. No nosso contexto, consideramos que é utilizada a transitividade na análise ao considerar como impactados os métodos que exercitam direta ou indiretamente uma entidade impactada.

Safira identifica os métodos impactados por cada subtransformação e todos os métodos que exercitam direta ou indiretamente os métodos impactados em qualquer nível de indireção. Desta maneira, ela considera transitividade na sua análise. Chianti e FaulTracer, não identificam como impactados os métodos que exercitam direta ou indiretamente as mudanças atômicas. Como essas abordagens não consideram transitividade, não é possível identificar todos os métodos que possam ter mudado de comportamento, como Safira faz. Seu foco é em identificar os testes afetados pela mudança.

Detecção de testes impactados

Neste critério vamos comparar a técnica utilizada por cada abordagem para identificar testes impactados. Esse critério pode impactar diretamente a corretude da análise, pois se um teste que mudou de comportamento não for identificado como impactado, significa que a análise não foi realizada corretamente. Para detectar os testes afetados, Chianti constrói grafos de chamadas executando os testes no programa original. Cada nó do grafo representa um método exercitado pelo teste. Os testes que tiverem alguma mudança do tipo modificar método, deletar método vazio ou *lookup change* no grafo de chamadas é considerado afetado. O FaultTracer faz de maneira similar. A diferença é que ele considera como afetados os testes que tiverem mudanças do tipo modificar método, modificar atributo, *lookup change* ou *lookup change* de atributo no grafo de chamadas.

A análise de Safira não depende de testes, então não podemos comparar desta forma

com as outras abordagens. Mas, vamos comparar de uma forma similar. Vamos considerar os casos de testes como um método qualquer do programa. Se Safira incluir um método de teste no conjunto de métodos impactados, concluímos que esse teste foi afetado. No exemplo ilustrando na Figura 3.1, Safira identifica como impactados os métodos $B.m$, $C.m$ e $C.teste$. Considerando também a coleção de testes 3.5 como parte do programa, Safira identifica o teste4 e o teste5 como impactados, pois exercitam os métodos $C.m$ e $C.teste$, também impactados. Da mesma forma, as análises de Chianti e FaultTracer também identificaram o teste4 e teste5 como afetados.

Tanto a abordagem de Chianti quanto a de FaultTracer analisam apenas o programa original para identificar os testes afetados. Essa técnica pode não funcionar muito bem quando envolve remoção e adição de métodos, onde apenas o método adicionado é exercitado pelos testes. Na Figura 3.5 mostramos um exemplo de uma transformação em um programa, com sua coleção de testes, que Chianti e FaultTracer não identificam um teste afetado quando na verdade ele foi impactado e mudou de comportamento. O programa original é formando pela classe A que contém dois métodos: $m(int)$ que retorna "42"e $k()$ que chama os métodos $java.lang.Integer.parseInt(String)$ e $java.lang.String.valueOf(int)$ da biblioteca de java, retornando 23. A transformação renomeia o método $A.m(int)$ para $A.valueOf(int)$. Após a mudança, o método $A.k()$ vai mudar de comportamento retornando 42. Isso acontece porque ele vai chamar o método renomeado $A.valueOf(int)$ ao invés do método da biblioteca de java.

A coleção de testes do programa exercita o único método em comum do programa: $A.k()$. Esse método muda de comportamento e o teste1 é afetado pela mudança. Chianti não detecta esse teste como afetado. Ela decompõe a transformação nas seguintes mudanças atômicas: CM (A.m(int)), DEM (A.m(int)), AEM(A.valueOf(int)) e CM(A.valueOf(int)). O grafo de chamadas de $teste1()$ no programa original vai incluir chamadas ao construtor implícito de A e aos métodos $A.k()$, $java.lang.Integer.parseInt(String)$ e $java.lang.String.valueOf(int)$. Nenhuma mudança atômica está no grafo de chamadas. Logo, esse teste não é considerado afetado, quando na verdade ele além de afetado, mudou de comportamento. O teste vai passar no programa original e vai falhar no programa modificado. A análise do FaultTracer, é feita de maneira similar ao Chianti e não identifica o $test1()$ como afetado.

Safira identifica como impactado o método de teste $teste1$. Nossa análise decompõe a transformação nas seguintes subtransformações: DM (A.m(int)) e AM (A.valueOf(int)). Como o método $A.k()$ e o método de teste $teste1()$ no programa modificado exercitam o método adicionado $A.valueOf(int)$, então eles são considerados como impactados. Chianti e FaultTracer falharam em identificar esse teste afetado porque eles analisam apenas o programa original. Além disso, eles consideram que apenas algumas mudanças atômicas são capazes de afetar um determinado teste. O método adicionado $A.valueOf(int)$ é exercitado pelos testes no programa modificado. Ele não foi identificado porque as abordagens avaliam apenas o programa original para detectar testes afetados. Safira também analisa o programa modificado e com isso identifica o teste impactado.

Figura 3.5: Renomear método resulta em sombreamento de método importado

Código Fonte 3.8: Programa Original

```
1  import static java.lang.String.*;
2
3  public class A {
4    static String m(int i) {
5      return "42";
6    }
7    public int k() {
8      return Integer.
9          parseInt(valueOf(23));
10   }
11 }
```

Código Fonte 3.9: Programa Modificado

```
1  import static java.lang.String.*;
2
3  public class A {
4    static String valueOf(int i) {
5      return "42";
6    }
7    public int k() {
8      return Integer.
9          parseInt(valueOf(23));
10   }
11 }
```

Código Fonte 3.10: Coleção de Testes

```
1  @Test
2  public void teste1() {
3    A a = new A();
4    int f = a.k();
5    assertEquals(f,23);
6  }
```

3.6 Limitações

Nesta seção, descrevemos algumas limitações da nossa análise de impacto da mudança. Uma limitação está relacionada com a API utilizada por Safira. É utilizada a API do ASM para realizar a análise de impacto do código. Essa API analisa o bytecode Java. Diferenças no código fonte que não reflitam no bytecode não são identificadas. Avaliar o código fonte é mais difícil e as ferramentas que encontramos disponíveis para analisar programas a nível de corpo de método, trabalham com bytecode. Outra limitação da abordagem é que atualmente, algumas construções da linguagem não são analisadas. Por exemplo, não analisamos inicializadores estáticos e de instância e classes anônimas na análise de impacto. Então mudanças nessas classes ou inicializadores e, seus respectivos impactos, não vão ser identificadas por Safira. A análise de classes anônimas é uma limitação do ASM. Já a análise dos blocos inicializadores pode ser adicionada na abordagem, pela definição de novas subtransformações e respectivas leis para identificar o conjunto impactado. Nós escolhemos as subtransformações de modo que fosse considerada a maior quantidade possível de estruturas da linguagem. Entretanto, como vimos, o conjunto de subtransformações considerado na nossa análise não é completo. Nas seções seguintes descrevemos outras limitações de Safira mostrando alguns exemplos.

3.6.1 Análise de Fluxo de Dados

O algoritmo de análise de impacto utilizado por nossa abordagem, não realiza análise de fluxo de dados. Então, alguns métodos impactados podem não ser identificados. Por exemplo, suponha a transformação ilustrada na Figura 3.6. O programa original está representado pelo Código 3.11. Ele contém duas classes A e B. A classe A contém um atributo privado var e os métodos $getVar()$ e $setVar(int)$ que retorna e muda o valor do atributo, respectivamente. O construtor da classe é vazio. A classe B contém o método $m()$ que retorna o método $getVar()$ da classe A. Se for aplicada uma transformação que mude o valor de $A.var$ através de uma chamada $setVar(int)$ no construtor da classe A, muda o comportamento do programa. O Código 3.12 representa o programa modificado por essa transformação. Veja que o método $A.getVar()$ retorna 1 no programa original e 2 no programa modificado. O mesmo acontece com o método $B.m()$ que retorna $A.getVar()$. Então, o conjunto de mé-

todos impactados por essa transformação será composto pelos métodos $A. < init > ()$, $A.getVar()$ e $B.m()$.

Utilizando a nossa abordagem, a transformação é decomposta em uma subtransformação: modificar corpo do método $A. < init > ()$. O conjunto de métodos impactados será $A. < init > ()$ que foi modificado e $B.m()$ que exercita o construtor modificado. O método $A.getVar()$ que foi impactado não é identificado. Essa é uma limitação da abordagem por não realizar análise de fluxo de dados.

Figura 3.6: Transformação aplicada no programa altera valor de atributo através de chamada de método e muda o comportamento do programa.

Código Fonte 3.11: Programa Original

```
1  public class A {
2    private int var = 1;
3    public A() {
4
5    }
6    public int getVar() {
7          return this.var;
8    }
9    public void setVar(int newVar){
10         this.var = newVar;
11   }
12 }
13 public class B  {
14   public int m() {
15     A a = new A();
16     return a.getVar();
17   }
18 }
```

Código Fonte 3.12: Programa Modificado

```
1  public class A {
2    private int var = 1;
3    public A() {
4      setVar(2);
5    }
6    public int getVar() {
7          return this.var;
8    }
9    public void setVar(int newVar){
10         this.var = newVar;
11   }
12 }
13 public class B  {
14   public int m() {
15     A a = new A();
16     return a.getVar();
17   }
18 }
```

3.6.2 Análise de Classes de Bibliotecas

A análise de Safira não analisa classes de bibliotecas. Então, alguns impactos podem não ser identificados. Por exemplo, suponha a transformação ilustrada na Figura 3.7. O pro-

grama original está representado pelo Código 3.13. Ele contém as classes A e C. A classe A contém o método $m(Collection < String >)$ que retorna o método $size()$ de $java.util.Collection$ e contém o método $teste()$ que retorna uma chamada ao método $m(Collection < String >)$ da própria classe, passando como parâmetro um objeto da classe C. Isso é possível porque a classe C estende a classe $java.util.Vector$ que implementa a interface $Collection$. Desta forma, recebendo um objeto do tipo C, o método $m(Collection < String >)$ chama $C.size()$. Logo, neste programa, $A.teste()$ retorna 1000. Se $C.size()$ for removido, resultado no programa representado no Código 3.14, o método $A.m(Collection < String >)$ vai passar a chamar o método da $size()$ de $Collection$ quando for chamado pelo método $teste()$, retornando 0. Então, os métodos impactados por essa transformação são: $A.m(Collection < String >)$, $A.teste()$ e $C.size()$.

Utilizando a nossa abordagem, a transformação é decomposta em uma subtransformação: remover método $C.size()$. O conjunto de métodos impactados será apenas o método removido $C.size()$. Os métodos $A.teste()$ e $C.size()$ não são identificados. A análise de Safira identifica que a classe C é subclasse de e $java.util.Vector$ mas não consegue identificar que $Vector$ implementa $Collections$. Essa é uma limitação da abordagem por não avaliar classes de bibliotecas.

Figura 3.7: Transformação aplicada no programa remove um método que sobrescreve método de biblioteca e muda comportamento do programa.

Código Fonte 3.13: Programa Original

```
1  public class A {
2    public int m(Collection<String> c)
         {
3      return c.size();
4    }
5    public int teste() {
6      C c = new C();
7      return m(c);
8    }
9  }
10 public class C<E> extends Vector<E>{
11   public int size() {
12     return 1000;
13   }
14 }
```

Código Fonte 3.14: Programa Modificado

```
1  public class A {
2    public int m(Collection<String> c)
         {
3      return c.size();
4    }
5    public int teste() {
6      C c = new C();
7      return m(c);
8    }
9  }
10 public class C<E> extends Vector<E>{
11
12
13
14 }
```

Capítulo 4

SafeRefactorImpact: Uma Abordagem para Avaliar Refatoramentos Baseada no Impacto da Mudança

Refatoramentos aplicados manualmente são susceptíveis a erros. Além disso, até as melhores ferramentas de refatoração podem aplicar transformações que não preservam o comportamento. Para avaliar se um refatoramento está correto quanto a preservação de comportamento, é necessário avaliar apenas partes do código que possam ter mudado de comportamento. Neste capítulo, descrevemos nossa abordagem para avaliar se uma transformação preservou o comportamento do programa a partir da geração automática de casos de testes apenas para os métodos impactados pela mudança.

O SafeRefactor [68], é uma ferramenta que avalia se uma transformação preserva o comportamento. A técnica original do SafeRefactor avalia todos os métodos em comum de duas versões do programa. Nós implementamos o SafeRefactorImpact, que acopla Safira, o analisador de impacto da mudança proposto neste trabalho, no SafeRefactor, para serem testados apenas os métodos que possam ter mudado de comportamento após a transformação. A seguir, discutimos um exemplo que mostra alguns problemas que ocorrem na prática para testar um refatoramento.

Suponha o programa representado no Código Fonte 4.1. Ele contém as classes A, B e C. A classe A implementa os métodos $k(int)$ e $foo()$. A classe B estende a classe A e implementa os métodos $m()$, $n(int)$ e $bar()$. Por fim, a classe C implementa o método $bar()$.

Suponha também que o desenvolvedor queira aplicar um refatoramento que renomeia o método $B.n(int)$ para $B.k(int)$, resultando no programa representado no Código Fonte 4.2.

Figura 4.1: Renomear método introduz mudança comportamental por habilitar sobrescrita.

Código Fonte 4.1: Programa Original.

```
1  public class A {
2    public int k(int a) {
3      return 1;
4    }
5    public int foo() {
6      return 5;
7    }
8  }
9  public class B extends A {
10   public int m() {
11     return k(2);
12   }
13   public int n(int a) {
14     return 0;
15   }
16   public int bar() {
17     return n(10);
18   }
19 }
20 public class C {
21   public int bar() {
22     return new A().foo();
23   }
24 }
```

Código Fonte 4.2: Programa Modificado.

```
1  public class A {
2    public int k(int a) {
3      return 1;
4    }
5    public int foo() {
6      return 5;
7    }
8  }
9  public class B extends A {
10   public int m() {
11     return k(2);
12   }
13   public int k(int a) {
14     return 0;
15   }
16   public int bar() {
17     return k(10);
18   }
19 }
20 public class C {
21   public int bar() {
22     return new A().foo();
23   }
24 }
```

Uma mudança no código pode afetar outras partes do programa, devido a dependências e inter-relacionamentos entre as entidades do sistema. Por isso, renomear o método $B.n(int)$ também impactou outras entidades do programa, como os métodos $B.bar()$ e $B.m()$. O método $B.bar()$ foi impactado porque ele chama o método $B.n(int)$ no programa original, então como consequência da mudança, essa chamada também foi renomeada no programa modificado. Já o método $B.m()$ chama o método $k(int)$ nas duas versões do programa.

Entretanto, no programa original, é chamado o método $k(int)$ da classe A que retorna 1. Após o refatoramento, $B.m()$ passa a chamar o $k(int)$ da classe B, retornando 0. Logo, seu comportamento foi modificado. Em resumo, os métodos modificados ou impactados por essa mudança foram: $B.n(int)$, $B.k(int)$, $B.bar()$ e $B.m()$. Desses métodos, não é possível testar no programa modificado apenas o método $B.n(int)$, pois ele não existe mais. Os demais métodos são comuns às duas versões do programa e podem ser testados.

Na abordagem tradicional, rodamos toda a coleção de testes para aumentar a confiança de que a mudança no código preservou o comportamento do programa. Entretanto, a coleção testa métodos que não foram impactados e pode deixar de testar métodos que possam ter mudado de comportamento. Como resultado, a execução de toda a coleção pode ser custosa e os testes podem não identificar uma mudança comportamental. Neste exemplo, a coleção de testes do programa está representada no Código 4.3. Ela possui 5 casos de testes. Observe que apenas o $teste3$ exercita um método que pode ter mudado de comportamento. Os demais casos de testes exercitam métodos que não foram modificados ou impactados. Logo, eles não são úteis para testar a mudança. Além disso, os métodos $B.k(int)$ e $B.m(int)$ que foram impactados pela mudança e tiveram seu comportamento modificado, não são exercitados pela coleção de testes. Então, a coleção não identifica a mudança porque não foca em testar os métodos impactados.

Nós mostramos o problema em um programa pequeno. Na prática o problema pode ser maior. Em programas grandes com coleções de testes maiores, o custo de executar toda a coleção para testar uma mudança pode ser grande e desnecessário, já que apenas alguns casos de testes são importantes para avaliar o que foi modificado. Além disso, a coleção de testes pode não exercitar a mudança. Uma boa cobertura dos testes no programa não significa que a mudança esteja coberta. Neste exemplo, os testes exercitam 5 de 7 métodos em comum, ou seja, temos uma cobertura de 71% dos métodos em comum das duas versões do programa. Entretanto, a coleção não identifica a mudança porque os métodos impactados que mudaram de comportamento ($B.k(int)$ e $B.m()$) não estão sendo testados. Então, perdemos tempo testando métodos que não foram impactados e deixamos de identificar mudanças comportamentais porque não testamos todos os métodos que podem ter mudado de comportamento. Por isso, para avaliar se uma transformação preservou o comportamento do programa, é importante testar apenas os métodos impactados pela mudança.

Código Fonte 4.3: Coleção de Testes do Programa Ilustrado na Figura 4.1

```
1  public class Teste {
2    A a = new A();
3    B b = new B();
4    C c = new C();
5
6    test1(){assertTrue(a.k(2) == 1); assertTrue(a.k(-1) == 1);}
7    test2(){assertTrue(a.foo() == 5);}
8    test3(){assertTrue(b.bar() == 0);}
9    test4(){assertTrue(c.bar() == 5);}
10   test5(){assertTrue(b.foo() == 5);}
11 }
```

A seguir, na Seção 4.1, descrevemos em detalhes cada passo da nossa abordagem e na Seção 4.2, descrevemos SafeRefactorImpact, a ferramenta que implementa a abordagem proposta.

4.1 Abordagem

Nesta seção descrevemos como funciona a abordagem para avaliar refatoramentos proposta neste trabalho. Iniciamos a seção com uma visão geral (Seção 4.1.1). Em seguida, descrevemos em detalhes cada etapa da abordagem proposta nas Seções 4.1.2 a 4.1.5. Por fim, mostramos na Seção 4.1.6 um exemplo de aplicação da abordagem.

4.1.1 Visão Geral

A abordagem proposta analisa o programa original e o programa modificado e reporta se a transformação mudou o comportamento do programa. A Figura 4.2 ilustra os 4 passos que são executados de forma sequencial pela abordagem. O Primeiro passo é realizar uma análise estática do código para identificar os métodos impactados que são comuns às duas versões do programa. Safira é utilizada para identificar os métodos impactados pela transformação (Passo 1.1). Dos métodos impactados, são identificados os métodos públicos e em comum (Passo 1.2). No Passo 2 é gerada automaticamente uma coleção de testes considerando apenas o conjunto de métodos identificado no Passo 1. A coleção é executada no programa

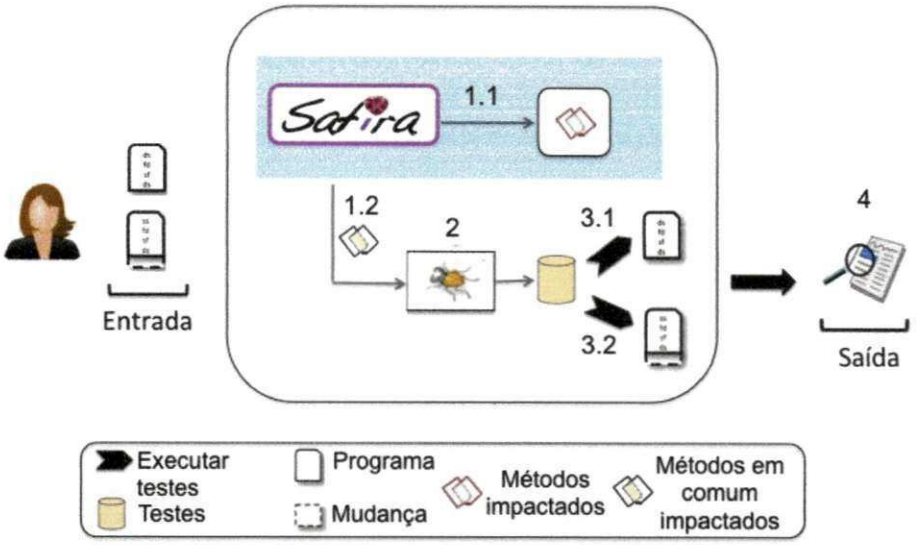


Figura 4.2: Abordagem para detectar mudanças comportamentais baseada em geração automática de testes e análise de impacto da mudança.

original (Passo 3.1) e no programa modificado pela transformação (Passo 3.2). Por fim, no Passo 4, os resultados são comparados para informar ao usuário se houve mudança de comportamento. A seguir, detalhamos cada fase da abordagem. Ao final, mostramos um exemplo prático de sua utilização para detectar uma mudança comportamental.

4.1.2 Primeira Fase: Análise de Impacto da Mudança

Como vimos, para avaliar se uma modificação realizada no programa preservou seu comportamento, é necessário analisar apenas os métodos que possam ter mudado de comportamento. Para isso, utilizamos Safira, nosso analisador de impacto da mudança, que identifica métodos que foram modificados ou apenas impactados pela mudança. Observe que os métodos que não foram impactados não podem ter seu comportamento modificado e consequentemente, não são úteis para avaliar se houve preservação de comportamento.

Safira analisa as duas versões do programa (o original e o refatorado), e tenta identificar os métodos impactados pela mudança. Para isso, Safira decompõe a transformação em um conjunto de subtransformações. Para cada subtransformação ela utiliza leis da análise de impacto para identificar os métodos diretamente impactados. O conjunto de métodos impactados será a união dos métodos impactados por cada subtransformação. Além disso, também

consideramos como impactados os métodos que exercitam direta ou indiretamente os métodos do conjunto impactado em qualquer nível de indireção. A análise de impacto realizada por Safira está descrita com um nível maior de detalhes no Capítulo 3 deste trabalho.

Ao ser realizada a análise de impacto, teremos métodos que possam ter mudado de comportamento após a mudança. A próxima etapa é testar esses métodos. Para isso, é necessário selecionar, do conjunto de métodos impactados, apenas os métodos que são públicos e comuns às duas versões do programa. O método a ser testado necessita ser público para ser acessível nas classes de testes. Ele também precisa ser um método em comum porque utilizamos a mesma coleção de testes para ser executada nas duas versões do programa. Suponha que seja criado um caso de teste para um método que só existe no programa original. Desta forma, não é possível testar o programa refatorado porque a coleção de testes não vai compilar. Os métodos públicos e em comum impactados serão passados como entrada para a fase de geração de testes.

4.1.3 Segunda Fase: Geração de Testes

Nesta fase da abordagem, é gerada automaticamente uma coleção testes de unidade apenas para os métodos comuns e impactados pela transformação, que foram identificados na primeira fase da abordagem. Para isso, é utilizado o Randoop [53], um gerador automático de testes de unidade que identificou *bugs* nas coleções de Java. O Randoop gera aleatoriamente uma coleção de testes dentro de um tempo limite informado pelo usuário considerando apenas os métodos recebidos como parâmetro. A saída desta fase é a coleção de testes gerada.

4.1.4 Terceira Fase: Execução dos Testes

A coleção de testes gerada pelo Randoop é executada, utilizando o framework JUnit, no programa original. Alguns casos de testes podem falhar nessa versão do programa. Isso pode ocorrer devido a algum *bug* já existente no programa e detectado pelo Randoop. Nesse caso, não tem relação com o refatoramento aplicado. Também é possível que isso aconteça em programas concorrentes, que não estão no escopo desta pesquisa. Por isso, os casos de testes gerados que falharem no programa original são desconsiderados na análise dos resultados.

Após a coleção de testes ser executada no programa original, ela será executada no programa refatorado. Observe que a mesma coleção de testes será executada nas duas versões do programa. Ou seja, os testes não sofreram mudanças durante o refatoramento e, com isso, temos uma maior confiança na análise da transformação.

4.1.5 Quarta Fase: Avaliação dos Resultados

Esta última fase da abordagem analisa os resultados dos testes executados no programa original e no programa refatorado. Não consideramos para avaliar a transformação os casos de testes que falharem no programa original, pois indicam que pelo menos algum método testado possui comportamento não determinístico. Desta forma não é útil para avaliar preservação de comportamento utilizando a nossa abordagem. Se algum caso de teste falhar apenas no programa modificado concluímos que a transformação não preserva o comportamento do programa, ou seja, que a transformação não é um refatoramento. Neste caso, mostramos os casos de testes que falharam apenas na versão modificada para ajudar na depuração do código. Se todos os casos de testes passarem nas duas versões, então podemos ter uma confiança maior que a transformação preserva o comportamento do programa.

4.1.6 Exemplo

Nesta seção mostramos como nossa abordagem detecta mudanças comportamentais utilizando um exemplo de um refatoramento automatizado que muda o comportamento do programa. Considere o Código Fonte 4.4 que descreve um programa em Java composto pela classe A e sua subclasse B. Neste programa o método $teste$ retorna 10. Quando aplicamos o refatoramento *Pull Up Method* do Eclipse 3.4.2 movendo o método m para a classe A obtemos o programa modificado mostrado no Código Fonte 4.5. Observe que o método $teste$ no programa modificado retorna 20 diferentemente do programa original. Ou seja, o comportamento foi modificado. Portanto, esta transformação aplicada pelo Eclipse não é um refatoramento. A seguir, descrevemos a aplicação de todos os passos da nossa abordagem na análise dessa transformação.

Figura 4.3: Pull Up Method aplicado pelo Eclipse 3.4.2 que não preserva comportamento

Código Fonte 4.4: Programa Original

```
1  public class A {
2    public int k(long l) {
3      return 10;
4    }
5    private int k(int l) {
6      return 20;
7    }
8  }
9  public class B extends A {
10   public int m() {
11     return k(2);
12   }
13   public int teste() {
14     return m();
15   }
16 }
```

Código Fonte 4.5: Programa Modificado

```
1  public class A {
2    public int k(long l) {
3      return 10;
4    }
5    private int k(int l) {
6      return 20;
7    }
8    public int m() {
9      return k(2);
10   }
11 }
12 public class B extends A {
13   public int teste() {
14     return m();
15   }
16 }
```

Análise de impacto

Para avaliar a transformação, o primeiro passo é realizar a análise de impacto da mudança. Nesta análise a transformação é decomposta em subtransformações e são identificados os métodos impactados por cada uma. Identificamos duas subtransformações: DM (deletar o método m da classe B) e AM (adicionar o método m na classe A). Para calcular o impacto de cada subtransformação, utilizamos as leis da análise de impacto descritas na Seção 3.2.3. O impacto da subtransformação DM ($B.m()$) é apenas o próprio método removido. Para subtransformação AM ($A.m()$) temos como impactados o método adicionado ($A.m()$) e o método herdado, $B.m()$. O conjunto impactado é a união do conjunto impactado por cada subtransformação. Neste exemplo, ele é formado pelos métodos $A.m()$ e $B.m()$. Também são considerados impactados os métodos que exercitam direta ou indiretamente pelo menos um método diretamente impactado. Então, identificamos o método $B.teste$ que exercita os dois métodos impactados. Logo, os métodos impactados pela transformação são: $B.m$, $A.m$ e $B.teste$. Dos métodos impactados, são identificados os métodos públicos e em comum: $B.m()$ e $B.teste()$. Observe que o método $B.m()$ é comum às duas versões do programa porque no programa modificado a classe B herda o método $m()$ da classe A. Logo, ele é considerado um método em comum. O método $A.m$ não é comum porque ele foi adicionado ao programa pela transformação e, portanto, não existe no programa original.

Testes e Análise do Resultado

A análise de impacto realizada na primeira fase da abordagem identificou 2 métodos públicos e em comum: $B.m()$ e $B.teste()$. O Randoop gera uma coleção de testes apenas para esses dois métodos. Um dos casos de testes gerados pode ser visto no Código 4.6. As asserções são verdadeiras para o programa original, mas são falsas para o programa modificado, pois os métodos vão retornar 20, ao invés de 10. Então, esse caso de teste vai passar no programa original, mas vai falhar no programa modificado. Por isso, é concluído que a transformação não preserva o comportamento do programa.

Código Fonte 4.6: Teste gerado pelo Randoop

```
1  public void test1() throws Throwable {
2          if (debug) {
3                  System.out.printf("\%nRandoopTest0.test1");
4          }
5          B var0 = new B();
6          int var1 = var0.teste();
7          int var2 = var0.m();
8
9          assertTrue(var1 == 10);
10         assertTrue(var2 == 10);
11 }
```

Cobertura da Mudança

Nós definimos neste trabalho a métrica cobertura da mudança como critério de adequação dos testes. O cálculo desta métrica está explicada na Seção 3.3. Vamos calcular a cobertura da mudança dos testes gerados neste exemplo. Os métodos impactados são: $A.m()$, $B.m()$ e $B.teste()$. Desses métodos, foram gerados casos de testes para os métodos $B.m()$ e $B.teste()$. No entanto, os testes exercitam o método $A.m()$ quando são executados no programa modificado. Então, a cobertura da mudança a nível de métodos vai ser: $C = \frac{3}{3} = 100\%$.

4.2 SafeRefactorImpact

Nesta seção nós apresentamos SafeRefactorImpact, uma ferramenta para detectar mudanças comportamentais em refatoramentos de programas Java, utilizando análise de impacto da mudança. Ela pode ser utilizada através de uma interface de linha de comando, que recebe como entrada duas versões de um programa Java (o original e o refatorado) e checa se o refatoramento introduziu alguma mudança comportamental. Além disso, os testes gerados para avaliar o refatoramento também podem ser disponibilizados para o usuário.

Código Fonte 4.7: Exemplo de como utilizar SafeRefactorImpact para analisar uma transformação

```
1  public static void main(String[] args) {
2
3      String original = "caminho do programa original";
4      String modificado = "caminho do programa modificado";
5      String lib = "";
6      String tempoLimite = "0.2";
7      String testes = "diretório para gravar os testes";
8
9      SafeRefactorImpact sri = new SafeRefactorImpact(original, modificado, lib
           , tempoLimite, testes);
10
11 }
```

Os parâmetros *original* e *modificado* são os caminhos para o programa original e o programa modificado, respectivamente. Se o programa analisado depender de algumas bibliotecas, o parâmetro *lib* é passado com o nome da pasta que estiverem essas dependências. Comumente, esta pasta é chamada de lib. O tempo limite de geração de testes, em segundos, é informado no parâmetro *tempoLimite*. O usuário pode ter acesso aos testes gerados para avaliar a transformação. Para isso, ele informa o diretório desejado para gravar os testes através do parâmetro *testes*. Se não for passada essa informação, os testes serão gravados em uma pasta temporária.

A ferramenta reporta a quantidade de casos de testes gerados, quantidade de casos de testes que falharam, se a transformação é um refatoramento e tempo total da análise em segundos. O SafeRefactorImpact identificou a mudança comportamental da transformação da Figura 4.3 em 4 segundos. Foram gerados 19 casos de testes e todos falharam no programa refatorado incorretamente. Isso aconteceu porque todos os métodos utilizados na geração de testes foram impactados e mudaram de comportamento.

4.2.1 Arquitetura

O SafeRefactorImpact foi implementado na linguagem de programação Java. Modificamos a ferramenta SafeRefactor [68] para substituir sua análise estática de identificar os métodos para geração de testes, pela análise de impacto de Safira, que identifica apenas métodos im-

pactados pela mudança. Todos os passos de SafeRefactorImpact são feitos automaticamente e de forma sequencial.

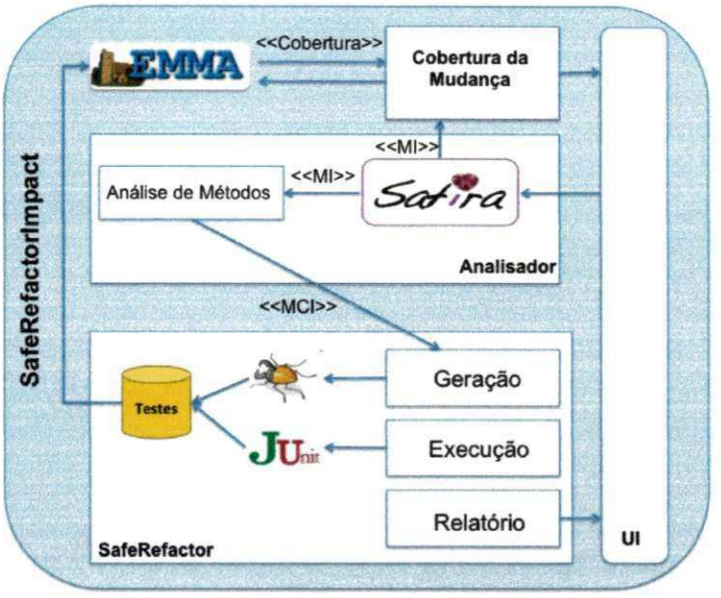


Figura 4.4: Arquitetura do SafeRefactorImpact

O SafeRefactorImpact contém 4 módulos: *UI*, *Analisador*, *SafeRefactor*, e *Cobertura da Mudança*. O módulo *UI* é responsável pela comunicação da ferramenta com o usuário. Como foi mostrado, a interface é de linha de comando mas é extensível para implementação de uma interface gráfica. O módulo *Analisador* tem dois componentes: *Safira*, que identifica métodos impactados pela transformação (MI) e *Analise de Métodos* que seleciona apenas os métodos impactados que são comuns às duas versões do programa (MCI). O módulo *SafeRefactor* utiliza os componentes *Geração*, *Execução* e *Relatório* da ferramenta SafeRefactor [68]. Ele recebe do módulo *Analisador* métodos em comum impactados e gera automaticamente uma coleção de testes utilizando o Randoop, apenas para esses métodos (*Geração*). Os métodos são executados nas duas versões do programa (*Execução*) e o resultado é analisado pelo componente *Relatório*. O módulo Cobertura da Mudança utiliza os métodos impactados identificados por *Safira* e os testes gerados pelo módulo *SafeRefactor* para calcular a cobertura da mudança dos métodos impactados. Para identificar os métodos exercitados pelos testes, é utilizada a ferramenta EMMA[1]. A Figura 4.4 ilustra a arquitetura

[1] http://emma.sourceforge.net/

do SafeRefactorImpact.

4.3 Limitações

Nesta seção, descrevemos algumas limitações da nossa abordagem. Algumas limitações estão relacionadas com o analisador de impacto, Safira, utilizado na abordagem. A ferramenta possui algumas limitações na análise e pode não identificar todos os métodos impactados por uma mudança no programa. Por exemplo, a análise de Safira não analisa mudanças em classes anônimas, inicializadores e classes de bibliotecas. Então, os métodos impactados por esses tipos de mudanças podem não ser considerados na geração de testes, e consequentemente, a abordagem pode não identificar alguma mudança comportamental.

Outras limitações da abordagem estão relacionadas com o gerador de testes, Randoop. Ele também possui limitações que dificultam a geração de testes para alguns métodos identificados na análise de impacto. Por exemplo, se o método depende de alguma classe de biblioteca, o Randoop só gera testes para esse método se algum outro método que está sendo testado retornar um objeto desta mesma classe. O Randoop também não é eficaz é gerar testes para métodos de interface gráfica do usuário e métodos que manipulam arquivos. Desta forma, os testes gerados podem não ser adequados para avaliar o programa. Além disso, diferenças nas mensagens da saída padrão (*System.out.println*) não são detectadas. Por fim, nossa abordagem não é adequada para avaliar transformações em programas concorrentes, pois o resultado de um método pode ser não determinístico. Desta forma, o teste gerado para o programa original, pode falhar no mesmo programa quando for executado.

Capítulo 5

Avaliação

Neste capítulo, apresentamos os experimentos realizados para avaliar nossa abordagem para identificar mudanças comportamentais em atividades de refatoramentos (Capítulo 4). Avaliamos a abordagem proposta em dois conjuntos. Um conjunto é composto por transformações aplicadas pelos desenvolvedores em um sistema real, o JHotDraw (Seção 5.1). O outro conjunto contém transformações aplicadas por ferramentas de refatoração, onde avaliamos 10 implementações do Eclipse e 8 do JRRT [61] (Seção 5.2). Na Seção 5.3 realizamos um estudo das ameaças a validade dos nossos experimentos.

5.1 JHotDraw

Nesta seção apresentamos o experimento realizado para avaliar o SafeRefactorImpact em um sistema real. Nós avaliamos transformações em 10 versões do JHotDraw, um framework para desenvolvimento de editores gráficos, aplicadas pelos desenvolvedores. Inicialmente, definimos o experimento fornecendo uma visão geral (Seção 5.1.1). Em seguida, descrevemos seu planejamento (Seção 5.1.2) e as configurações experimentais (Seção 5.1.3). Por fim, apresentamos e discutimos os resultados nas Seções 5.1.4 e 5.1.5, respectivamente.

5.1.1 Definição

Na fase de definição, descrevemos a intenção da realização do experimento e os aspectos de qualidade que estão sendo estudados. Também apresentamos as questões de pesquisa que

guiaram o experimento e as métricas utilizadas para avaliar a nossa abordagem.

Objetivo do Estudo

O objetivo deste estudo é analisar o SafeRefactorImpact e o SafeRefactor com o propósito de avaliar o benefício de incorporar a análise de impacto da mudança na abordagem original do SafeRefactor. Avaliamos as abordagens com respeito a equivalência dos resultados e custo de análise, do ponto de vista de desenvolvedores de software que aplicam refatoramentos e no contexto de um sistema real com milhares de linhas de código. O SafeRefactor implementa uma abordagem para detectar mudanças comportamentais em refatoramentos de programas. Ele gera uma coleção de testes para todos os métodos públicos e em comum do programa, dentro de um tempo limite informado pelo usuário. Os casos de testes gerados são executados no programa original e no programa refatorado. Se houver algum resultado diferente é concluído que existe uma mudança comportamental e que o refatoramento não está correto. O SafeRefactorImpact gera uma coleção de testes apenas para métodos impactados pela mudança.

A principal diferença entre as duas abordagens é o conjunto de métodos identificados para geração de testes. No pior caso, os conjuntos vão ser iguais se a mudança impactar todos os métodos do programa. Em programas de milhares de linhas de código é mais difícil dessa situação ocorrer. Por isso, queremos investigar neste estudo a diferença na quantidade de métodos identificados por cada abordagem.

O principal critério a ser avaliado é o resultado da análise das abordagens. Vamos comparar as mudanças comportamentais identificadas e o tempo limite de geração de testes necessário para identificar a mudança. Outro critério importante é o custo de cada abordagem com relação ao tempo para avaliar uma transformação. A análise estática do SafeRefactorImpact é mais custosa pois realiza a análise de impacto. Queremos analisar se esse custo adicional é muito grande ou se ele é compensado no tempo de geração de testes. Por isso, também precisamos avaliar a quantidade de casos de testes gerados. Quanto menos casos de testes forem gerados, menor vai ser o custo na compilação e execução dos testes. Por fim, também analisaremos a cobertura da mudança da coleção de testes gerada para avaliar se ao gerar testes apenas para o que foi impactado cobre mais a mudança do que gerar testes para todo o programa.

Questões de Pesquisa

Para conduzir o estudo de acordo com os objetivos discutidos na seção anterior, definimos as seguintes questões de pesquisa:

Q 1 *O SafeRefactorImpact tem um resultado equivalente ao SafeRefactor?*

Q 2 *A análise de impacto do SafeRefactorImpact diminui a quantidade de métodos identificados para geração de testes em relação ao SafeRefactor?*

Q 3 *O SafeRefactorImpact gera menos casos de testes do que o SafeRefactor?*

Q 4 *O tempo total da análise é menor no SafeRefactorImpact?*

Q 5 *A cobertura da mudança dos testes gerados pelo SafeRefactorImpact é maior do que nos testes gerados pelo SafeRefactor?*

Métricas

As métricas utilizadas para ajudar a responder as questões de pesquisa e assim, medir os aspectos de qualidade estudados neste experimento, são:

- **Acurácia, Precisão e Revocação**

  Para responder a questão de pesquisa Q 1, nós medimos três métricas nos resultados de cada abordagem: acurácia (*Accuracy*), precisão (*Precision*) e revocação (*Recall*). Para calcular essas métricas, vamos definir os seguintes termos:

  - **Verdadeiro-positivo (*vPos*)** : análise correta de que uma transformação preserva o comportamento
  - **Verdadeiro-negativo (*vNeg*)** : análise correta de que uma transformação não preserva o comportamento
  - **Falso-positivo (*fPos*)** : análise incorreta de que uma transformação preserva o comportamento
  - **Falso-negativo (*fNeg*)** : análise incorreta de que uma transformação não preserva o comportamento

Acurácia representa a porcentagem de transformações avaliadas corretamente. Ela é calculada por:

$$\textbf{Acurácia} = \frac{\#vPos+\#vNeg}{\#vPos+\#vNeg+\#fPos+\#fNeg}$$

Precisão representa a porcentagem de análises que indicam refatoramentos que estão corretas. Ela é calculada por:

$$\textbf{Precisão} = \frac{\#vPos}{\#vPos+\#fPos}$$

Revocação representa a porcentagem de transformações que preservam o comportamento, avaliadas corretamente. Ela é calculada por:

$$\textbf{Revocação} = \frac{\#vPos}{\#vPos+\#fNeg}$$

- **Quantidade de Métodos**

  Para responder a questão de pesquisa Q 2, precisamos medir a quantidade de métodos identificados pela análise estática de cada abordagem. Esse conjunto de método identificados será utilizado para geração de testes. A quantidade de métodos é reportada por cada ferramenta ao ser executada.

- **Quantidade de Testes**

  Essa métrica é medida para responder a questão de pesquisa Q 3. As abordagens geram uma coleção de testes, dentro de um tempo limite informado pelo usuário, para os métodos do programa identificados na análise estática. Esta métrica mede a quantidade de casos de testes gerados por cada abordagem. Um caso de teste é representado por um método de teste em uma classe de testes. Cada método de teste exercita pelo menos um método do programa.

- **Tempo**

  Esta métrica mede o tempo total que cada abordagem leva para analisar um refatoramento. Ela é utilizada para responder a questão de pesquisa Q 4. Descrevemos abaixo como esse tempo é medido:

  Tempo Total = tempo da análise estática + tempo de geração dos testes + tempo da compilação dos testes + tempo de execução dos testes + tempo da análise dos resultados

A análise estática do SafeRefactorImpact é a análise de impacto da mudança realizada por Safira e a do SafeRefactor é a análise do código para identificar os métodos em comum entre as duas versões do programa. O tempo de geração dos testes é o tempo limite passado como parâmetro para o Randoop gerar os testes. Todas essas etapas são feitas de forma automática pelas ferramentas. O tempo total é medido pelo tempo de análise de toda a transformação, que inclui essas etapas.

- **Cobertura da Mudança**

  Esta métrica é utilizada para responder a questão de pesquisa Q 5. Ela mede a cobertura da mudança dos testes gerados a nível de métodos impactados. A seguir, mostramos como é calculada a cobertura da mudança:

  Seja I o conjunto de métodos impactados e E o conjunto de métodos impactados exercitados pelos testes. Onde $E \subset I$. A cobertura da mudança a nível de métodos C, é definida como $C = \frac{\#E}{\#I}$

5.1.2 Planejamento

Nesta seção, descrevemos o planejamento deste experimento. Primeiramente, mostramos como foi realizada a seleção das transformações, incluindo os critérios de seleção e características de cada transformação escolhida. Em seguida, apresentamos as variáveis do experimento, que incluem fatores e variáveis de resposta e por fim, exibimos o design do experimento. A execução do experimento seguiu este planejamento.

Seleção das Transformações

Em um trabalho anterior, Soares et al. [67] avaliou o SafeRefactor em 40 pares de versões do repositório do JHotDraw selecionados aleatoriamente. Para cada versão selecionada, foi utilizada a versão anterior para formar um par. Deste conjunto de 40 transformações, nós selecionamos 10 pares de versões. Nosso critério foi selecionar transformações com diferentes características: granularidade da mudança, escopo e tamanho do programa. A Tabela 5.1 exibe algumas características das transformações selecionadas para realização deste experimento. Nós priorizamos selecionar mais transformações que não são refatoramentos para avaliar o SafeRefactorImpact na detecção de mudanças comportamentais.

A primeira coluna representa o número da versão do JHotDraw selecionada. A segunda coluna exibe o tamanho do programa em linhas de código. O tamanho foi medido da versão selecionada (modificada pela transformação). A granularidade está exposta na terceira coluna da tabela. Refatoramentos de granularidade alta são transformações que afetam assinaturas de classes e métodos, incluindo atributos de classe. Por exemplo, Extrair Método, Renomear Método e Adicionar Parâmetro. Por outro lado, refatoramentos de granularidade baixa, como Renomear Variável Local, e Extrair Variável Local, modificam apenas o corpo do método. Na quarta coluna, mostramos o escopo da transformação, que pode ser local ou global. Uma mudança local, atinge apenas uma classe ou pacote. Uma mudança global atinge mais partes do sistema. Por fim, na última coluna, mostramos o *baseline* que indica se a transformação é ou não um refatoramento. Esse *baseline* é o mesmo utilizado no estudo de Soares et al. [67].

Caracteristicas das Transformações do JHotDraw				
Versão	**LOC**	**Granularidade**	**Escopo**	**Refatoramento**
134	20422	Baixa	Local	Não
173	28052	Baixa	Global	Não
176	28055	Baixa	Local	Não
193	28298	Baixa	Global	Sim
322	39480	Alta	Local	Não
324	39551	Alta	Global	Não
344	51596	Alta	Global	Não
596	72553	Alta	Global	Não
650	76220	Alta	Global	Não
700	79741	Baixa	Global	Não

Tabela 5.1: Características das versões do JHotDraw avaliadas no experimento.

Fatores e Variáveis de Resposta

Em um experimento temos dois tipos de variáveis: as variáveis independentes e dependentes. As variáveis independentes, também chamadas de fatores, representam a entrada do experimento, elas apresentam a causa que afeta o resultado final. Essas variáveis são combinadas

de várias formas com o intuito de avaliar sua relação com a saída do experimento. As variáveis dependentes, ou variáveis de resposta, apresentam o efeito dos fatores escolhidos em uma execução, ou seja, são a saída do experimento.

Os fatores deste experimento são:

- **Abordagem:** SafeRefactorImpact e SafeRefactor
- **Tempo limite de geração de testes:** 1s, 5s, 60s e 120s

Nós escolhemos valores baixos (1s e 5s), intermediários (60s) e altos (120s) de tempo limite para avaliar as variáveis de resposta em cada abordagem. Consideramos o tempo limite de 120s alto porque ele foi considerado o tempo adequado pelo SafeRefactor [67] para avaliar as transformações do JHotDraw. As variáveis de resposta são:

- Resultado da análise: se é ou não um refatoramento
- Quantidade de métodos
- Quantidade de testes gerados
- Tempo total da análise
- Cobertura da mudança

Design do Experimento

Nós escolhemos um design fatorial completo, onde vamos executar cada abordagem com cada tempo limite de geração de testes, para medir as variáveis de resposta para cada tratamento. Temos um total de 8 execuções para cada versão do JHotDraw, totalizando 80 execuções. A Tabela 5.2 ilustra todas as execuções para cada transformação.

5.1.3 Configurações Experimentais

Nós automatizamos o experimento para coletar as variáveis de resposta do SafeRefactorImpact e do SafeRefactor. O experimento foi executado em um computador com Processador Intel Core i5 (2.3GHz), 4GB de memória RAM e sistema operacional Mac OS 10.6. Utilizamos o SafeRefactor versão linha de comando 1.1.4. Tanto o SafeRefactor quanto o

Abordagem							
SRImpact				SafeRefactor			
Tempo Limite				Tempo Limite			
1s	5s	60s	120s	1s	5s	60s	120s

Tabela 5.2: Design fatorial completo utilizado no experimento do JHotDraw. Cada abordagem é executada uma vez utilizando cada tempo limite.

SafeRefactorImpact utilizam o Randoop [53], versão 1.3.2. para gerar os testes. O Eclipse versão Helios Service Release 2 e JDK 1.6. foram utilizados para implementar e executar o experimento. O tempo foi medido pela função System.currentTimeMillis() da biblioteca de Java.

Nós automatizamos o cálculo da cobertura da mudança. Foi implementado um módulo a parte no SafeRefactorImpact para calcular essa métrica. Esse módulo primeiramente, calcula a cobertura (a nível de métodos) dos testes gerados por cada abordagem no programa utilizando a ferramenta EMMA[1] versão 2.1. Em seguida, ele utiliza Safira (Capítulo 3) para identificar os métodos impactados e analisa o resultado de EMMA para identificar os métodos impactados que são cobertos pelos testes.

5.1.4 Resultados

Nesta seção, nós descrevemos os resultados da nossa avaliação nas transformações aplicadas em versões JHotDraw. Primeiramente, discutimos os resultados das métricas acurácia, precisão e revocação e da redução dos métodos identificados por cada abordagem para geração de testes. Em seguida, mostramos os resultados das outras métricas para cada tempo limite de geração de testes utilizado neste experimento.

Acurácia, Precisão e Revocação

Nós calculamos as métricas acurácia, precisão e revocação para cada transformação do JHotDraw avaliada com 1s, 5s, 60s e 120s de tempo limite de geração de testes. O cálculo se

[1] http://emma.sourceforge.net/

baseou no *baseline* mostrado na Tabela 5.1. Os resultados das métricas estão exibidos na Tabela 5.3. Quanto maior o tempo limite, maior a chance de analisar corretamente a transformação. Então, tanto a acurácia quanto a precisão foram maiores com 60s e 120s. A acurácia do SafeRefactorImpact foi 0,7 com 120s de tempo limite de geração de testes. Esse resultado indica que 7 transformações foram avaliadas corretamente, de um total de 10 transformações. Já o SafeRefactor avaliou 4 transformações corretamente com o mesmo tempo limite, logo sua acurácia foi 0,4. Com 1s o SafeRefactorImpact avaliou 2 transformações corretamente, enquanto SafeRefactor avaliou 1 transformação corretamente.

JHotDraw								
Versão	**1s**		**5s**		**60s**		**120s**	
	SRI	**SR**	**SRI**	**SR**	**SRI**	**SR**	**SRI**	**SR**
134	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos
173	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos
176	Fpos	Fpos	Fpos	Fpos	**Vneg**	**Vneg**	**Vneg**	**Vneg**
193	**Vpos**	**Vpos**	**Vpos**	**Vpos**	**Vpos**	**Vpos**	**Vpos**	**Vpos**
322	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos
324	Fpos	Fpos	Fpos	Fpos	**Vneg**	Fpos	**Vneg**	**Vneg**
344	Fpos	Fpos	Fpos	Fpos	**Vneg**	Fpos	**Vneg**	Fpos
596	**Vneg**	Fpos	**Vneg**	Fpos	**Vneg**	**Vneg**	**Vneg**	**Vneg**
650	Fpos	Fpos	Fpos	Fpos	**Vneg**	Fpos	**Vneg**	Fpos
700	Fpos	Fpos	Fpos	Fpos	Fpos	Fpos	**Vneg**	Fpos
Acurácia	**0,2**	**0,1**	**0,2**	**0,1**	**0,6**	**0,3**	**0,7**	**0,4**
Precisão	**0,11**	**0,1**	**0,11**	**0,1**	**0,2**	**0,12**	**0,25**	**0,14**
Revocação	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Tabela 5.3: Métricas acurácia, precisão e revocação calculadas para cada transformação avaliada do JHotDraw.

A precisão foi baixa porque apenas uma transformação é refatoramento (versão 193). Além disso, o SafeRefactorImpact avaliou incorretamente, utilizando 120s de tempo limite de geração de testes, 3 transformações que não são refatoramentos, resultado em uma precisão de 0,25. Já o SafeRefactor avaliou, com o mesmo tempo limite, 6 transformações que não são refatoramentos incorretamente. Sua precisão foi 0,14. A revocação foi 1 (valor máximo)

para as duas abordagens em todos os tempos limites de geração de testes. Esse resultado é explicado por existir apenas um refatoramento, onde este é avaliado corretamente em todas as análises.

Métodos identificados para geração de testes

A Tabela 5.4 exibe a quantidade de métodos identificados na análise estática do SafeRefactorImpact (segunda coluna) e do SafeRefactor (terceira coluna). Na última coluna temos a redução de métodos do SafeRefactorImpact devido a análise de impacto. Ele reduziu em pelo menos 82% e em até 99,9%.

JHotDraw			
Versão	Métodos		Redução (%)
	SRI	SR	
134	58	16890	99,65
173	12	26567	99,95
176	2600	26567	90,21
193	444	26396	98,31
322	2151	12263	82,45
324	1693	12237	86,16
344	1773	18339	90,33
596	366	30433	98,79
650	83	30877	99,73
700	2461	34592	92,88

Tabela 5.4: Quantidade de métodos identificados pelo SafeRefactorImpact e SafeRefactor.

Análise com 1s de tempo limite de geração de testes

A Tabela 5.5 apresenta os resultados (tempo total, quantidade de testes, resultado e cobertura da mudança) da análise das transformações do JHotDraw para o tempo limite de geração de testes de 1s. Com esse tempo limite, a única mudança comportamental identificada foi na versão 596 pelo SafeRefactorImpact. O SafeRefactor não identificou. O SafeRefactorImpact gerou 49% menos casos de testes que o SafeRefactor e ainda assim teve uma cobertura da mudança 41% maior. Entretanto, o tempo total do SafeRefactor foi 64% menor.

JHotDraw com 1s de tempo limite de geração de testes								
Versão	Tempo (min)		Testes		É refatoramento?		Cobertura da mudança (%)	
	SRI	SR	SRI	SR	SRI	SR	SRI	SR
134	0,25	0,28	5	11	Sim	Sim	32	0
173	0,31	0,53	2	37	Sim	Sim	42	0
176	1	0,53	25	31	Sim	Sim	54	49
193	2,03	0,53	24	103	Sim	Sim	69	57
322	0,73	0,28	10	31	Sim	Sim	58	54
324	1,2	0,25	65	58	Sim	Sim	76	62
344	1,63	0,45	19	21	Sim	Sim	66	30
596	1,06	0,45	8	75	Não	Sim	41	17
650	1,06	0,43	30	80	Sim	Sim	47	1
700	2,51	0,46	63	49	Sim	Sim	69	53
Redução SRI (%)			49,39		-		41,69	
Redução SR (%)	64,43				-			

Tabela 5.5: Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 1s.

Análise com 5s de tempo limite de geração de testes

A Tabela 5.6 apresenta os resultados para o tempo limite de geração de testes de 5s. Com esse tempo limite, também foi identificada apenas uma mudança comportamental pelo SafeRefactorImpact (versão 596). O SafeRefactor também não identificou. O SafeRefactorImpact reduziu em 26% a quantidade de testes e teve uma cobertura da mudança 30% maior que o SafeRefactor. O tempo total do SafeRefactor foi 61% menor.

Análise com 60s de tempo limite de geração de testes

Os resultados para o tempo limite de geração de testes de 60s estão apresentados na Tabela 5.7. O SafeRefactorImpact e o SafeRefactor identificaram mudanças comportamentais nas versões 176 e 596. Nas versões 324, 344 e 650 apenas o SafeRefactorImpact identificou as mudanças. Com esse tempo limite, o SafeRefactorImpact reduziu em 16% a quantidade de

JHotDraw com 5s de tempo limite de geração de testes								
Versão	Tempo (min)		Testes		É refatoramento?		Cobertura da mudança (%)	
	SRI	SR	SRI	SR	SRI	SR	SRI	SR
134	0,35	0,33	337	180	Sim	Sim	32	0
173	0,36	0,60	2	160	Sim	Sim	42	0
176	0,90	0,6	178	190	Sim	Sim	74	67
193	2,26	0,58	154	466	Sim	Sim	84	70
322	0,68	0,41	187	194	Sim	Sim	77	73
324	1,23	0,36	274	158	Sim	Sim	80	67
344	1,53	0,43	175	66	Sim	Sim	78	60
596	1,25	2,15	116	277	Não	Sim	60	22
650	1,1	0,48	49	435	Sim	Sim	56	35
700	2,63	0,5	272	246	Sim	Sim	76	66
Redução SRI (%)			26,47		-		30,19	
Redução SR (%)	61,21				-			

Tabela 5.6: Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 5s.

testes, teve uma cobertura da mudança 17% maior e foi 7% mais rápido que o SafeRefactor.

Análise com 120s de tempo limite de geração de testes

Utilizando o tempo limite de geração de testes de 120s, o SafeRefactorImpact identificou mudanças comportamentais nas versões 344, 650 e 700, enquanto o SafeRefactor não identificou essas mudanças. O SafeRefactor identificou a mudança da versão 324 que o SafeRefactorImpact identificou com 60s. O SafeRefactorImpact gerou 36% menos casos de testes que o SafeRefactor, teve uma cobertura da mudança 16% maior e foi 10% mais rápido. Os resultados deste tempo limite estão apresentados na Tabela 5.8.

JHotDraw com 60s de tempo limite de geração de testes								
Versão	Tempo (min)		Testes		É refatoramento?		Cobertura da mudança (%)	
	SRI	SR	SRI	SR	SRI	SR	SRI	SR
134	1,68	1,55	2139	1106	Sim	Sim	32	0
173	1,3	2,03	2	1309	Sim	Sim	42	57
176	2,68	2,18	811	1729	Não	Não	79	74
193	4,01	2,03	1403	2587	Sim	Sim	84	76
322	1,6	2,1	544	601	Sim	Sim	80	80
324	2,66	1,9	1230	397	Não	Sim	81	73
344	3,3	3,6	1221	552	Não	Sim	82	60
596	2,35	8,68	563	1127	Não	Não	60	48
650	2,13	2,1	487	1007	Não	Sim	60	14
700	4,45	2,1	1333	1253	Sim	Sim	77	75
Redução SRI (%)	7,46		16,58		-		17,72	
Redução SR (%)					-			

Tabela 5.7: Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 60s.

5.1.5 Discussão

Nesta seção, vamos discutir os principais resultados deste experimento. Organizamos nossa discussão falando separadamente sobre cada métrica utilizada para avaliar as abordagens.

Equivalência dos Resultados da Análise

O resultado da análise do refatoramento é influenciada pelo tempo limite de geração de testes escolhido. Dependendo do tempo limite, os casos de testes gerados podem não exercitar uma mudança comportamental e assim, não identificá-la. Com a análise de impacto do SafeRefactorImpact, a quantidade de métodos para testar é menor, aumentando as chances da coleção de testes gerada exercitar a mudança. Além disso, é necessário um tempo limite menor para geração de testes. Então, o SafeRefactorImpact deve encontrar uma mudança comportamental, se houver, utilizando um tempo limite para geração de testes menor do que

JHotDraw com 120s de tempo limite de geração de testes								
Versão	Tempo (min)		Testes		É refatoramento?		Cobertura da mudança (%)	
	SRI	SR	SRI	SR	SRI	SR	SRI	SR
134	2,7	2,7	2100	1941	Sim	Sim	32	0
173	2,28	3,98	2	3024	Sim	Sim	42	42
176	4,53	3,88	1135	2995	Não	Não	79	70
193	5,23	3,36	1785	4310	Sim	Sim	84	75
322	3,1	3,3	1004	1071	Sim	Sim	80	80
324	4,9	4,01	1509	1004	Não	Não	81	73
344	5,41	8,11	1943	1004	Não	Sim	82	63
596	3,6	10,56	1007	1603	Não	Não	61	50
650	3,9	3,2	807	1468	Não	Sim	60	54
700	6,36	4,0	1958	2284	Não	Sim	77	77
Redução SRI (%)	10,80		36,00		-		16,00	
Redução SR (%)					-			

Tabela 5.8: Resultados das execuções de SafeRefactorImpact e SafeRefactor utilizando um tempo limite de geração de testes de 120s.

o necessário para o SafeRefactor encontrar a mesma mudança.

Utilizar 1s de tempo limite de geração de testes não é suficiente para testar programas de milhares de linhas de código. Ainda assim, o SafeRefactorImpact identificou a mudança comportamental da versão 596 com esse tempo, enquanto o SafeRefactor não identificou. Apesar da mudança ser global e de alta granularidade, SafeRefactorImpact identificou 366 métodos impactados, enquanto o SafeRefactor gera testes para mais de 30 mil métodos. Por isso, 1s não foi suficiente para testar todos esses métodos. Com 5s de tempo limite de geração de testes também não foi suficiente para identificar essa mudança. O SafeRefactor identifica a mudança utilizando 60s.

Na versão 173, SafeRefactor e SafeRefactorImpact não identificam a mudança com 120s de tempo limite de geração de testes. A mudança nessa versão adiciona um método e modifica o corpo de 2, resultando em 12 métodos impactados. Entretanto, 10 métodos dependem de uma classe de biblioteca. Então, o Randoop não gera testes para esses métodos. Tanto o

SafeRefactor quanto SafeRefactorImpact identificaram a mudança da versão 176 com 60s. Na transformação dessa versão, a única mudança foi no corpo de um método. No entanto, impactou 2600 métodos, pois esses métodos exercitam de maneira direta ou indireta ou herdam de métodos que exercitam o método modificado.

Na versão 700, o SafeRefactor não identificou uma mudança neste experimento mas identificou em um experimento anterior [67]. Nesta versão, os desenvolvedores mudaram algumas instruções para atribuir a cópia de um array ao invés do próprio array. Embora essa mudança tenha consertado a exibição do array, o Randoop não conseguiu detectar a mudança comportamental. A execução de alguns métodos ou sequência de métodos produz um resultado não determinístico. Além disso, como os testes são gerados de forma aleatória, o resultado pode ser diferente em cada execução. Esse foi o motivo pelo qual SafeRefactorImpact identificou que a transformação dessa versão não preserva o comportamento. Pelo mesmo motivo, o SafeRefactor identificou no experimento anterior.

Nas versões 344 e 650, SafeRefactorImpact identificou a mudança comportamental que o SafeRefactor não conseguiu identificar. Na versão 344 o corpo de um método do programa foi modificado e consequentemente seu comportamento também. Essa mudança não é muito fácil de ser identificada pelos testes porque para detectá-la é necessário executar uma sequência de dois métodos impactados, onde um muda o valor de um atributo utilizado pelo método modificado e o outro exercita o mesmo método. Por isso, focando a geração de testes apenas para os métodos impactados foi possível gerar casos de testes que exibiram a mudança. Na versão 650, também foi modificado o corpo de um método, mudando seu comportamento. SafeRefactorImpact identificou a mudança porque o Randoop gerou casos de testes para um método impactado que exercita indiretamente o método modificado.

Alguns métodos do programa contém a instrução $System.exit(0)$, que quando executada pára a execução corrente. Quando um caso de teste é gerado para esse método, a execução dos testes é finalizada ao executar o método. Logo, se existia algum caso de teste em seguida que exercitasse uma mudança comportamental, esta não seria identificada. Essa é uma limitação do SafeRefactor. O SafeRefactorImpact resolveu esse problema excluindo da geração de testes os métodos que contém ou exercitam essa instrução direta ou indiretamente. Ele analisa o corpo do método para procurar a instrução. Com isso, o SafeRefactorImpact pode identificar mudanças que o SafeRefactor não identifica por essa limitação.

Quantidade de Métodos

Na versão 176 foi identificada a maior quantidade de métodos impactados. Nesta versão, foram impactados 2600 métodos de mais de 26 mil métodos em comum. Apesar de baixa granularidade e escopo local, essa mudança impactou muitos métodos devido a grande dependência e relacionamentos dos métodos do programa com as entidades modificadas. Na versão 173, o SafeRefactorImpact identificou 12 métodos impactados de um total de mais de 26 mil métodos em comum. Foi a versão que teve a menor quantidade de métodos impactados. Na versão 344 a análise de impacto reduziu em mais de 90% a quantidade de métodos para geração de testes. Já na versão 650 ela reduziu em 99,7%,

Quantidade de Testes

O SafeRefactor gerou mais casos de testes que SafeRefactorImpact porque a quantidade de métodos para testar também é maior. O Randoop gera testes aleatoriamente dentro de um tempo limite, para os métodos passados por parâmetro. Em um caso de teste ele pode chamar uma sequência de métodos. No final da geração dos testes, o Randoop descarta os casos de testes redundantes. Então, com poucos métodos para testar é mais provável que o Randoop gere mais casos de testes redundantes e os descarte. Enquanto que, com muitos métodos para testar, é mais difícil que isso aconteça.

A Figura 5.9 mostra a redução na quantidade de casos de testes gerados nas transformações das versões 650 e 700 e na média de todas as transformações. As barras em azul significam que o SafeRefactorImpact reduziu a quantidade de casos de testes gerados em relação ao SafeRefactor. As barras em vermelho significam que o SafeRefactor gerou menos casos de testes. Em média, nossa abordagem reduziu a quantidade de casos de testes gerados em todos os tempos limites de geração de testes. Mas em algumas transformações, como por exemplo, da versão 700, nós geramos mais testes para alguns tempos limites. Isso aconteceu porque o Randoop possui algumas limitações, ele não lida muito bem com interfaces gráficas e manipulação de arquivos, o que tem muito no JHotDraw. Então, como o SafeRefactor identifica um conjunto maior de métodos para geração de testes, o Randoop não consegue ser muito eficiente, devido às suas limitações. Com o conjunto menor de métodos identificado pelo SafeRefactorImpact, o efeito das limitações do Randoop é minimizado, tornando-o mais

eficiente e gerando mais casos de testes em algumas transformações.

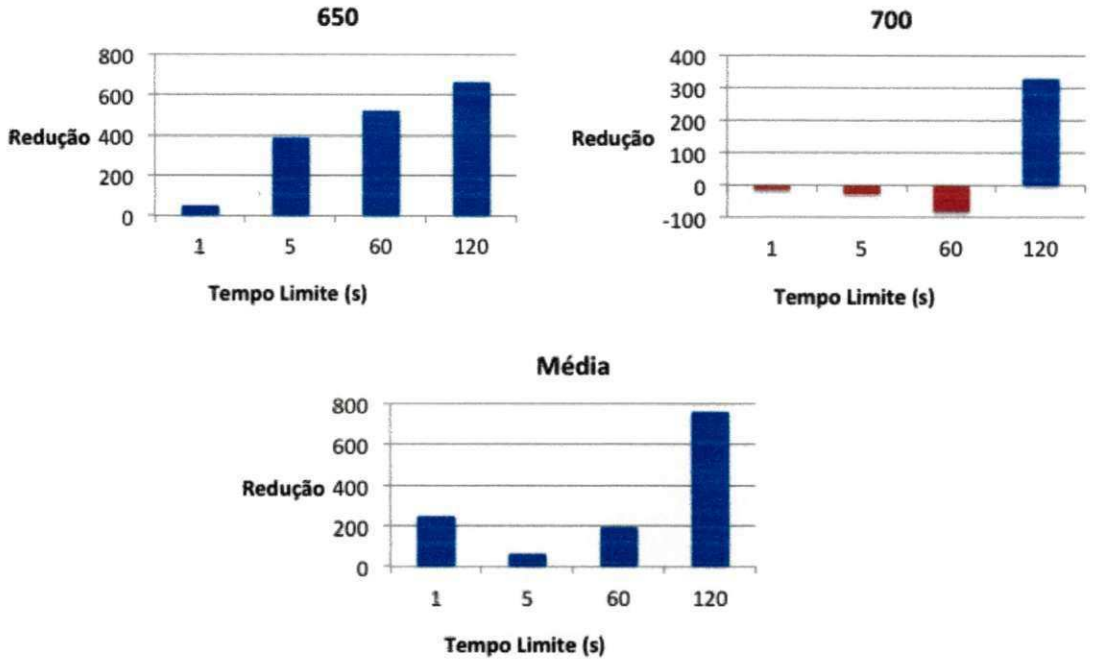


Tabela 5.9: Redução da quantidade de casos de testes gerados nas transformações das versões 650 e 700 e na média de todas as transformações.

Tempo

Os gráficos ilustrados na Figura 5.10 mostram as reduções do tempo total para avaliar as transformações das versões 344 e 650 e a redução média do tempo total para avaliar todas as transformações. As barras em azul significam que o SafeRefactorImpact reduziu o tempo total em relação ao SafeRefactor. As barras em vermelho significam que o SafeRefactor foi mais rápido.

Na média, o tempo total para avaliar as transformações foi menor no SafeRefactor para os tempos limites de geração de testes de 1s e 5s. Realizar a análise de impacto em programas grandes como o JHotDraw pode não ser rápida. Por exemplo, neste experimento, Safira demorou de alguns segundos até 2 minutos para analisar o impacto de cada transformação. Por isso, mesmo gerando testes para um conjunto menor de métodos, o tempo total para avaliar as transformações foi, em média, menor no SafeRefactor para os tempos limites de

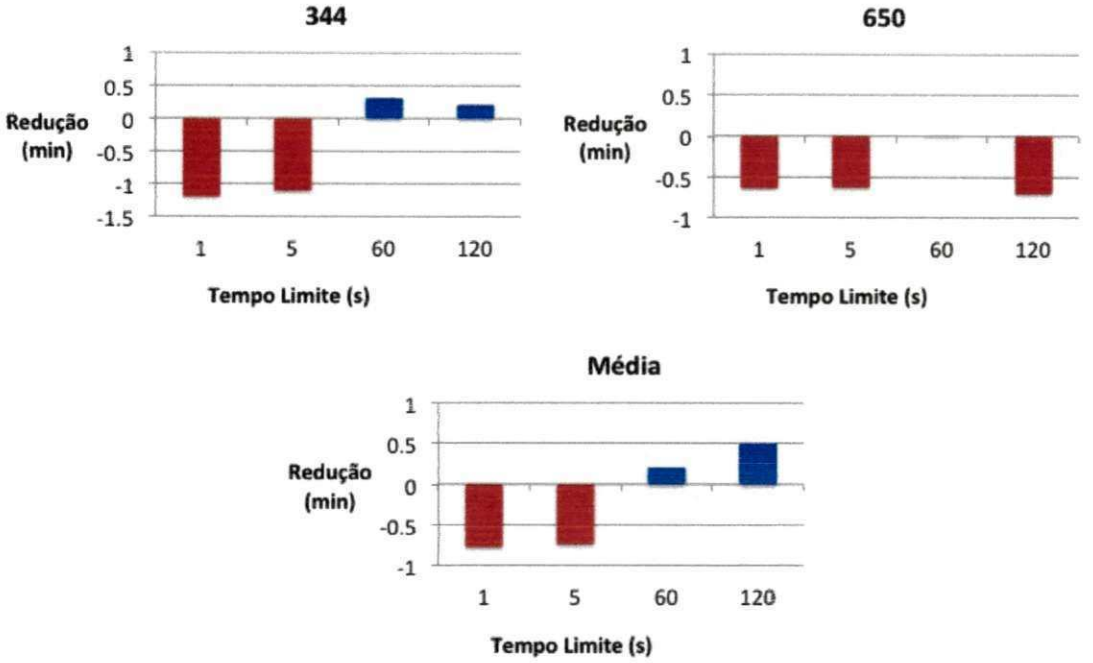


Tabela 5.10: Redução do tempo total para avaliar as transformações das versões 344 e 650 e a média de redução de todas as transformações.

geração de testes de 1s e 5s. Já com 60s e 120s de tempo limite, o SafeRefactorImpact foi em média mais rápido.

Em programas grandes, a tendência é que quanto maior o tempo limite de geração de testes utilizado, maior será a diferença entre o tempo total do SafeRefactorImpact e SafeRefactor. O tempo do SafeRefactorImpact tende a ser menor, pois o tempo gasto na análise de impacto pode ser compensado pelo tempo gasto na execução da coleção de testes, já que o SafeRefactor tende a gerar mais casos de testes pela quantidade de métodos para testar ser maior. Entretanto, em algumas transformações, como por exemplo, a da versão 650, o tempo total foi maior no SafeRefactorImpact para todos os tempos limites de geração de testes, devido a análise de impacto ser mais custosa que o tempo utilizado pelo SafeRefactor executar os testes. Ainda assim, esse custo adicional de alguns segundos no tempo total para avaliar a transformação, pode valer a pena por identificar a mudança comportamental que o SafeRefactor não identifica.

Não conseguimos observar uma relação do escopo e granularidade da transformação com o tempo de análise. É intuitivo que transformações grandes e globais sejam mais difíceis, e

consequentemente, mais demoradas de serem analisadas. Entretanto, não é uma regra. Este experimento mostrou que o fator mais relacionado com o tempo de análise é a complexidade e o grau de dependência das entidades do código. Um exemplo é a versão 176, que modificou apenas um método do programa, mas impactou 2600 métodos. O escopo dessa transformação é local e sua granularidade é baixa. Por outro lado, na versão 173 que o escopo é global, foram impactados apenas 12 métodos.

Cobertura da Mudança

Os gráficos ilustrados na Figura 5.11 representam o aumento da cobertura da mudança dos testes gerados pelo SafeRefactorImpact em relação aos gerados pelo SafeRefactor, nas transformações das versões 650 e 700, e na média de todas as transformações.

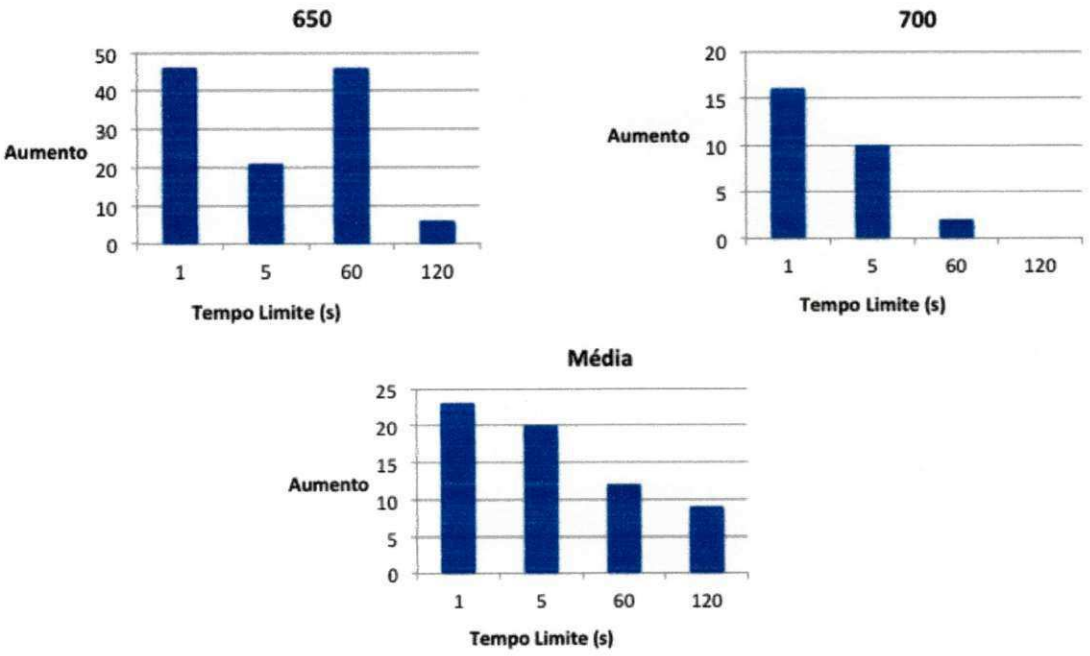


Tabela 5.11: Aumento da cobertura da mudança dos testes gerados pelo SafeRefactorImpact em relação aos gerados pelo SafeRefactor, nas transformações das versões 650 e 700, e na média de todas as transformações.

A cobertura da mudança do SafeRefactorImpact foi maior do que a do SafeRefactor em todos os tempos limites. Neste experimento, foi possível observar que quanto maior o tempo

limite de geração de testes, menor a diferença da cobertura da mudança entre as duas abordagens. Com 1s de tempo limite, SafeRefactor cobriu 41% a menos que SafeRefactorImpact. Com 5s foi 30%, com 60s foi 17% e por fim, com 120s foi 16%. Entretanto, observe que de 1s para 60s a diferença diminuiu em 24%. Já de 60s para 120s diminuiu 1%. Ou seja, a diferença da cobertura da mudança entre as abordagens não diminui proporcionalmente ao aumento do tempo limite de geração de testes. Em algum momento as duas coberturas podem ser iguais, mas baseando-se nesses resultados, se ocorrer, pode ser com um tempo limite de geração de testes alto.

5.1.6 Conclusões

Como conclusão dos resultados obtidos, vamos responder as questões de pesquisa definidas no início do experimento:

Q1: O SafeRefactorImpact tem um resultado equivalente ao SafeRefactor?

Sim. O SafeRefactorImpact não só encontrou as mudanças comportamentais com o mesmo tempo limite de geração de testes que o SafeRefactor, como identificou algumas mudanças com tempos limites menores. A acurácia e precisão do SafeRefactorImpact foram maiores do que do SafeRefactor e a revocação foi igual em todos os tempos limites avaliados.

Q2: A análise de impacto de SafeRefactorImpact diminui a quantidade de métodos identificados para geração de testes em relação ao SafeRefactor?

Sim. A quantidade de métodos identificados pelo SafeRefactorImpact para geração de testes foi menor do que o do SafeRefactor em todas as transformações, devido a análise de impacto da mudança do SafeRefactorImpact.

Q3: O SafeRefactorImpact gera menos casos de testes do que o SafeRefactor?

Sim. No total, o SafeRefactorImpact gerou menos casos de testes para avaliar os refatoramentos do que o SafeRefactor em todos os tempos limites.

Q4: O tempo total da análise é menor no SafeRefactorImpact?

Em parte. Com 1s e 5s de tempos limites o tempo do SafeRefactorImpact foi maior que o do SafeRefactor. Já com 60s e 120s o tempo do SafeRefactorImpact foi menor.

Q5: A cobertura da mudança do SafeRefactorImpact é maior que no SafeRefactor?

Sim. A cobertura da mudança do SafeRefactorImpact foi maior que o a do SafeRefactor em todos os tempos limites.

5.2 Teste de Implementação de Refatoramentos

Nesta seção apresentamos o experimento realizado para avaliar o SafeRefactorImpact em refatoramento implementados pelo Eclipse e JRRT [61]. Inicialmente definimos o experimento, fornecendo uma visão geral. Em seguida, descrevemos seu planejamento e as configurações experimentais. Por fim, apresentamos e discutimos os resultados obtidos.

5.2.1 Definição

Nesta fase, descrevemos os objetivos do experimento, as questões de pesquisa que guiaram o experimento e as métricas utilizadas para avaliar a nossa abordagem.

Objetivo do Estudo

O objetivo deste estudo é analisar o SafeRefactorImpact e o SafeRefactor em uma técnica de testar ferramentas de refatoração com o propósito de avaliar o benefício de utilizar o SafeRefactorImpact ao invés do SafeRefactor. Avaliamos as abordagens com respeito ao tempo total para testar os refatoramentos e equivalência das abordagens em identificar *bugs* nas ferramentas, do ponto de vista de desenvolvedores de ferramentas de refatoração e no contexto de geração de pequenos programas como entradas de testes.

Essa técnica de testar ferramentas de refatoração foi proposta por Soares et al. [66] e está ilustrada na Figura 5.1. A técnica utiliza um gerador automático de programas Java, o JDolly. Ele gera programas exaustivamente, dentro de um escopo e conjunto de restrições específicas passadas pelo usuário. O escopo é o número máximo de pacotes, classes, métodos e atributos que os programas gerados devem ter. As restrições especificam construções específicas dos programas. Por exemplo, para testar o refatoramento que move um método para sua subclasse, os programas devem ter pelo menos uma superclasse declarando um método que pode ser movido para a subclasse.

O primeiro passo da técnica é gerar os programas utilizando o JDolly, a partir da especificação dos programas passada pelo usuário. Em seguida, no segundo passo, para cada programa gerado, a ferramenta de refatoração que está sendo testada, tenta aplicar o refatoramento. Se a ferramenta aplicar o refatoramento no programa, o SafeRefactor avalia se o refatoramento está correto com relação a preservação de comportamento (Passo 3).

Em programas pequenos, de no máximo 4 métodos, a redução absoluta da quantidade de métodos identificados, da quantidade de casos de testes gerados e do tempo de análise entre as duas abordagens é pequena. Entretanto, como a técnica avalia milhares de transformações, queremos investigar a redução total da quantidade de casos de testes gerados e do tempo total de testar uma implementação de refatoramento. Além disso, também avaliaremos com relação a detecção de refatoramentos defeituosos de cada abordagem.

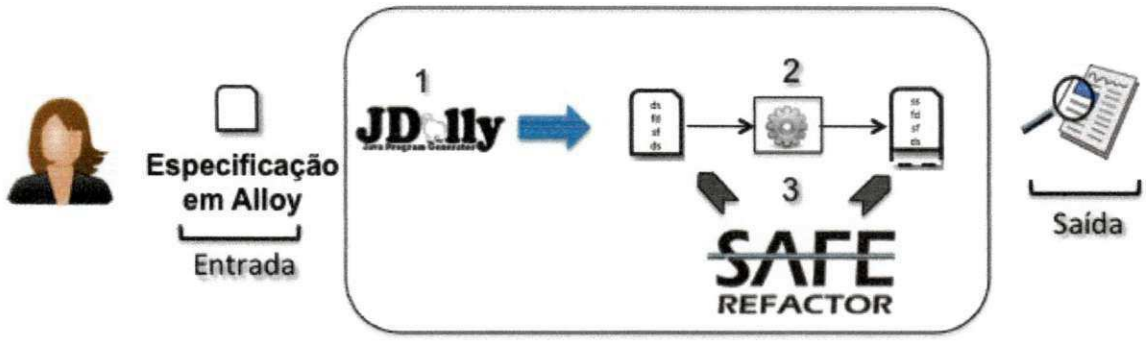


Figura 5.1: Técnica de testar implementação de refatoramentos.

Questões de Pesquisa

As questões de pesquisa que guiam este experimento são:

Q 1 *O SafeRefactorImpact tem um resultado equivalente ao SafeRefactor?*

Q 2 *O SafeRefactorImpact gera menos casos de testes do que o SafeRefactor?*

Q 3 *O tempo total da análise é menor no SafeRefactorImpact?*

Métricas

Para responder as questões de pesquisa Q 2 e Q 3 nós utilizamos as métricas *quantidade de testes* e *tempo total* explicadas na Seção 5.1.1. Para responder a questão de pesquisa Q 1, nós utilizamos a seguinte métrica:

- **Resultado da análise: se é ou não um refatoramento**

Não temos um *baseline* para indicar as transformações que foram avaliadas corretamente e, analisar manualmente milhares de programas é inviável. Então, nós analisamos manualmente apenas as transformações que os resultados foram diferentes entre SafeRefactorImpact e SafeRefactor para analisar qual resultado está correto. Nas transformações que as abordagens fornecem o mesmo resultado, nós assumimos que as análises das duas abordagens estão corretas.

5.2.2 Planejamento

Nesta seção, descrevemos o planejamento deste experimento. Primeiramente, mostramos como foi realizada a seleção das transformações, incluindo os critérios de seleção e características de cada transformação escolhida. Em seguida, apresentamos as variáveis do experimento, que incluem fatores e variáveis de resposta e por fim, exibimos o design do experimento. A execução do experimento seguiu este planejamento.

Seleção das Transformações

Nós avaliamos refatoramentos aplicados nos programas gerados automaticamente pelo JDolly. Esses refatoramentos incluem 10 tipos de implementações do Eclipse e 8 do JRRT. Eclipse é uma IDE bastante conhecida e utilizada por desenvolvedores Java. Ele implementa mais de 20 refatoramentos automatizados. JRRT é uma ferramenta do estado da arte que formaliza algumas pré-condições de dezessete refatoramentos para Java e implementa-os. Os programas em Java são traduzidos para outra linguagem, que é mais fácil avaliar a preservação de comportamento. O objetivo dessas implementações foi aumentar a corretude em relação aos refatoramentos do Eclipse.

É utilizada uma especificação para guiar a geração dos programas para cada tipo de refatoramento. Os programas precisam ter a mínima estrutura necessária para que seja possível a aplicação do refatoramento pela ferramenta de refatoração. A Tabela 5.12 mostra os tipos de refatoramentos testados por cada ferramenta e o escopo utilizado. A quantidade de pacotes (P), classes (C), atributos (A) e métodos (M) dos programas gerados, está descrita para cada tipo de refatoramento na segunda coluna da tabela. Esse é o escopo utilizado para gerar os programas.

Refatoramento	Ecopo P – C – A – M	Eclipse	JRRT
rename class	2 – 3 – 0 – 3	x	x
rename method	2 – 3 – 0 – 3	x	x
rename field	2 – 3 – 2 – 1	x	x
push down method	2 – 3 – 0 – 4	x	
push down field	2 – 3 – 2 – 1	x	x
pull up method	2 – 3 – 0 – 4	x	x
pull up field	2 – 3 – 2 – 1	x	x
encapsulate field	2 – 3 – 1 – 3	x	x
move method	2 – 3 – 1 – 3	x	
Add parameter	2 – 3 – 0 – 3	x	x

Tabela 5.12: Resumo das implementações de refatoramentos avaliadas no Eclipse e JRRT; Escopo = Pacotes (P), Classes (C), Atributos (A) e Métodos (M)

Fatores e Variáveis de Resposta

No experimento realizado para testar ferramentas de refatoração no trabalho de Soares et al. [67], foi utilizado o tempo limite 1s para o SafeRefactor gerar testes. Os autores argumentaram que esse tempo é adequado para o tamanho das transformações. Como SafeRefactorImpact gera testes para um conjunto menor de métodos, nós escolhemos o tempo limite de 0,2s de geração de testes para avaliar se SafeRefactorImpact detecta as mesmas mudanças comportamentais e assim, diminui o tempo total do experimento.

Os fatores deste experimento são:

- **Abordagem:** SafeRefactorImpact e SafeRefactor
- **Tempo limite de geração de testes:** 0,2s e 1s

As variáveis de resposta são:

- Resultado da análise: se é ou não um refatoramento
- Quantidade de testes gerados

- Tempo total da análise

Design do Experimento

Nós escolhemos um design fatorial fracionário, onde vamos executar a abordagem SafeRefactorImpact com 0,2s e SafeRefactor com 0,2s e 1s. Temos um total de 3 execuções para cada refatoramento aplicado. A Tabela 5.13 ilustra todas as execuções para cada transformação.

Abordagem		
SRImpact	SafeRefactor	
Tempo Limite	**Tempo Limite**	
0,2s	0,2s	1s

Tabela 5.13: Design fatorial fracionário utilizado no experimento de teste de implementação de refatoramentos.

5.2.3 Configurações Experimentais

Nós reusamos o experimento utilizado no trabalho de Soares et al. [66], que utiliza o SafeRefactor na técnica de testar implementação de refatoramentos. Para executar a técnica com o SafeRefactorImpact, nós trocamos o SafeRefactor pelo SafeRefactorImpact. O experimento foi executado em dois computadores: um com processador Intel Core i5 (2.3GHz), 4GB de memória RAM e sistema operacional Mac OS 10.6 e outro com 2,7 GHz dual-core, 4GB de memória RAM e sistema operacional Ubuntu 10.04.

Nós utilizamos as versões linha de comando das ferramentas SafeRefactorImpact e SafeRefactor 1.0, o Eclipse versão Helios Service Release 2 e JDK 1.6. para executar os experimentos. Também utilizamos a versão linha de comando do JDolly. Tanto o SafeRefactor quanto o SafeRefactorImpact utilizam o Randop [53], versão 1.3.2. modificado para gerar os testes. Nós modificamos o Randoop para ele aceitar um tempo limite menor que 1s.

5.2.4 Resultados

Para comparar as mudanças comportamentais identificadas por cada abordagem, tempo total de análise das implementações e quantidade de testes gerados na técnica de testar implementação de refatoramentos [66] utilizando o SafeRefactor e o SafeRefactorImpact, nós executamos novamente, o experimento feito no trabalho. Os autores mostraram o tempo total para testar cada tipo de refatoramento, mas achamos justo executar as duas abordagens no mesmo ambiente. Além disso, eles não calcularam a quantidade de testes gerados, pois não era o foco do trabalho.

Neste experimento, o SafeRefactor utilizou um tempo limite de geração de testes de 1s. Ao replicar o experimento, SafeRefactorImpact identificou todas as falhas identificadas pelo SafeRefactor. Além disso, o SafeRefactorImpact identificou algumas falhas não identificadas pelo SafeRefactor. No experimento que utilizamos o tempo limite de 0,2s para o SafeRefactor, ele não identificou 44 falhas identificadas pelo SafeRefactorImpact com o mesmo tempo limite e pelo SafeRefactor com 1s de tempo limite. A Tabela 5.16 mostra a quantidade de falhas não identificadas em algumas implementações de refatoramentos do Eclipse e JRRT.

Replicamos o experimento do trabalho [66], onde o SafeRefactor utiliza 1s de tempo limite de geração de testes. O SafeRefactorImpact reduziu em 72% a quantidade de testes e avaliou os mesmos refatoramentos do Eclipse em 25,8 horas a menos. Para avaliar o JRRT, o SafeRefactorImpact gerou 50% menos casos de testes do que o SafeRefactor e reduziu em 31,29 horas o tempo para avaliou os mesmos refatoramentos No experimento que o SafeRefactor utiliza 0,2s de tempo limite de geração de testes, o SafeRefactorImpact reduziu em 52% a quantidade de testes e foi 7,31h mais rápido que o SafeRefactor para avaliar os refatoramentos do Eclipse. Para avaliar os refatoramentos do JRRT, o SafeRefactorImpact reduziu em 3% a quantidade de testes e foi 1,88h mais rápido que o SafeRefactor. Entretanto, com esse tempo limite o SafeRefactor não identifica algumas mudanças comportamentais.

As Tabelas 5.14 e 5.15 mostram os resultados do experimentos do Eclipse e JRRT, respectivamente.

Refatoramentos do Eclipse														
Refatoramento	Prog.	Fal.	Tempo (h)			Testes			Redução de:					
			SRI	SR		SRI	SR		Tempo (h)		Tempo (%)		Testes (%)	
			0,2s	0,2s	1s	0,2s	0,2s	1s	0,2s	1s	0,2s	1s	0,2s	1s
rename class	15.322	145	0,41	0,98	1,85	0,25	12,02	48,45	0,57	1,43	57,86	77,44	97,85	99,46
rename method	11.263	0	0,69	1,28	2,10	1,15	6,65	21,82	0,58	1,40	45,55	66,78	82,68	94,72
rename field	19.424	0	4,62	6,36	11,71	1,89	2,94	2,97	1,74	7,08	27,32	60,49	35,50	36,23
push down method	20.544	853	1,31	1,81	3,58	20,56	29,66	153,00	0,49	2,26	27,46	63,18	30,69	86,56
push down field	11.936	92	0,66	0,98	1,68	2,38	2,96	2,99	0,32	1,02	32,61	60,53	19,52	20,45
pull up method	8.937	202	0,98	1,40	2,70	21,00	32,04	157,68	0,42	1,72	30,27	63,70	34,46	86,68
pull up field	10.927	546	1,15	1,70	2,91	2,45	2,97	2,99	0,55	1,76	32,52	60,49	17,56	18,21
encapsulate field	2.000	0	0,33	0,46	1,04	0,99	30,11	133,21	0,13	0,71	28,33	68,41	96,68	99,25
move method	22.905	3.586	1,09	1,59	2,97	14,43	24,56	110,86	0,49	1,87	31,26	63,11	41,22	86,97
add parameter	30.186	2.238	3,71	5,73	10,30	3,52	11,86	45,84	2,02	6,58	35,20	63,8	70,27	92,3
Total	153.444	7.662	14,95	22,29	40,84				7,31	25,83				
Média						6,86	15,56	67,98			34,83	64,79	52,64	72,08

Tabela 5.14: Resultados do experimento nas implementações de refatoramentos do Eclipse.

5.2.5 Discussão

Nesta seção, vamos discutir os principais resultados deste experimento. A seguir, falaremos sobre equivalência dos resultados, quantidade de testes e tempo total da análise.

Equivalência dos Resultados da Análise

Em algumas transformações do refatoramento *encapsulate field* do JRRT, o SafeRefactor com 1s de tempo limite de geração de testes, não identificou algumas falhas que o SafeRefactorImpact identificou. Nós analisamos manualmente essas transformações. O *bug* é o mesmo reportando por Soares et al. [66]. Nessas transformações que analisamos, o SafeRefactor identifica o mesmo conjunto de métodos que o SafeRefactorImpact. Entretanto, a ordem que os métodos são passados para o Randoop não é a mesma. Nós executamos o SafeRefactor com um tempo limite de geração de testes de 5s e ele identificou a mudança.

Nos refatoramentos do Eclipse, o SafeRefactor não identificou 33 falhas em 5 tipos de

Refatoramentos do JRRT														
Refatoramento	Prog.	Fal.	Tempo (h)			Testes			Redução de:					
			SRI	SR		SRI	SR		Tempo (h)		Tempo (%)		Testes (%)	
			0,2s	0,2s	1s	0,2s	0,2s	1s	0,2s	1s	0,2s	1s	0,2s	1s
rename class	15.322	0	4,21	4,46	9,66	0,99	14,37	61,61	0,25	5,44	5,61	56,33	93,05	98,38
rename method	11.263	482	2,99	3,37	6,99	10,53	10,75	40,43	0,37	3,99	11,23	57,19	1,97	73,93
rename field	14.000	0	5,07	5,29	11,16	2,99	2,31	2,99	0,21	6,08	4,02	54,49	-1,16	0,20
push down field	11.936	0	1,16	1,15	2,72	2,98	2,96	3,0	-0,01	1,55	-1,60	57,14	-0,34	0,58
pull up method	8.937	10	1,38	1,49	3,43	33,59	27,71	154,07	0,10	2,04	7,07	59,61	-21,13	78,19
pull up field	10.927	0	1,97	1,92	4,62	2,99	2,97	3,0	-0,05	2,65	-2,53	57,32	-0,64	0,16
encapsulate field	2.000	437	0,52	0,57	0,80	36,03	24,00	169,85	0,05	0,27	9,34	34,37	-49,78	78,78
add parameter	30.186	0	6,48	7,45	15,76	9,54	9,75	34,37	0,96	9,27	12,97	58,84	2,13	72,22
Total	**104.571**	**929**	**23,78**	**25,70**	**55,14**				**1,88**	**31,29**				
Média						**12,45**	**12,55**	**58,66**			**5,76**	**54,41**	**3,01**	**50,30**

Tabela 5.15: Resultados do experimento nas implementações de refatoramentos do JRRT.

refatoramentos utilizando 0,2s de tempo limite de geração de testes. Em 3 tipos de refatoramentos, não foram identificados *bugs* no experimento do trabalho [66]. Nos 8 tipos de refatoramentos do JRRT que nós avaliamos, foram identificados *bugs* em 3 tipos. Em dois tipos (*rename method* e *encapsulate field*) o SafeRefactor não identificou 11 falhas utilizando o tempo limite de 0,2s de geração de testes.

Quantidade de Testes

Em relação ao SafeRefactor utilizando 1s de tempo limite de geração de testes, o SafeRefactorImpact diminuiu em 72% a quantidade de testes gerados na avaliação do eclipse e 50% no JRRT. Essa redução já era esperada, pois ele teve mais tempo que o SafeRefactorImpact para gerar os testes. Com 0,2s, a redução no experimento do eclipse foi de 52%. Mesmo com o mesmo tempo limite de geração de testes, o SafeRefactorImpact ainda gerou menos casos de testes. Nossa explicação é a mesma do experimento do JHotDraw. Como o SafeRefactorImpact identifica um conjunto menor de métodos, ele gera menos casos de testes. No JRRT, o SafeRefactorImpact gerou mais casos de testes em 5 refatoramentos: *rename*

Falhas não detectadas pelo SR 0,2s		
Refatoramento	Eclipse	JRRT
rename class	3	
rename method		4
push down method	2	
pull up field	2	
encapsulate field		7
move method	14	
add parameter	12	
Total	**33**	**11**

Tabela 5.16: Falhas não detectadas pelo SafeRefactor utilizando um tempo limite de 0,2s.

field, *push down field*, *pull up method*, *pull up field* e *encapsulate field*. No total a redução do SafeRefactorImpact em relação ao SafeRefactor foi de 3%, porque no refatoramento rename class, a redução foi de 93%.

Observe na Tabela 5.12 o escopo das transformações *rename field*, *push down field* e *pull up field*. Os programas gerados para testar esses refatoramentos contém no máximo 1 método. Então, a redução da quantidade de métodos do SafeRefactorImpact e SafeRefactor é pequena (nó máximo um método). Por isso, a diferença na quantidade de testes não foi grande. No experimento do eclipse a redução foi maior, mas os refatoramentos *push down field* e *pull up field* tiveram as menores reduções (19% e 17%, respectivamente), enquanto a redução total foi de 50%. O refatoramento *rename field* também reduziu abaixo do total: 35%. A maior redução foi do *rename class* com 97%.

Tempo

Utilizando o SafeRefactorImpact, o tempo total para testar os refatoramentos do Eclipse e JRRT foi reduzido em 25,8h e 31h, respectivamente, em relação ao SafeRefactor utilizando o tempo limite de geração de testes de 1s (tempo considerado adequado pelos autores da técnica [66]). No total, temos uma redução de 56h, o equivalente a 60% do tempo total. Na execução do SafeRefactor utilizando 0,2s de tempo limite de geração de testes, a redução foi menor (7,3h no Eclipse e 1,8h no JRRT), entretanto com esse tempo de geração de testes, o

SafeRefactor não identificou algumas falhas. A redução de tempo no experimento do JRRT foi menor pelo mesmo motivo explicado na redução da quantidade de testes.

5.2.6 Conclusões

Como conclusão dos resultados obtidos, vamos responder as questões de pesquisa definidas no início do experimento:

Q1: O SafeRefactorImpact tem um resultado equivalente ao SafeRefactor?

Utilizando o mesmo tempo limite de geração de testes (0,2s) para as duas abordagens, o SafeRefactor não identifica 44 falhas identificadas pelo SafeRefactorImpact. Utilizando o tempo limite de 1s o resultado do SafeRefactor é equivalente ao resultado do SafeRefactorImpact com 0,2s, com relação a identificação de mudanças comportamentais.

Q2: O SafeRefactorImpact gera menos casos de testes do que o SafeRefactor?

Sim. No total o SafeRefactorImpact gerou menos casos de testes para avaliar os refatoramentos do que o SafeRefactor. Em alguns refatoramentos do JRRT, o SafeRefactorImpact gerou mais casos de testes do que o SafeRefactor utilizando o tempo limite de geração de testes de 0,2s.

Q3: O tempo total da análise é menor no SafeRefactorImpact?

Sim. O SafeRefactorImpact avaliou os refatoramentos do Eclipse e JRRT mais rápido que o SafeRefactor. Para avaliar o push down field e o pull up field do JRRT, o SafeRefactor foi no total 0,01h e 0,05h mais rápido que o SafeRefactorImpact, respectivamente.

5.3 Ameaças a Validade

A seguir nós descrevemos as ameaças a validade interna (Seção 5.3.1), externa (Seção 5.3.2) e de construção (Seção 5.3.3) dos nossos experimentos.

5.3.1 Validade Interna

Algumas ameaças a validade interna estão relacionadas com o analisador de impacto Safira, utilizado pelo SafeRefactorImpact. Safira utiliza a API do ASM para realizar a análise de impacto do código. Essa API analisa o bytecode Java. Diferenças no código fonte que não

reflitam no bytecode não são identificadas. Atualmente, não analisamos classes anônimas na análise de impacto. Se alguma mudança que não possa ser testada (exemplo: visibilidade baixa) for exercitada diretamente apenas por algum método de uma classe anônima, não será possível identificá-la. Além disso, Safira não analisa classes de bibliotecas. Se algum método tiver um parâmetro que seja do tipo de uma classe de uma biblioteca, o Randoop pode não gerar casos de testes para ele se não tiver algum método no programa que gera um objeto do tipo da dependência e esse método estiver na lista de métodos para testar.

Outras ameaças estão relacionadas com o Randoop. A escolha do tempo limite utilizado para geração de testes tem influência na detecção de mudanças comportamentais. Ele tem que ser criteriosamente escolhido e depende do tamanho do programa e da transformação. Então, é possível que aumentando o tempo limite sejam detectadas mais mudanças comportamentais. Com a aleatoriedade do Randoop, o SafeRefactorImpact e o SafeRefactor podem dar resultados diferentes para uma mesma entrada. Além disso, a ordem que os métodos são passados para o Randoop influencia a geração de testes. Então, uma mudança ainda não identificada, pode ser detectada apenas mudando a ordem dos métodos sem alterar o tempo limite. Por fim, diferenças nas mensagens da saída padrão (*System.out.println*) não são detectadas.

5.3.2 Validade Externa

Algumas ameaças a validade externa estão relacionadas ao Randoop. Ele não lida bem com concorrência. Nessas situações, a abordagem pode produzir resultados não determinísticos. Então, é recomendado que nossa abordagem seja utilizada apenas em programas sequenciais para ter uma confiança maior nos resultados. Além disso, o Randoop não pode gerar casos de testes que exercitem algumas mudanças relacionadas a interface gráfica (GUI). Essas alterações podem não ser triviais para serem testadas através de testes JUnit. Por fim, foram avaliadas 10 versões do JHotDraw no primeiro experimento. Estes programas podem não ser representativos de todos os programas a fim de generalizar os resultados.

5.3.3 Validade de Construção

Uma ameaça a validade de construção é relacionada a hipótese de mundo aberto e mundo fechado. No mundo fechado temos que usar a coleção de testes fornecida pelo programa que está sendo refatorado. Nossa abordagem segue um pressuposto de mundo aberto, em que cada método público pode ser um alvo em potencial para a coleção de testes gerada pelo Randoop. Ele pode gerar um caso de teste que expõe uma mudança comportamental. No entanto, o caso de teste pode mostrar um cenário inválido de acordo com o domínio do software.

Capítulo 6

Conclusões

Neste trabalho propomos uma abordagem para avaliar preservação de comportamento em refatoramentos baseada em geração automática de testes e análise de impacto da mudança. A abordagem é uma extensão do SafeRefactor [68], uma ferramenta que também avalia preservação de comportamento em refatoramentos mas não utiliza análise de impacto. A abordagem analisa duas versões de um programa para identificar os métodos impactados pela mudança e utiliza o Randoop [53] para gerar automaticamente uma coleção de testes, dentro de um tempo limite informado pelo usuário, apenas para métodos impactados. Implementamos a abordagem proposta em uma ferramenta chamada SafeRefactorImpact. Para identificar os métodos impactados, propomos uma abordagem para avaliar o impacto da mudança. Implementamos a análise de impacto em uma ferramenta chamada Safira que é utilizada pelo SafeRefactorImpact. Avaliamos o SafeRefactorImpact em um conjunto de 10 transformações aplicadas a um sistema real (JHotDraw), e em uma técnica proposta por Soares et al. [66] para testar implementações de refatoramentos. Além disso, comparamos SafeRefactorImpact com SafeRefactor. O objetivo da avaliação foi analisar o benefício de incorporar a análise de impacto da mudança na abordagem original do SafeRefactor. O SafeRefactorImpact conseguiu identificar mudanças comportamentais não identificadas pelo SafeRefactor utilizando o mesmo tempo limite de geração de testes e reduziu em torno de 60% o tempo para testar implementações de refatoramentos utilizando o tempo limite de geração de testes que as duas abordagens identificam as mesmas mudanças comportamentais.

Na avaliação do JHotDraw, vimos que na análise de programas grandes, a análise de impacto do SafeRefactorImpact reduziu em média, 93% a quantidade de métodos a serem

testados e assim, focando a geração de testes apenas em métodos impactados, foi possível encontrar mais mudanças comportamentais que o SafeRefactor. Em 3 versões do JHotDraw, o SafeRefactorImpact identificou mudanças comportamentais que o SafeRefactor não identificou no experimento. Em outras 2 versões, o SafeRefactor precisou de um tempo limite de geração de testes maior que o SafeRefactorImpact para identificar a mudança.

No experimento de testes de ferramentas, foi mostrado que utilizando o SafeRefactorImpact tornou a técnica de testar implementações de refatoramentos mais rápida. A análise de impacto do SafeRefactorImpact permitiu que fosse utilizado um tempo limite de geração de testes menor, devido a quantidade de métodos para geração de testes também ser menor. Utilizando o mesmo tempo limite do SafeRefactorImpact de 0,2s, o SafeRefactor não identificou algumas mudanças comportamentais. Utilizando o tempo limite de geração de testes de 1s o resultado do SafeRefactor é equivalente ao resultado do SafeRefactorImpact com 0,2s, com relação a identificação de mudanças comportamentais e o SafeRefactorImpact reduziu em torno de 60% o tempo para testar as implementações de refatoramentos avaliadas do Eclipse e JRRT. Essa redução resultou em 56h a menos para testar todas as implementações.

O tempo limite escolhido para geração de testes pode influenciar a detecção de mudanças comportamentais. O SafeRefactorImpact mostrou-se menos sensível ao tempo limite passado para o gerador automático de testes. Pois, com a geração de testes apenas para os métodos impactados pela mudança, é necessário um tempo limite menor para identificar uma mudança comportamental. Além disso, a análise de impacto permitiu diminuir algumas limitações do gerador automático de testes enfrentadas pelo SafeRefactor. Por exemplo, o Randoop não é eficaz em gerar casos de testes para componentes de interface gráfica ou métodos que manipulam arquivos. Como consequência, ele utiliza o tempo de geração de testes para tentar testar essas limitações e pode deixar de gerar outros casos de testes que identificariam uma mudança comportamental. Então, o SafeRefactorImpact pode melhorar esse problema quando esses métodos não forem impactados pela mudança, pois não serão gerados casos de testes para eles. Da mesma forma, se a análise de impacto diminuir muito o escopo de métodos identificados para geração de testes, pode permitir avaliar transformações aplicadas a sistemas concorrentes, que hoje é uma limitação do SafeRefactor.

O analisador de impacto da mudança, Safira, utilizado no SafeRefactorImpact, realiza análise estática para identificar métodos impactados. Uma característica da análise estática

é ser conservadora, ou seja, tenta identificar todos os possíveis impactos. Já uma análise dinâmica tem a vantagem de ser mais precisa. Entretanto, ela pode deixar de identificar alguns impactos que não foram exercitados pela coleção de testes. Por isso, nós decidimos utilizar uma análise totalmente estática pois, preferimos identificar uma quantidade maior de métodos impactados para diminuir os falso-negativos (métodos impactados que não são identificados). Para aumentar a confiança na análise de impacto, nós especificamos em Alloy, uma linguagem de especificação formal, as leis que formalizam o conjunto de métodos diretamente impactados por cada subtransformação. Nós fizemos algumas asserções para avaliar a especificação, utilizando a ferramenta Alloy Analyzer, que permite checar se a especificação está consistente. Além de aumentar a confiança no analisador de impacto, essas análises permitiram entender melhor o domínio e ajudaram na implementação da ferramenta.

Uma alternativa para a nossa abordagem é utilizar métodos formais para definir formalmente e provar alguns tipos de implementações de refatoramentos para um subconjunto de Java. Desta forma, garante que as transformações aplicadas serão corretas para o subconjunto e escopo definido. Uma desvantagem, é que restringe os programas e as transformações que podem ser aplicadas. Além disso, provas formais requer um esforço grande. Por isso, propomos uma abordagem mais prática, que utiliza uma coleção de testes gerados automaticamente como o oráculo para avaliar se uma transformação preservou o comportamento do programa. Não temos garantia que todas as mudanças comportamentais serão identificadas mas podemos aumentar a confiança em atividades de refatoramentos.

6.1 Trabalhos Relacionados

Relacionamos nosso analisador de impacto da mudança, Safira, com outros trabalhos que propõem analisadores de impacto para sistemas orientados a objetos. Nossa abordagem para avaliar refatoramentos baseada no impacto da mudança pode ser relacionada com outras abordagens que também avaliam preservação de comportamento em refatoramentos ou que propõem soluções para garantir corretude nessa atividade. A seguir, falaremos um pouco sobre alguns trabalhos relacionados na área de análise de impacto, refatoramentos e testes.

6.1.1 Análise de Impacto

Law e Rothermel [39] propuseram uma abordagem baseada em particionamentos estáticos e dinâmicos e algoritmos recursivos de grafos de chamadas para identificar os métodos impactados por uma mudança. Em resumo o algoritmo para identificar os métodos impactados funciona da seguinte forma: suponha uma mudança proposta para um método p. Será analisado apenas o impacto que pode propagar-se pelos caminhos que passaram por p, identificados dinamicamente. Então, qualquer método chamado após a chamada de p e qualquer método que está na pilha de chamada depois do retorno de p é incluído no conjunto dos possíveis métodos impactados. A técnica é dinâmica e não requer análise estática do programa. Então, a análise pode deixar de incluir métodos que mudam de comportamento se a mudança for aplicada. E essa quantidade pode ser maior do que em uma análise estática, pois depende do quanto os testes cobrem a mudança. Ela pode ser usada nos casos em que não é exigido maximizar o conjunto impactado e se deseja informações mais precisas de impacto em um perfil operacional específico ou conjunto de execuções. A análise estima o impacto da mudança antes dela ter sido realizada. Lile Hattori [28] propôs o analisador de impacto Impala, que implementa uma técnica de análise de impacto probabilística. A técnica identifica os impactos de uma mudança antes de sua implementação e atribui probabilidades de ocorrência aos impactos através do uso de informação sobre o histórico de mudanças do software analisado. O Impala recebe um conjunto de parâmetros que indicam características da transformação que deseja ser aplicada, e estima o impacto utilizando a técnica proposta. A nossa abordagem de análise de impacto, implementada por Safira, é estática e identifica métodos impactados em qualquer mudança no código após ela ter sido realizada. Além disso, Safira não necessita de informações adicionais para avaliar a mudança.

Kung et al. [36] propuseram um método para identificar classes impactadas devido a mudanças estruturais em classes de biblioteca de linguagens orientadas a objetos. O método é baseado em uma abordagem de engenharia reversa que extrai da biblioteca informações de classes e seus relacionamentos. Essas informações são representadas em grafos de dependências utilizados para automaticamente identificar as mudanças e seus efeitos. Li e Offut [44] realizaram um estudo para examinar como mudanças feitas em programas orientados a objetos podem afetar classes do programa. Eles propuseram um algoritmo que calcula o fechamento transitivo do grafo de dependência do programa. O cálculo do efeito

da mudança no programa inclui a identificação de efeitos dentro de uma classe, a identificação dos efeitos entre classes relacionadas e a identificação dos efeitos causados por relações de herança entre classes. Essas abordagens identificam o impacto da mudança a nível de classes. Elas analisam mudanças feitas no programa para identificar as classes impactadas. Nossa abordagem analisa a nível de métodos e identifica os métodos impactados por uma mudança.

Chianti [56] é uma ferramenta de análise de impacto de mudanças para Java. Com base em uma coleção de testes do usuário e da mudança efetuada, ela decompõe a mudança em mudanças atômicas e gera um grafo de dependência. A ferramenta indica os testes que foram impactados pela mudança e que por isso, devem ser executados novamente. Zhang et al. [80] propuseram o FaultTracer, um analisador de impacto da mudança que reimplementa o Chianti e adiciona melhorias, refinando as dependências entre as mudanças atômicas e acrescentando mais regras para calcular o impacto da mudança. Da mesma forma que Safira, as abordagens recebem duas versões de um programa e decompõem a mudança em transformações menores. Entretanto, as subtransformações da análise de Safira diferem das mudanças atômicas consideradas nas abordagens pela granularidade. Uma mudança atômica tem a menor granularidade possível para tornar a abordagem precisa. Essa granularidade pequena é útil porque as abordagens também trabalham com depuração. Então quanto menor a granularidade de uma mudança atômica, maior a precisão da depuração do código. As abordagens não calculam o impacto de cada mudança atômica separadamente. Nós fazemos isso com as subtransformações. Se considerarmos subtransformações de granularidades menores, pode resultar em um custo adicional desnecessário para calcular o conjunto impactado. Outra diferença é que Chianti e FautTracer dependem de uma coleção de testes para avaliar o impacto da mudança. Grafos de chamadas são construídos a partir da execução dos testes no programa e são analisados para identificar os testes impactados. Safira não depende de coleção de testes. Ela recebe duas versões de um programa e identifica métodos que foram impactados pela mudança.

6.1.2 Refatoramento

Opdyke [51] propôs alguns refatoramentos para C++ e especificou condições para garantir preservação de comportamento. Roberts [57] automatizou os refatoramentos propostos por

Opdyke. Entretanto, não houve nenhuma prova formal da corretude dessas transformações. Mais tarde, Tokuda e Batory [74] mostraram que as condições propostas por Opdyke não garantiam preservação de comportamento.

Schäfer et al. [61] propuseram uma ferramenta (JRRT) que formaliza algumas pré-condições de dezessete refatoramentos para Java e implementa-os. Os programas em Java são traduzidos para outra linguagem, que é mais fácil avaliar a preservação de comportamento. O objetivo foi aumentar a corretude das implementações. Entretanto, no experimento feito neste trabalho (Seção 5.2), nós encontramos mudanças comportamentais nos refatoramentos *rename method*, *pull up method* e *encapsulate field* do JRRT. Soares et al [66] encontrou mais de 20 *bugs* diferentes nos refatoramentos do JRRT. Nossa abordagem pode ser usada em conjunto com estas ferramentas para aumentar a confiança que a transformação preserva comportamento. Além disso, nossa abordagem pode ser aplicada a qualquer transformação.

Steimann e Thies [70] mostraram que mudar a visibilidade de entidades em Java pode acarretar em erros de compilação e alterações comportamentais durante refatoramentos. Eles propõem um conjunto de restrições relacionadas a visibilidade que alguns refatoramentos devem satisfazer para preservar o comportamento. Eles identificaram vários *bugs* no Eclipse que também pode ser detectados pela nossa abordagem. Como eles não formalizam todas as condições necessárias para preservar o comportamento, nossa ferramenta pode ser útil para aumentar a confiança, avaliando se o refatoramento aplicado preserva o comportamento do programa.

Borba et al. [7] propuseram um conjunto de refatoramentos para um subconjunto de Java com a semântica de cópia (ROOL). Eles provaram os refatoramentos com relação a uma semântica formal. Silva et al. [65] definiram e provaram formalmente um conjunto de leis para que algumas transformações preservassem o comportamento, em uma linguagem OO sequencial, com respeito a uma semântica formal. No entanto, essas abordagens não consideram todas as construções de Java, como por exemplo, sobrecarga. Além disso, existem muitos tipos de refatoramentos e formalizar refatoramentos é custoso. Por fim, sempre que a linguagem evoluir, os refatoramentos precisam ser provados novamente. Nossa abordagem avalia qualquer transformação. Ela pode ser utilizada em conjunto com essas abordagens para identificar mudanças comportamentais que não estão no escopo desses trabalhos.

Soares et al. [68] propuseram o SafeRefactor, uma ferramenta que analisa uma transformação e gera testes para detectar mudanças comportamentais. A análise do SafeRefactor identifica todos os métodos em comum (métodos com a mesma assinatura) nas duas versões do programa e gera automaticamente uma coleção de testes dentro de um tempo limite informado pelo usuário. O SafeRefactor gera testes para todo o programa. Esse é um problema que ocorre na prática para testar um refatoramento (Ver Seção 2.3.2). O grande diferencial da nossa ferramenta está na análise de impacto inexistente no SafeRefactor. A nossa geração de testes é guiada pelo impacto da mudança. Devido a análise de impacto, melhoramos a corretude na análise de transformações em grandes sistemas, como o JHotDraw. Encontramos mudanças comportamentais não identificadas pelo SafeRefactor com o mesmo tempo limite de geração de testes.

Bozó et al. [8] apresentaram uma técnica para avaliar refatoramentos baseada no impacto da mudança. A técnica utiliza análise de impacto para selecionar os testes de regressão que serão executados após um refatoramento. Ela foi proposta para Erlang, uma linguagem funcional dinamicamente tipada. A análise de impacto implementada pela técnica utiliza estratégias de grafos de dependências e fatiamento de programas. O trabalho difere do nosso em alguns sentidos. Primeiramente, a técnica proposta depende de uma coleção de testes. Ela prioriza casos de testes de regressão que foram afetados por um refatoramento. A nossa abordagem não depende de uma coleção de testes. Ela gera testes para avaliar a mudança. Além disso, a técnica foi proposta para uma linguagem funcional, enquanto a nossa abordagem é para programas Java, podendo ser adaptada para outras linguagens orientadas a objetos.

Rachatasumrit e Kim [55] realizaram um estudo do impacto da aplicação de refatoramentos nos testes de regressão. Neste estudo, foram levantadas três questões de pesquisa: Q1: Os refatoramentos estão sendo bem testados? Q2: Qual a porcentagem de testes que exercitam refatoramentos? Q3: Dos testes que falharam, quais exercitam refatoramentos? Eles utilizam o FaultTracer [80], um analisador de impacto da mudança que estende o Chianti para identificar os métodos impactados por um refatoramento e os refatoramentos que impactam um determinado teste. Os resultados da avaliação em três grandes projetos *open source*, mostraram que os refatoramentos não estão sendo bem testados. Apenas 22% dos métodos e atributos refatorados são cobertos pelos testes de regressão. Esse estudo é mais

uma motivação para o nosso trabalho. Ele mostrou que na prática, os testes não focam na mudança. A nossa abordagem gera testes para o que foi modificado e impactado, ajudando a melhorar a cobertura dos testes nos refatoramentos aplicados.

6.1.3 Testes

JUnitMX [78] é uma ferramenta que indica se uma coleção de testes está exercitando todas as entidades impactadas por uma mudança. Eles propuseram a métrica cobertura da mudança que avalia o quanto uma coleção de testes exercitou determinada mudança. O JUnitMX utiliza o Chianti para realizar a análise de impacto. A cobertura da mudança é medida a nível de mudanças atômicas identificadas na análise de Chianti. A cobertura da mudança proposta neste trabalho, calcula a cobertura a nível de métodos e *statements* impactados. Ela está implementada no SafeRefactorImpact que calcula a cobertura da mudança, dada uma coleção de testes e duas versões de um programa.

Soares et al. [66; 69] apresentaram uma técnica de testar ferramentas de refatoramentos que utiliza um gerador automático de programas e o SafeRefactor [68]. Eles identificaram vários *bugs* no Eclipse, JRRT e NetBeans. Nós avaliamos a técnica utilizando o SafeRefactorImpact e vimos que tornou a técnica mais rápida. A análise de impacto do SafeRefactorImpact permitiu que fosse utilizado um tempo limite de geração de testes menor, devido a quantidade de métodos para geração de testes também ser menor.

Andrade e Machado [2] propuseram uma abordagem de testes de conformidade de software para sistemas de tempo real baseada em modelos que lida com requisitos de dados e tempo. Além disso, foi proposto o SymbolRT, um gerador de casos de testes para esses sistemas, que utiliza um modelo simbólico, chamado de TIOSTS, incluindo uma definição formal de conceitos e propriedades a serem seguidos durante o processo de geração dos casos de testes. Por ser baseado em definições formais, o gerador de casos de testes é mais confiável que o Randoop, a ferramenta que utilizamos na nossa abordagem, que utiliza uma técnica de geração aleatória de casos de testes. Entretanto, para utilizar o SymbolRT, é necessário especificar modelos para o sistema a ser testado. Na nossa abordagem, passamos apenas o conjunto de métodos e um tempo limite de geração de testes, o que torna a abordagem mais prática.

6.2 Trabalhos Futuros

Pretendemos em um trabalho futuro, diminuir algumas limitações de Safira, como por exemplo, analisar classes de bibliotecas. É possível que analisando também as bibliotecas dos programas, sejam identificados mais métodos impactados. Também pretendemos realizar uma análise de fluxo de dados, que permite identificar um conjunto maior de entidades impactadas. Além disso, outro trabalho futuro seria provar, de acordo com uma semântica formal, as leis da análise de impacto formalizadas neste trabalho, que identificam o conjunto de métodos impactados por uma subtransformação. Desta forma, podemos ter mais confiança no analisador de impacto.

Neste trabalho, avaliamos o SafeRefactorImpact em transformações aplicadas a um sistema real. Futuramente, vamos avaliar mais sistemas reais, especialmente sistemas grandes de milhões de linhas de código e sistemas concorrentes, para analisar melhor o efeito da análise de impacto em reduzir algumas limitações do SafeRefactor. Também pretendemos avaliar novamente as ferramentas de refatoração utilizando outro escopo e estrutura dos programas gerados. É possível que, aumentando o escopo dos programas gerados e extendendo o JDolly para gerar programas com mais construções de Java, sejam detectados mais *bugs* nas ferramentas de refatoração. Entretanto, ao aumentar o escopo e as construções da linguagem, o espaço de estados aumenta, gerando muito mais programas e aumentando também o tempo da avaliação. Como o SafeRefactorImpact mostrou ser mais rápido em testar as ferramentas, podemos utilizá-lo para realizar essa avaliação com um custo menor, tornando a análise mais viável.

Bibliografia

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, DC, USA, 1993.

[2] W. L. Andrade and P. Machado. Generating test cases for real-time systems based on symbolic models. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2013.

[3] L. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley Sons, New York, NY, USA, 1988.

[4] L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 167–175, Washington, DC, USA, 2005.

[5] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] S. A. Bohner and R. S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, New York, NY, USA, 1996.

[7] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, pages 53–100, 2004.

[8] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, 2011.

[9] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, 2003.

[10] Y. Chen, D. S. Rosenblum, and K. Vo. Testtube: a system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 211–220, Los Alamitos, CA, USA, 1994.

[11] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip. Crisp–a fault localization tool for java programs. In *Proceedings of the 29th international conference on Software Engineering*, Washington, DC, USA, 2007.

[12] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, 1990.

[13] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2007.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[15] D. Dig and R. Johnson. The role of refactorings in api evolution. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005.

[16] Eclipse.org. Eclipse project. At http://www.eclipse.org, 2011.

[17] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, pages 102–112, 2000.

[18] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, pages 159–182, 2002.

[19] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, pages 63–86, 1996.

[20] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring java applications to use generic libraries. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 71–96, Berlin, Heidelberg, 2005.

[22] R. Gheyi, T. Massoni, and P. Borba. A rigorous approach for proving model refactorings. In *Proceedings of the IEEE/ACM international Conference on Automated Software Engineering*, pages 372–375, New York, NY, USA, 2005.

[23] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *SIGPLAN Notices*, pages 493–510, 1975.

[24] V.L. Hamilton and M.L. Beeby. Issues of traceability in integrating tools. In *Tools and Techniques for Maintaining Traceability During Design*, pages 4/1 –4/3, 1991.

[25] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[26] F. M. Haney. Module connection analysis - a tool for scheduling software debugging activities. In *Proceedings of the Fall Joint Computer Conference*, page 173, 1972.

[27] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. *SIGSOFT Software Engineering Notes*, pages 154–163, 1994.

[28] L. P. Hattori. Análise probabilística de impacto de mudanças baseada em históricos de mudanças do software. Master's thesis, Federal University of Campina Grande, 2008.

[29] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, pages 35–46, 1988.

[30] L. Huang and Y. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Proceedings of the ACIS International Conference on Software Engineering Research, Management & Applications*, pages 374–384, Washington, DC, USA, 2007.

[31] M. Hutchins and K. Gallagher. Improving visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 294–, Washington, DC, USA, 1998.

[32] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[33] P. Jönsson and Blekinge T. H. *Impact Analysis: Organisational Views and Support Techniques*. Blekinge Institute of Technology Licentiate Series. Blekinge Institute of Technology, 2005.

[34] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. John Wiley Sons, New York, NY, USA, 2001.

[35] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal System Software*, pages 187–195, 1990.

[36] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 202 –211, Arlington, TX , USA, 1994.

[37] Baresi L., P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. *International Conference on Software Testing, Verification and Validation*, pages 185–194, 2010.

[38] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 430–, Washington, DC, USA, 2003.

[39] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003.

[40] M. Lee, A.J. Offutt, and R.T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, pages 61 –70, 2000.

[41] D. Leffingwell and D. Widrig. *Managing Software Requirements: A Use Case Approach*. Pearson Education, 2 edition, 2003.

[42] S. Lehnert. A review of software change impact analysis. Technical report, Department of Software Systems / Process Informatics, Ilmenau, Germany, 2011.

[43] A. Leitner, L. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the the Joint Meeting of the European Software Engineering Conference*, pages 425–434, New York, NY, USA, 2007.

[44] Li Li and A. Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 171–184, Washington, DC, USA, 1996. IEEE Computer Society.

[45] M. Lindvall. *An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Systems Evolution*. PhD thesis, Linköping University at Linköping, Sweden, 1997.

[46] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.

[47] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, pages 126–139, 2004.

[48] B. Meyer, L. Ciupa, A. Leitner, and L. L. Liu. Automatic testing of object-oriented software. In *Proceedings of the Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129, Berlin, Heidelberg, 2007.

[49] Inc Sun Microsystems. Netbeans ide. At http://www.netbeans.org/, 2009.

[50] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[51] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[52] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, pages 338–345, Boppard, Germany, 1998.

[53] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[54] N. W. Queille, J. F. Voidrot and M. Munro. The impact analysis task in software maintenance: A model and a case study. In *IEEE Conference on Software Maintenance*, Victoria - Canada, 1994.

[55] N. Rachatasumrit and Kim M. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings International Conference on Software Maintenance*, pages 357 –366, Austin, TX USA, 2012.

[56] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004.

[57] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999.

[58] A.R.C. Rocha, J.C. Maldonado, and K.C. Weber. *Qualidade de software: teoria e prática*. Prentice Hall, 2001.

[59] G. Rohit. *A Refinement Theory for Alloy*. PhD thesis, Federal University of Pernambuco, 2007.

[60] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, New York, NY, USA, 2001.

[61] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, pages 286–301, New York, NY, USA, 2010.

[62] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the Workshop on Programming Languages Meets Program Verification*, pages 67–72, New York, NY, USA, 2008.

[63] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 277–294, New York, NY, USA, 2008.

[64] M. Schäfer, M. Verbaere, T. Ekman, and O. Moor. Stepping stones over the refactoring rubicon. In *Proceedings of th European Conference on Object-Oriented Programming*, pages 369–393, Berlin, Heidelberg, 2009.

[65] L. Silva, A. Sampaio, and Z. Liu. Laws of object-orientation with reference semantics. In *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods*, pages 217–226, Washington, DC, USA, 2008.

[66] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, pages 147–162, 2013.

[67] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 2012.

[68] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE Software*, pages 52–57, 2010.

[69] G. Soares, M. Mongiovi, and R. Gheyi. Identifying too strong conditions in refactoring implementations. In *Proceedings of the Conference on Software Maintenance*, 2011.

[70] F. Steimann and A. Thies. From public to private to absent: Refactoring java programs under constrained accessibility. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 419–443, Berlin, Heidelberg, 2009.

[71] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 57–68, New York, NY, USA, 2006.

[72] Inc Embarcadero Technologies. Jbuilder. At http://www.codegear.com/br/products/jbuilder, 2009.

[73] F. Tip. A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, The Netherlands, 1994.

[74] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, pages 89–120, 2001.

[75] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, pages 601–614, 2005.

[76] E. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, pages 35–52, 1994.

[77] Willem V., Corina S. P., and Radek P. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2006.

[78] J. Wloka, B. G. Ryder, and F. Tip. Junitmx - a change-aware unit testing tool. In *Proceedings of the International Conference on Software Engineering*, pages 567–570, Washington, DC, USA, 2009.

[79] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification Reliability*, pages 67–120, 2012.

[80] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 23–32, Washington, DC, USA, 2011.

[81] Y. Zhou, M. Würsch, E Giger, H. C. Gall, and J. Lü. A bayesian network based approach for change coupling prediction. In *Proceedings of the Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2008.

[82] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, pages 366–427, 1997.

Apêndice A

Especificação em Alloy das Leis da Análise de Impacto

Nessa seção, descrevemos a especificação em Alloy utilizada para codificar a formalização matemática das leis da análise de impacto propostas neste trabalho. A seguir, na Listagem A, mostramos a especificação completa.

```
1  fun impactedMethods [source, target: Program] : set Method {
2    {m: Method |
3      contains[m, law1[source, target]] ||
4      contains[m, law2[source, target]] ||
5      contains[m, law3[source, target]] ||
6      contains[m, law4[source, target]] ||
7      contains[m, law5[source, target]] ||
8      contains[m, law6[source, target]]
9    }
10 }
11
12 fun law1[source, target: Program] : set Method {
13   {m: Method | contains[m, addedMethods[source, target]] ||
14     contains[m, exerciseMethod[addedMethods[source, target]]] ||
15     contains[m, inheritedMethods[addedMethods[source, target]]]
16   }
17 }
18
19 fun law2[source, target: Program] : set Method {
```

```
20    {m: Method | contains[m, removedMethods[source, target]] ||
21      contains[m, exerciseMethod[removedMethods[source, target]]] ||
22      contains[m, inheritedMethods[removedMethods[source, target]]]
23    }
24  }
25
26  fun law3[source, target: Program] : set Method {
27    {m: Method | contains[m, changedMethods[source, target]] ||
28      contains[m, exerciseMethod[changedMethods[source, target]]] ||
29      contains[m, inheritedMethods[changedMethods[source, target]]]
30    }
31  }
32
33  fun law4[source, target: Program] : set Method {
34    {m: Method | contains[m, exerciseField[addedFields[source, target]]] ||
35      contains[m, exerciseField[inheritedFields[addedFields[source, target
          ]]]]
36    }
37  }
38
39  fun law5[source, target: Program] : set Method {
40    {m: Method |
41      contains[m, exerciseField[removedFields[source, target]]] ||
42      contains[m, exerciseField[inheritedFields[removedFields[source, target
          ]]]]
43    }
44  }
45
46  fun law6[source, target: Program] : set Method {
47    {m: Method | contains[m, exerciseField[changedFields[source, target]]]
        ||
48      contains[m, exerciseField[inheritedFields[changedFields[source, target
          ]]]]
49    }
50  }
51
52  fun exerciseMethod[impactedMethods: set Method] : set Method {
```

```
53    {m2: Method | some mi: MethodInvocation, m: Method |
54      contains[m, impactedMethods] && m.id = mi.id &&  mi = m2.b
55    }
56  }
57
58  fun exerciseField [fields: set Field]  : set Method {
59    {m: Method | some mi: FieldInvocation, f: Field |
60      containsF[f, fields] && mi.id = f.id && m.b = mi
61    }
62  }
63
64  fun inheritedMethods[impactedMethods: set Method] : set Method{
65    {m2: Method | some c1,c2: Class, m: Method |
66      methodIsNotImplementedBetweenC1andC2[c1,c2,m] &&
67      contains[m, impactedMethods] &&
68      m in c1.methods && m.inherited = false &&
69      C1InheritsAMethodOfC2[c1,c2,m,m2]
70    }
71  }
72
73  pred C1InheritsAMethodOfC2 [c1,c2: Class, m,m2: Method] {
74    c1 in c2.^extend && m2.inherited = true &&
75    m2 in c2.methods && m2.id = m.id
76  }
77
78  pred methodIsNotImplementedBetweenC1andC2[c1,c2: Class, m: Method] {
79    (no c: Class | some m3: Method | (c1 in c.^extend) &&
80      (c2 not in c.^extend) &&  (m3 in c.methods) &&
81        m3.inherited = false && m3.id = m.id && m3.param = m.param)
82  }
83
84  fun inheritedFields[impactedFields: set Field] : set Field{
85    {f2: Field | some c1,c2: Class, f: Field |
86      c1 in c2.^extend &&
87      fieldIsNotImplementedBetweenC1andC2[c1,c2,f] &&
88      containsF[f, impactedFields] && f in c1.fields && f.inherited = false
          &&
```

```
89      C2InheritsAFieldOfC1[c1,c2,f,f2]
90    }
91  }
92
93  pred C2InheritsAFieldOfC1 [c1,c2: Class, f,f2: Field] {
94    c1 in c2.^extend && f2.inherited = true &&
95    f2 in c2.fields && f2.id = f.id
96  }
97
98  pred fieldIsNotImplementedBetweenC1andC2[c1,c2: Class, f: Field] {
99    (no c: Class | some f3: Field | (c1 in c.^extend) &&
100     (c2 not in c.^extend) &&  (f3 in c.fields) &&
101       f3.inherited = false && f3.id = f.id)
102 }
103
104 fun addedMethods [source,target: Program] : set Method {
105   {m: Method | not (contains[m,source.methods]) &&
106     contains[m, target.methods] }
107 }
108
109 fun removedMethods [source,target: Program] : set Method {
110   {m: Method | contains[ m, source.methods] &&
111     not contains[m, target.methods] }
112 }
113
114 fun changedMethods [source,target: Program] : set Method {
115   {m: Method | some m2: Method |
116     contains[m, source.methods] &&
117     contains[m2, target.methods] && equals[m,m2] &&
118     (m.b != m2.b || m.acc != m2.acc )
119   }
120 }
121
122 fun changedFields [source,target: Program] : set Field {
123   {f: Field | some f2: Field |
124     containsF[f, source.fields] &&
125     containsF[f2, target.fields] && f.id = f2.id && f != f2
```

```
126       && (f.acc != f2.acc)
127     }
128   }
129
130   fun addedFields [source,target: Program] : set Field {
131     {f: Field | (not containsF[f, source.fields]) &&
132       containsF[f, target.fields] }
133   }
134
135   fun removedFields [source,target: Program] : set Field {
136     {f: Field | containsF[f, source.fields] &&
137       not containsF[f, target.fields] }
138   }
139
140   pred equals[m1,m2: Method] {
141     some c1,c2: Class | m1 in c1.methods && m2 in c2.methods
142       && c1.id = c2.id && c1 != c2 &&
143       m1.id= m2.id && m1.param = m2.param && m1 != m2
144   }
145
146   fact {
147     MethodId in Method.id
148     Method in Program.methods
149     Field in Program.fields
150     AMethodCanNotBeInMoreThenOneProgram[]
151     noProgramHaveTheSameName[]
152     inheritedMethods[]
153   }
154
155   pred noProgramHaveTheSameName[] {
156     no p1, p2:Program | p1.id = p2.id && p1 != p2
157   }
158
159   pred AMethodCanNotBeInMoreThenOneProgram[] {
160     no m:Method | some p1,p2: Program | p1 != p2
161       && m in p1.methods && m in p2.methods
162   }
```

```
163
164 pred contains[m: Method, methodsSet: set Method] {
165   some m1: Method | some c,c1:Class |
166     m1 in methodsSet && m1.id = m.id
167     && m1.param = m.param
168     && m in c.methods && m1 in c1.methods && c.id = c1.id
169 }
170
171 pred inheritedMethods[] {
172   all m: Method |
173     (m.inherited = true ) => (some m1: Method, c: Class |
174     m in c.methods && m1 in c.^extend.methods &&
175     m1.inherited = false && m.id = m1.id && m.param = m1.param)
176 }
177
178 pred containsF[f: Field, fieldsSet: set Field] {
179   some f1: Field | some c: Class | f1 in c.fields && f in c.fields &&
180     f in fieldsSet && f1 in fieldsSet && f.id = f1.id
181 }
182
183 assert testLaw1 {
184   all m: Method, p1,p2: Program |
185     (not contains[m, p1.methods] && contains[m, p2.methods] ) =>
186       contains[m, impactedMethods[p1,p2]]
187 }
188
189 assert testLaw2 {
190   all m: Method, p1,p2: Program |
191     (contains[m, p1.methods] && not contains[m, p2.methods] ) =>
192       contains[m, impactedMethods[p1,p2]]
193 }
194
195 assert testLaw3 {
196   all m1,m2: Method, p1,p2: Program |
197     (m1 != m2 && p1 != p2 && contains[m1, p1.methods] &&
198       contains[m2, p2.methods] && equals[m1,m2]
199       && (m1.b != m2.b || m1.acc != m2.acc)) =>
```

```
200         contains[m1, impactedMethods[p1,p2]]
201        && contains[m2, impactedMethods[p1,p2]]
202 }
203
204 assert testLaw4 {
205   all f: Field, m: Method, fi: FieldInvocation, p1,p2: Program |
206     (not containsF[f, p1.fields] && containsF[f, p2.fields]
207       && fi.id = f.id && m.b = fi) =>
208         contains[m, impactedMethods[p1,p2]]
209 }
210
211 assert testLaw5 {
212   all f: Field, m: Method, fi: FieldInvocation, p1,p2: Program |
213     (containsF[f, p1.fields] && not containsF[f, p2.fields]
214       && fi.id = f.id && m.b = fi) =>
215         contains[m, impactedMethods[p1,p2]]
216 }
217
218 assert testLaw6 {
219   all f1,f2: Field, m: Method, fi: FieldInvocation, p1,p2: Program |
220     (f1 != f2 && p1 != p2 && containsF[f1, p1.fields] &&
221       containsF[f2, p2.fields] && f1.id = f2.id && f1.acc != f2.acc &&
222         ((fi.id = f1.id && m.b = fi) || (fi.id = f2.id && m.b = fi))) =>
223           contains[m, impactedMethods[p1,p2]]
224 }
```