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

Uma Abordagem para Aumentar a Segurança em Refatoramentos de Programas

Gustavo Araújo Soares

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 e Dalton Serey
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S676a Soares, Gustavo Araújo.
Uma abordagem para aumentar a segurança em refatoramentos de programas /Gustavo Araújo Soares. — Campina Grande, 2010.
131 f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Profº Dr. Rohit Gheyi, Profº Dr. Dalton Serey
Referências.

1. Testes de Programas. 2. Manutenção de Programas. 3. Refatoramento. I. Título.

CDU 004.415.53(043)

"UMA ABORDAGEM PARA AUMENTAR A SEGURANÇA EM REFATORAMENTOS DE PROGRAMAS"

GUSTAVO ARAUJO SOARES

DISSERTAÇÃO APROVADA EM 12.04.2010

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

ROHIT GHEYI, Dr.
Orientador(a)

TIAGO LIMA MASSONI, Dr.
Examinador(a)

PAULO HENRIQUE MONTEIRO BORBA, Ph.D
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Refatoramentos são transformações de programas que preservam o comportamento observável e melhoram a qualidade do código. Cada transformação pode possuir um conjunto de pre-condições que devem ser satisfeitas para que o comportamento seja preservado. Muitas ferramentas como Eclipse e NetBeans automatizam vários desses refatoramentos. Entretanto, essas ferramentas podem realizar transformações errôneas, gerando programas que não compilam (mais simples de serem detectadas) ou com comportamentos diferentes dos originais (mais difíceis de serem detectadas). Isso ocorre porque cada ferramenta implementa os refatoramentos com base em um conhecimento informal do conjunto de pre-condições que precisam ser checadas. Não existe nenhuma teoria formal estabelecendo todas as pre-condições para cada refatoramento em linguagens como Java. Nesse trabalho, nós propomos uma abordagem e sua implementação (SAFEREFACTOR) para aumentar a segurança em refatoramentos de programas seqüenciais em Java. Nossa técnica pode detectar mudanças comportamentais na aplicação de quaisquer refatoramentos em programas seqüenciais. Avaliamos nossa técnica de três formas. Primeiramente aplicamos nossa técnica em um conjunto de 16 transformações aplicadas por ferramentas de refatoramento, já sabia-se, que elas não preservavam o comportamento. Nossa técnica detectou todas as mudanças comportamentais. Além disso, avaliamos em refatoramentos aplicados a sete sistemas reais em Java (3-100 KLOC). Eles foram realizados por desenvolvedores que asseguraram a corretude dos refatoramentos por meio de ferramentas e testes de unidades. Nossa técnica detectou uma mudança comportamental em um refatoramento aplicado ao JHotDraw (23 KLOC). Por fim, propomos o JDolly, um gerador de programas Java, com o objetivo de gerar entradas para testar implementações de refatoramentos. Utilizamos o SAFEREFACTOR e o JDolly para testar automaticamente 11 refatoramentos implementados pelo Eclipse. O JDolly gerou automaticamente um número de entradas que foram refatoradas utilizando cada implementação. O SAFEREFACTOR identificou 50 bugs (35 mudanças comportamentais e 15 erros de compilação) no Eclipse 3.4.2. Manualmente, identificamos que alguns destes bugs ocorrem também no NetBeans 6.7.

Abstract

Program Refactorings are behavior-preserving transformations. Each transformation can have a number of preconditions that must be satisfied to assure the behavior preservation. Many IDEs, such as Eclipse and NetBeans, automate a number of refactorings. However, these tools may perform incorrect transformations that introduce compilation errors (simple to be detected) or behavioral changes (very often go undetected). This happens because each tool implements refactorings based on an informal set of refactoring preconditions. In fact, there is no formal theory that establishes all preconditions for each refactoring in Java. In this work, we propose an approach and its implementation (SAFEREFACTOR) for making Java sequential program refactorings safer. Our technique detects behavioral changes in transformations performed on sequential programs. We evaluated it in three experiments. First, our technique was applied against 16 non-behavior-preserving transformations that are not detected by IDEs that implement refactorings. SAFEREFACTOR detected all behavioral changes. Next, we evaluated it on seven refactorings of real Java programs (from 3 to 100 KLOC) performed by developers that used refactoring tools and unit tests to guarantee the behavior preservation. Our technique detected a behavioral change in a refactoring applied to JHotDraw (23 KLOC). Finally, we propose a Java program generator (JDolly) in order to generate program inputs to refactoring implementations. We used SAFEREFACTOR and JDolly to test 11 refactorings implemented by Eclipse 3.4.2. JDolly automatically generated a number of programs to be refactored using Eclipse API. SAFEREFACTOR identified 50 *bugs* in these transformations (35 behavioral changes and 15 compilation errors). We manually identified that some of these bugs also happen in NetBeans 6.7.

Agradecimentos

Primeiramente, quero agradecer a Deus por estar ao meu lado e ter me dado a oportunidade de realizar esse mestrado. Também não teria conseguido essa conquista sem a contribuição de algumas pessoas. Abaixo, meus agradecimentos:

- a minha família por ter me dado todas as condições para que eu pudesse terminar o mestrado, além de sempre me apoiar e incentivar. Em especial, minha irmã Aninha por ter convivido comigo em Campina Grande durante esse período;
- a Rafaela, minha namorada, por ter ficado ao meu lado e me apoiado em mais etapa da minha vida;
- aos meus orientadores e amigos professor Rohit e professor Dalton pela preocupação com minha formação e por seus ensinamentos;
- aos professores Tiago Massoni, Márcio Cornélio, e Paulo Borba pelas sugestões e contribuições no meu trabalho;
- a todos os membros do GMF pelas contribuições na minha pesquisa e pela amizade que construímos;
- aos professores e funcionários da COPIN;
- ao INES e CNPq por apoiarem meu projeto.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Durante o ciclo de vida de um *software*, sua manutenção e evolução são atividades inevitáveis. Novos requisitos são solicitados pelos clientes depois de seu lançamento. Além disso, defeitos são descobertos e precisam ser corrigidos. Quanto mais o *software* é modificado, e adaptado para novos requisitos, seu código torna-se mais complexo, tornando mais difícil sua manutenção. Para evitar que isso ocorra, é necessário reestruturá-lo, melhorando sua estrutura interna, mas preservando suas funcionalidades; é o tipo de manutenção a que se denomina perfectiva [2]. Em programas orientados a objetos, o processo de alterar a estrutura de um programa para melhorar alguma qualidade sem alterar seu comportamento externo é conhecido como refatoramento. Esse termo foi cunhado por por Opdyke e Johnson [55; 54], e posteriormente, popularizado na prática por Fowler [21].

Para cada refatoramento, existe um conjunto de pre-condições que garante a preservação do comportamento do programa. Por exemplo, o refatoramento *Push Down Method* [21] move um método da classe pai para a classe filha. Para aplicá-lo, temos que checar se já existe outro método com a mesma assinatura declarado na classe filha para evitar um erro de compilação após a transformação. Fowler [21] 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 [60] 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 Integrado (*IDE*) utilizados no desenvolvimento de sistemas, como Eclipse [18], Netbeans [71], JBuilder [20], IntelliJ [38], possuem um módulo para automatizar refatora-

mentos.

1.1 Problema

Ferramentas de refatoramento não são totalmente confiáveis. Elas ajudam a automatizar a transformação e a checar parcialmente as pre-condições. Porém, elas podem realizar transformações errôneas, gerando programas que não compilam ou com comportamentos diferentes dos originais. Atualmente, cada IDE implementa os refatoramentos de programa com base em um conhecimento informal do conjunto de pre-condições que precisam ser checadas. Uma comprovação deste fato é que algumas IDEs permitem aplicar um refatoramento de maneira errônea, mas outras não [19; 14]. A tendência é que esse cenário ruim permaneça, pois estabelecer todas as pre-condições para um refatoramento em uma linguagem complexa como Java é um trabalho não trivial, e ainda é tido como um desafio [62]. Alguns trabalhos vêm contribuindo nessa linha. Por exemplo, Borba et al. [4; 5; 13] propuseram um conjunto de refatoramentos para a linguagem Refinement Object-Oriented Language (ROOL), um subconjunto de Java seqüencial só que com semântica de cópia. Estes refatoramentos são provados corretos com relação a uma semântica formal. Entretanto, não existe nenhum trabalho especificando todas as pre-condições para refatoramentos em Java. Portanto, os desenvolvedores de IDEs com suporte a refatoramentos têm que escolher entre implementarem algo bem restrito e que não tem erro, ou algo bem geral e útil na prática, mas que pode possuir erros.

Daniel et al. [14] propuseram uma técnica para testar ferramentas de refatoramentos de forma automatizada. Como resultado, eles encontraram 21 *bugs* no Eclipse e 24 *bugs* no Netbeans. A maioria dos *bugs* encontrados foi relacionada a introdução de erros de compilação. Já outros trabalhos [61; 19; 70] apresentam várias transformações aplicadas por IDEs que não preservam o comportamento observável. Essas transformações foram descobertas com base na experiência dos autores; eles não propuseram nenhuma abordagem sistemática para descobri-las.

Na prática, os desenvolvedores refatoram seus programas utilizando ferramentas de refatoração, e em seguida executam testes para checar se ocorreu alguma mudança comportamental. Caso algum teste falhe, as IDEs oferecem a funcionalidade de *undo* para desfazer a

transformação. Porém, alguns refatoramentos invalidam os testes mesmo sem alterar o comportamento do programa [47]. Isso acontece porque alguns testes dependem do código do programa. Por isso, freqüentemente eles precisam ser refatorados para se adequarem ao novo código. Algumas IDEs, como o Eclipse, refatoram automaticamente os testes de unidade em alguns refatoramentos. Contudo, da mesma forma que as ferramentas podem introduzir erros ao refatorar um programa, elas também podem introduzir erros na coleção de testes, fazendo com que a mudança comportamental no programa não seja detectada.

1.2 Exemplo Motivante

Nessa seção, mostramos dois exemplos de refatoramentos automatizados que introduzem erros de compilação ou mudanças comportamentais no programa. Primeiramente, considere o programa exibido no Código Fonte 1.1 [67]. A classe `B` estende `A` e possui um método `k()`. A classe `C` possui um atributo do tipo `A` e um método `k()`.

Figura 1.1: *Move method* reduz a visibilidade de um método herdado introduzindo um erro de compilação

Código Fonte 1.1: Versão Original

```
public class A {
}
public class B extends A {
  protected long k(){
    return 22;
  }
}
public class C {
  public A a;
  public long k(){
    return 17;
  }
}
```

Código Fonte 1.2: Versão Refatorada

```
public class A {
  public long k(){
    return 17;
  }
}
public class B extends A {
  protected long k(){
    return 22;
  }
}
public class C {
  public A a;
}
```

Se aplicarmos o refatoramento *Move Method* utilizando o Eclipse 3.4 para mover o mé-

todo `k()` da classe `C` para `A`, a ferramenta produz o programa exibido no Código Fonte 1.2. Após a transformação, o programa não compila, pois o método `k()` em `B` está reduzindo a visibilidade do método `k()` declarado em `A`, o que não é permitido em Java. Perceba que ao mover o método, checar se já existia um método com o mesmo nome na classe de destino não foi suficiente para evitar o erro de compilação. Definir todas as pre-condições para que um refatoramento não introduza erros de compilação é difícil. Porém, esses tipos de erros são mais simples de serem detectados que os erros de mudança comportamental (basta checar se o programa refatorado compila). A seguir, mostramos um exemplo de uma mudança comportamental introduzida no programa por um refatoramento automatizado.

Figura 1.2: Push Down Method pode habilitar sobrecarga de método introduzindo uma mudança comportamental

Código Fonte 1.3: Versão Original

```
public class A {
  public int k(long l) {
    return 23;
  }
  public int m() {
    return k(2);
  }
}
public class B extends A {
  public int k(int i) {
    return 42;
  }
  public int test() {
    return m();
  }
}
```

Código Fonte 1.4: Versão Refatorada

```
public class A {
  public int k(long l) {
    return 23;
  }
}
public class B extends A {
  public int k(int i) {
    return 42;
  }
  public int test() {
    return m();
  }
  public int m() {
    return k(2);
  }
}
```

Suponha um programa com as classes `A` e `B`, como mostrado no Código Fonte 1.3 [67]. O método `m()` é declarado em `A`. Ele invoca o método `k(long)` com a expressão `k(2)`; nessa situação, Java faz um *cast* implícito [31] no valor `2` de `int` para `long`. O método

`test()` declarado em `B` faz uma chamada a `m()`. Note que `m()` herdado em `B` tem acesso aos métodos `k(int)` e `k(long)`, sendo que ele dá preferência ao último, pois o valor de `2` foi convertido implicitamente para `long`. Por isso, o método `test()` retorna o valor de `23`. Ao aplicarmos o refatoramento *Push Down Method* no método `m()` utilizando o Eclipse 3.4 ou o Netbeans 6.7, o programa refatorado fica similar ao exibido no Código Fonte 1.4. Ambas movem o método para a classe filho e não fazem mais nenhuma alteração no programa. Essa transformação não preserva o comportamento. O método `test()` no programa refatorado retorna o valor de 42, ao invés de 23. A mudança comportamental ocorre porque ao mover `m()` para `B`, o valor `2` na expressão contida no corpo desse método não é mais convertido implicitamente para `long`. Sendo assim, `m()` passa a chamar `k(int)` ao invés de `k(long)`.

O exemplo acima é pequeno, e uma coleção de testes pode identificar a mudança comportamental. Porém, ele é suficiente para mostrar a complexidade da linguagem Java, pois, apesar do programa possuir apenas 2 classes e poucas linhas de código, as mais utilizadas IDEs dessa linguagem não conseguem detectar o problema. Em programas maiores, a verificação torna-se mais difícil.

1.3 Solução

Nesse trabalho, nós propomos uma abordagem para aumentar a segurança em refatoramentos de programa. Nossa técnica gera automaticamente um conjunto de testes de unidade específicos para detectar mudanças comportamentais em refatoramentos de programas sequenciais orientados a objetos. Os testes são gerados apenas para os métodos em comum entre a versão original e a refatorada do programa. Dessa forma, podemos executar a mesma coleção de testes nas duas versões. Se algum teste passar em uma versão e falhar na outra, nós detectamos uma mudança comportamental. Caso contrário, aumentamos a confiança de que a transformação está correta. Desenvolvemos uma ferramenta chamada SAFEREFACTOR que implementa nossa técnica.

Propomos também um gerador de programas Java (JDolly) com o objetivo de gerar automaticamente entradas para testar implementações de refatoramentos. Ele é baseado na linguagem de especificação formal Alloy [36], e no ASTGen [14], um *framework* para gera-

ção de ASTs Java. O JDolly recebe como entrada o número de classes, atributos, e métodos que os programas devem possuir, e gera automaticamente um número de programas Java. Além disso, o usuário pode especificar restrições em Alloy sobre as características estruturais desejadas para os programas.

Adicionalmente, propomos uma abordagem para automatizar os testes de ferramentas de refatoramento utilizando a nossa técnica para detectar mudanças comportamentais em refatoramentos, e o nosso gerador de programas. Os programas gerados pelo JDolly são automaticamente refatorados utilizando a API de refatoramento do Eclipse, e o SAFEREFACTOR checa a existência de erros de compilação ou mudança comportamental em cada programa refatorado.

1.4 Avaliação

Primeiramente, avaliamos nossa técnica para detectar mudanças comportamentais em um experimento com um conjunto contendo 16 transformações aplicadas por ferramentas de refatoramento que foram catalogados na literatura [19; 70]. Essas transformações alteram o comportamento do programa. Nós avaliamos a eficiência de nossa técnica em detectar essas mudanças comportamentais. Ela detectou todas as mudanças comportamentais.

Em seguida, realizamos um experimento envolvendo a aplicação de refatoramentos em sete sistemas reais em Java (3-100 KLOC). Utilizamos nossa técnica para checar se as transformações preservavam o comportamento dos programas. Ela detectou uma mudança comportamental em um refatoramento aplicado ao JHotDraw, um programa com mais de 23 KLOC. Esse refatoramento foi aplicado com auxílio de IDEs com suporte a refatoramento e testes de unidade por desenvolvedores programando em pares e com revisão sistemática do código. Eles acreditavam que a transformação realizada preservava o comportamento do programa.

Por último, utilizamos nossa técnica e o gerador de programas para testar o módulo de refatoramento do Eclipse 3.4. Geramos aproximadamente 75.000 programas e testamos 11 implementações de refatoramentos do Eclipse. Identificamos 50 *bugs* [67] distintos nessa IDE, sendo que 35 deles provocam mudanças comportamentais e 15 introduzem erros de compilação. Checamos manualmente que muitos desses *bugs* também existem nos refato-

ramentos implementados pelo Netbeans. Além disso, baseado nesses *bugs*, identificamos problemas nas leis de refatoramentos para Java propostas por Duarte [17].

1.5 Resumo de Contribuições

Em resumo, as principais contribuições desse trabalho são:

- uma técnica para tornar a atividade de refatoramento de programas mais segura [69; 65; 68; 66], apresentada no Capítulo 3;
- um gerador de programas Java [65], apresentado no Capítulo 4;
- uma abordagem para automatizar o teste de ferramentas de refatoramento [65], como o Eclipse, apresentada na Seção 5.3;
- uma avaliação de 16 transformações que introduzem mudanças comportamentais que não são detectadas por algumas das IDEs com suporte a refatoramento mais utilizadas [69; 65; 68] (Seção 5.1);
- uma avaliação de 7 refatoramentos aplicados em programas reais por desenvolvedores que asseguraram, usando ferramentas de refatoramentos e testes de unidade, que as transformações preservavam o comportamento [69; 65] (Seção 5.2);
- uma avaliação de 11 implementações de refatoramentos do Eclipse [65; 67] (Seção 5.3);
- um plugin para o Eclipse para aumentar a segurança dos refatoramentos automatizados por essa IDE [69; 66; 65] (Capítulo 6).

1.6 Organização

No capítulo seguinte, apresentamos a fundamentação teórica necessária para o entendimento do nosso trabalho (Capítulo 2). No Capítulo 3, aprestamos nossa técnica para detectar mudanças comportamentais em refatoramentos, e no Capítulo 4, mostramos nosso gerador de programas Java. Em seguida (Capítulo 5), descrevemos a avaliação da nossa técnica e do

gerador. Adicionalmente, no Capítulo 6, nós descrevemos a implementação da nossa técnica. Por fim, apresentamos as considerações finais e trabalhos futuros, além dos trabalhos relacionados (Capítulo 7).

Capítulo 2

Fundamentação Teórica

Nesse capítulo, apresentamos a fundamentação teórica necessária para o entendimento do nosso trabalho. Inicialmente, mostramos uma visão geral da fase de manutenção e evolução de software (Seção 2.1). Em seguida, apresentamos os conceitos fundamentais de refatoramento, bem como o estado da arte e problemas em aberto (Seção 2.2). Posteriormente, na Seção 2.3, descrevemos a linguagem de especificação formal Alloy [36], e, na Seção 2.4, o *framework* para geração de programas Java ASTGen.

2.1 Manutenção e Evolução de Software

Durante ciclo de vida de um *software*, sua manutenção e evolução são atividades inevitáveis para adequá-lo a novos requisitos e corrigir defeitos. A maior parte do custo total de um *software* é relativo a essa atividade [10]. Manutenção de software é definida pelo padrão IEEE 1219 [1] como: "A modificação de um produto de *software* depois de seu lançamento para corrigir falhas, aumentar seu desempenho ou outras qualidades, ou adaptar o produto para um ambiente alterado". Evolução de software não possui uma definição padrão, mas muitos pesquisadores e profissionais preferem esse termo à manutenção de *software*.

As mudanças realizadas no sistema após sua entrega podem ser relacionadas a diversas causas. Podemos ter mudanças simples de correção de código, até mudanças mais custosas para corrigir o *design* do programa, ou melhorá-lo para adicionar novas funcionalidades. O padrão de engenharia de software e manutenção [2] divide essas mudanças em quatro categorias:

1. **manutenção corretiva.** Modificações realizadas no software para corrigir problemas encontrados. Essa atividade permite corrigir as funcionalidades utilizadas pelo usuário;

2. **manutenção adaptativa.** Modificações realizadas no software para adaptá-lo à adição ou modificação de requisitos de negócio;

3. **manutenção perfectiva.** Modificações no software para melhorar seu desempenho ou sua manutenibilidade. Neste último, as modificações são realizadas para melhorar o entendimento do sistema para facilitar a aplicação de uma mudança adaptativa ou corretiva;

4. **manutenção preventiva.** Modificações no software para prevenir problemas futuros.

Uma pesquisa conduzida por Lientz e Swanson [43] sugere que 65% das manutenções de *software* estão relacionadas com a adição de novas funcionalidades (manutenção adaptativa). Para facilitar a adição dessas novas funcionalidades, pode ser feita uma reestruturação no programa (manutenção perfectiva). Chikofsky e Cross [7] definem reestruturação como "*uma transformação de uma representação para outra com o mesmo nível de abstração, preservando o comportamento externo do sistema (funcionalidades e semântica)*".

2.2 Refatoramento de Programas

O termo refatoramento (*refactoring*) foi cunhado por Opdyke em sua tese de doutorado [54] no início dos anos 90. Refatoramento pode ser visto como uma variação de reestruturação para o caso específico de sistemas orientados a objetos. Fowler [21] define refatoramento como:

> Refatoramento é uma mudança realizada na estrutura interna do programa para deixá-lo mais fácil de ser entendido e mais barato de ser modificado, sem alterar seu comportamento observável. (p. 53)

Fowler também define o verbo *refatorar* como [21]:

> Refatorar é reestruturar o programa aplicando uma série de refatoramentos sem alterar seu comportamento observável. (p. 54)

2.2.1 Exemplo

Nessa seção, exibimos um exemplo de um refatoramento de programa. Primeiramente, mostramos a identificação da parte do programa que deve ser refatorada, e em seguida, descrevemos a escolha e aplicação do refatoramento. Para esse exemplo, considere a classe pai `Empregado` e suas classes filhas `Engenheiro` e `Analista`, exibidas no Código Fonte 2.1.

Código Fonte 2.1: Programa com códigos duplicados

```
public class Empregado {
  ...
}
public class Engenheiro extends Empregado {
  private double salario;
  public double getSalario() {
    return salario;
  }
  ...
}
public class Analista extends Empregado {
  private double salario;
  public double getSalario() {
    return salario;
  }
  ...
}
```

Bad Smells

Primeiramente, devemos identificar as partes do código que precisam ser refatoradas. Para ajudar ao desenvolvedor nessa etapa, Beck [21] categorizou 21 situações em que há indícios no código de que ele precisa ser melhorado. Beck refere-se a essas situações como *bad Smells*. O primeiro *bad smell* que ele apresenta é o código duplicado. Quando o mesmo trecho de código aparece em mais de uma parte do programa, a manutenção desse código torna-se mais difícil, pois é necessário realizar a mudança em todas as partes em que ele aparece. Por isso, o melhor é encontrar uma maneira de unificar o código. Ao inspecionar

o código exibido no Código Fonte 2.1, percebemos que o método `getSalario()` e o atributo `salario` estão declarados nas duas últimas classes. Devemos refatorar esse código para eliminar essa duplicidade.

Outros exemplos de *bad smells* são: métodos longos, classes grandes, e lista extensa de parâmetros [21; 75].

Escolha e Aplicação dos Refatoramentos

Fowler [21; 59] define um catálogo de refatoramentos contendo uma motivação e o mecanismo para a aplicação de cada um deles. Nesse exemplo, aplicaremos dois desses refatoramentos para remover o código duplicado.

Primeiramente, utilizamos o refatoramento *Pull Up Field* para mover os atributos para a classe pai. Fowler [21] define os seguintes passos para a aplicação desse refatoramento:

1. inspecione todos os usos dos atributos candidatos para assegurar que eles são inicializados da mesma forma;
2. se os atributos não tiverem nomes iguais, renomeie-os para que todos tenham o nome desejado para o atributo da super classe;
3. compile e teste;
4. crie um novo atributo na classe pai. Se os atributos forem privados, você terá que declará-lo como `protected` para que as classes filhos tenham acesso a ele.
5. remova os atributos das classes filhos;
6. compile e teste.

Aplicamos o refatoramento usando pequenos passos intercalados de compilação e testes para garantir que a transformação preserve o comportamento observável do programa. Se após uma alteração, o código não compilar ou os testes falharem, nós desfazemos a mudança.

Os testes podem ser realizados utilizando o JUnit [46], um *framework* para automatização de testes de unidade. Ele fornece o método `assertEquals` que compara o valor do método retornado na execução do programa com o valor esperado. Se os valores forem diferentes, o teste falha e uma barra vermelha é exibida na interface gráfica do JUnit; caso

contrário, ele passa e uma barra verde é exibida. Criamos um teste de unidade para o o método `getSalario()` da classe `Analista`. Nesse teste, instanciamos um objeto do tipo `Analista`, atribuímos um valor para o atributo `salário`, e comparamos esse valor com o valor retornado pelo método `getSalario` (Código Fonte 2.2). Criamos um teste similar para o método `getSalario` classe `Engenheiro` (Código Fonte 2.3). A Figura 2.1 exibe a interface do Junit mostrando o resultado da execução desses testes.

Código Fonte 2.2: Teste de unidade para o método getSalario() da classe Analista

```
public void testGeSalario() {
  Analista analista = new Analista();
  analista.setSalario(3000);
  double valorEsperado = 3000;
  double valorRetornado = analista.getSalario();
  assertEquals(valorEsperado, valorRetornado);
}
```

Código Fonte 2.3: Teste de unidade para o método getSalario() da classe Engenheiro

```
  Engenheiro engenheiro = new Engenheiro();
  engenheiro.setSalario(2500);
  double valorEsperado = 2500;
  double valorRetornado = engenheiro.getSalario();
  assertEquals(valorEsperado, valorRetornado);
}
```

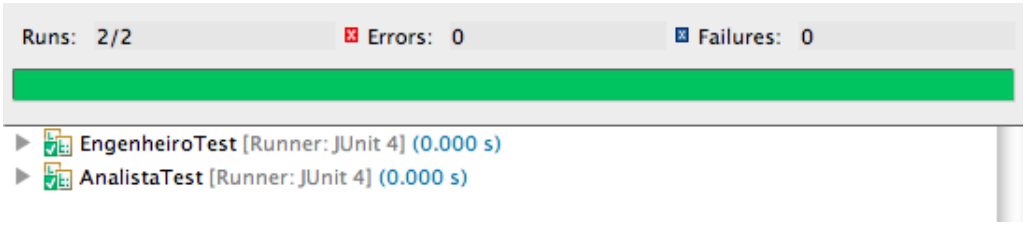

Figura 2.1: Interface gráfica do JUnit exibindo o resultado dos testes de unidade

O Código Fonte 2.4 exibe o programa após o refatoramento. Perceba que foi necessário, como previsto no Passo 4, alterar a visibilidade do atributo de `private` para `protected` para permitir seu acesso pelas classes filhas.

Após remover o atributo duplicado, aplicamos o refatoramento *Pull Up Method* [21] para mover o método `getSalario` para a classe pai. Podemos aplicar esse refatoramento porque esse método possui o mesmo comportamento nas duas classes filhas. O Código Fonte 2.5 exibe o resultado final do programa após os dois refatoramentos.

Código Fonte 2.4: Programa refatorado com o Pull Up Field

```
public class Empregado {
  protected double salario;
  ...
}
public class Engenheiro extends Empregado {
  public double getSalario() {
    return salario;
  }
  ...
}
public class Analista extends Empregado {
  public double getSalario() {
    return salario;
  }
  ...
}
```

Código Fonte 2.5: Programa refatorado com Pull Up Method

```
public class Empregado {
  protected double salario;
  public double getSalario() {
    return salario;
  }
  ...
}
public class Engenheiro extends Empregado { ... }
public class Analista extends Empregado { ... }
```

Aplicar refatoramentos manualmente é uma atividade custosa e propícia a erros. Fowler [21] sugeriu passos sistemáticos para aplicar cada refatoramento de forma mais segura. Além disso, existem ferramentas que automatizam essa atividade. A seguir, mostramos uma

visão geral dessas ferramentas.

2.2.2 Ferramentas de Refatoramentos

A primeira *ferramenta de refatoramento*, Refactoring Browser [60], foi proposta por Roberts em sua tese de doutorado. Ela implementa alguns refatoramentos para a linguagem Smalltalk [29]. Refatoramentos vem ganhando popularidade com a inclusão das ferramentas de refatoramentos nas IDEs de desenvolvimento de *software*. A Tabela 2.1 exibe algumas ferramentas de refatoramentos para a linguagem Java [75].

Ferramenta	Empresa	Tipo	URL
Borland Together	Borland	Ferramenta UML e Java com suporte a refatoramentos	www.borland.com
CodeGuide	Omnicore	IDE	www.omnicore.com
Eclipse		IDE	www.eclipse.org
Idea	IntelliJ	IDE	www.intellij.com
JavaRefactor		Plugin para jEdit	plugins.jedit.org/plugins/?JavaRefactor
JBuilder	Borland	IDE	www.borland.com/jbuilder
JFactor	Instatiations	Plugin para JBuilder e VisualAge	www.instatiations.com/jfactor
JRefactory		Plugin para Elixir, JBuilder e NetBeans	jrefactory.sourceforge.net
NetBeans	Sun Microsystems	IDE	www.netbeans.org
Transmogrify		Plugin para JBuilder e Forte4Java	transmogrify.sourceforge.net
XRefactory	Xref-Tech	Plugin para Emacs, jEdit e XEmacs	www.xref-tech.com

Tabela 2.1: Ferramentas de refatoramentos para Java [75]

Nas ferramentas de refatoramentos, o programador seleciona qual refatoramento deve ser aplicado e os parâmetros. A ferramenta automaticamente checa as pre-condições do refatoramento, por exemplo, ao aplicar o *Rename Method*, ela checa a existência de outros métodos com o mesmo nome do método renomeado. Se todas as pre-condições forem satisfeitas, ela aplica a transformação. Para exemplificar, mostramos o processo para aplicar o refatoramento *Pull Up Field* visto na Seção 2.2.1 utilizando o Eclipse. O desenvolvedor seleciona o atributo que será refatorado, e escolhe *Pull Up* no menu *Refactor* (Figura 2.2(a)). O Eclipse exibe uma janela para o desenvolvedor selecionar parâmetros adicionais e aplicar o refatoramento (Figura 2.2(b)).

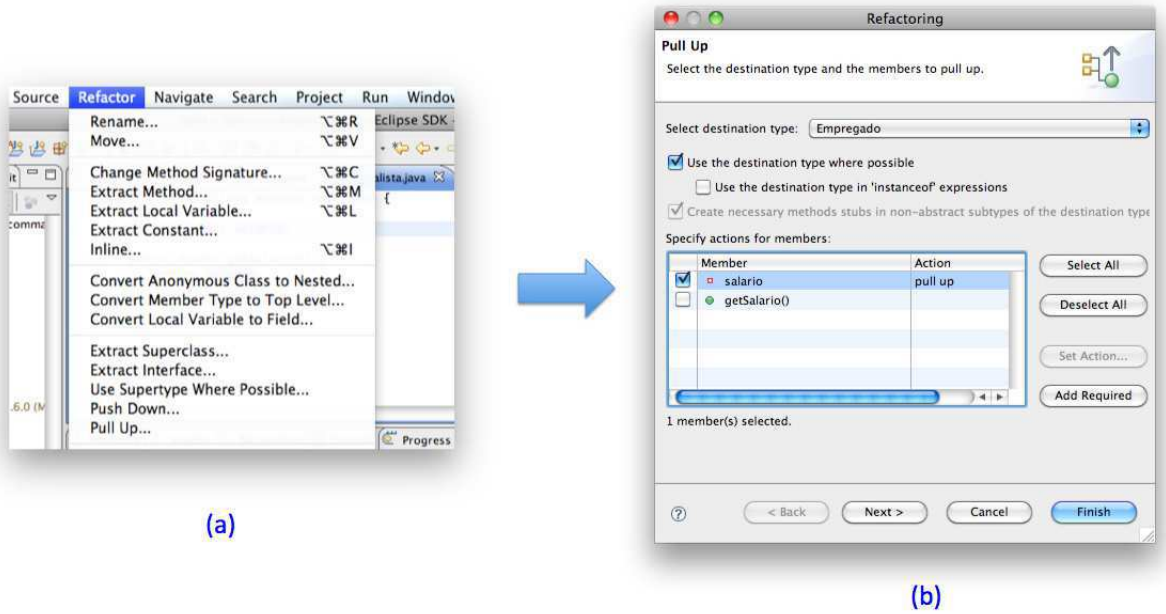

Figura 2.2: Pull Up Field no Eclipse. (a) desenvolvedor seleciona o refatoramento desejado; (b) desenvolvedor configura parâmetros adicionais e confirma o refatoramento apertando o botão OK

Adicionalmente, o Eclipse permite visualizar uma prévia do refatoramento ao clicar no botão *next* (Figura 2.2(b)). Isso permite ao desenvolvedor inspecionar manualmente se a transformação está correta. A Figura 2.3 exibe as alterações que serão realizadas em cada classe afetada pelo refatoramento. Note que o Eclipse automaticamente altera a visibilidade do atributo `salario` de `private` para `protected`.

O Eclipse foi uma das primeiras IDEs a implementar refatoramentos. Sua primeira versão, lançada no final do ano 2001, incluía os refatoramentos: *Rename*, *Move*, e *Extract Method* [23]. Os refatoramentos implementados pelo Eclipse 3.4 podem ser vistos na Tabela 2.2. Murphy et al. [51] conduziram uma pesquisa sobre o desenvolvimento de programas Java utilizando o Eclipse. Eles analisaram o uso da ferramenta de refatoramento do Eclipse por 41 desenvolvedores. Os cinco refatoramentos mais usados segundo essa pesquisa são: *Rename*, *Move*, *Extract Method*, *Pull Up Method*, e *Add Parameter*.

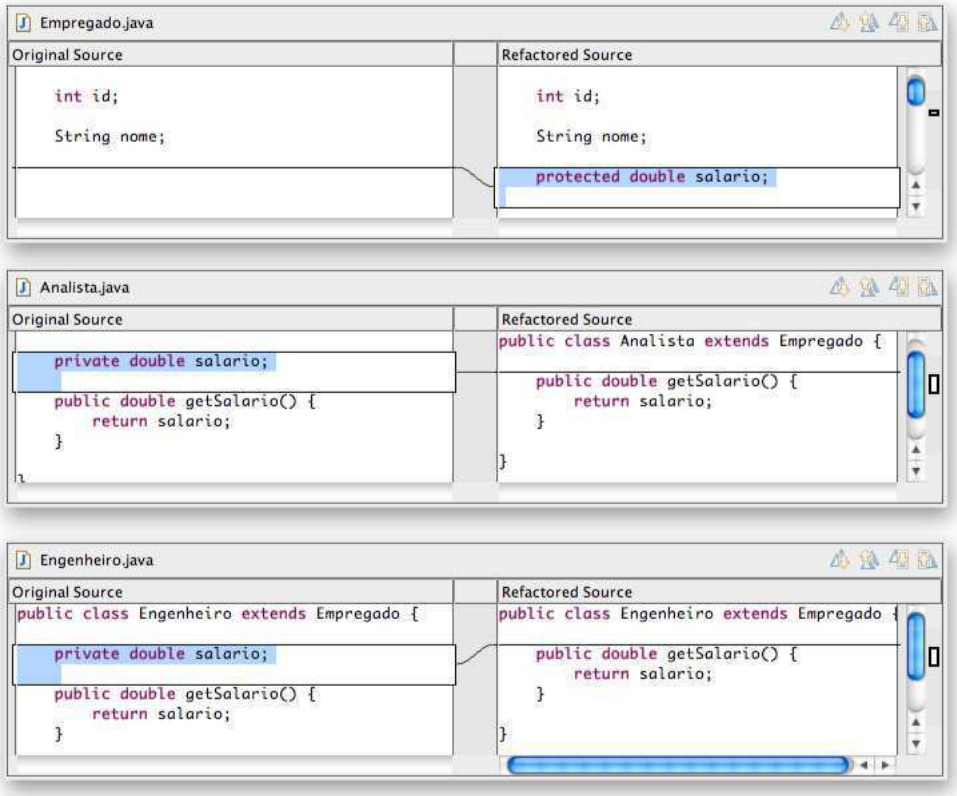

Figura 2.3: Prévia do refatoramento fornecido pelo Eclipse

2.2.3 Preservação do Comportamento

De acordo com a definição de refatoramento apresentada na Seção 2.2, dois programas são equivalentes se eles possuírem o mesmo *comportamento observável*. Em sua tese de doutorado, Opdyke definiu formalmente 23 refatoramentos primitivos e mais três compostos [21]. Cada refatoramento primitivo tem um conjunto de pre-condições que garantem a preservação do comportamento do programa. Por exemplo, Opdyke define as seguintes pre-condições para o refatoramento *Pull Up Field* apresentado na Seção 2.2.1:

1. o atributo tem que estar definido igualmente em todas as subclasses em que ele é definido;

2. o atributo não pode estar definido na classe pai.

Note que se a segunda pre-condição for violada, a transformação produzirá um programa com erro de compilação, pois ao subir o atributo para a classe pai, ocorrerá um conflito

Refatoramentos Implementados pelo Eclipse			
Rename (package, class, method, field)	Move (class, method)	Change Method Signature	Extract Method
Extract Local Variable	Extract Constant	Inline (method, variable)	Convert Anonymous Class to Nested
Convert Member Type to Top Level	Extract Superclass	Extract Interface	Use Super Type Where Possible
Push Down (method, field)	Pull Up (method, field)	Extract Class	Introduce Parameter Object
Introduce Indirection	Introduce Factory	Introduce Parameter	Encapsulate Field
Generalize Declared Type	Infer Generic Type Arguments		

Tabela 2.2: Refatoramentos Implementados pelo Eclipse

entre esse atributo e o atributo previamente declarado na classe. Por outro lado, se a primeira pre-condição for violada, o programa pode continuar compilando, mas pode ter um comportamento diferente, pois o valor do atributo de uma das classes será alterado.

As pre-condições propostas por Opdyke são baseadas em sete propriedades definidas por ele. Segundo ele, essas propriedades asseguram a corretude da transformação. São elas:

1. *única classe pai*. Cada classe do programa resultante deve ter no máximo uma classe pai;

2. *diferentes nomes de classes.* Todas as classes do programa resultante devem possuir nomes diferentes;

3. *diferentes nomes de membros.* Cada classe no programa resultante deve possuir variáveis e métodos com nomes diferentes;

4. *não redefinir variáveis de membros herdados.* Uma variável de um membro herdado de uma classe pai não é redefinida em nenhuma classe filha;

5. *assinaturas compatíveis nas funções redefinidas.* Funções redefinidas devem possuir assinaturas compatíveis;

6. *expressões tipadas corretamente.* No programa resultante, toda expressão que é atribuída a uma variável devem possuir o mesmo tipo ou subtipo da variável;

7. *semântica equivalente de referência e operação.* Dado um conjunto de entradas, o programa resultante deve possuir o mesmo conjunto de saída do programa original.

As seis primeiras propriedades são relacionadas com a preservação das regras de boa formação sintática dos programas. Podemos checar isso recompilando o programa após a transformação. Se houver algum *erro de compilação*, significa que a transformação não foi aplicada corretamente. Por outro lado, a última propriedade é relacionada com a preservação da semântica do programa, por isso, compilar o programa não é suficiente. O programa pode continuar compilando, mas com um comportamento diferente do programa original (*erro de mudança comportamental*).

Opdyke [54] define noção de equivalência semântica entre o programa original e refatorado da seguinte forma: "*Seja a função main a interface externa do programa. Se a função main é chamada duas vezes (antes e depois do refatoramento) com o mesmo conjunto de entradas, a saída resultante deve ser a mesma.* (p. 40)". Essa definição de noção de equivalência permite que um refatoramento altere a estrutura interna do programa, desde que o mapeamento de entradas e saídas da função *main* seja preservado. Ela pode ser vista como uma aplicação da noção de refinamento de dados [32; 34].

Outra forma de lidar com a preservação do comportamento em refatoramentos é através de testes. Roberts [60] define que um refatoramento é correto se um programa após o refatoramento continua em conformidade com sua especificação. Ele considera uma coleção de testes como uma forma de especificação. Dessa forma, sua definição de equivalência é baseada em testes. Para ele, um refatoramento é correto se um programa que passa nos testes continuar a passar após a transformação. Fowler [21] possui essa mesma noção de equivalência.

Adicionalmente, em alguns domínios de aplicação, garantir que para um conjunto de entradas, o programa transformado possui o mesmo conjunto de saída do original é insuficiente para afirmar que a transformação preservou o comportamento [47]. Por exemplo, em sistemas de tempo real, um aspecto essencial para o comportamento é o tempo de execução do programa. Por outro lado, em sistemas embarcados, pode ser necessário preservar a memória e consumo de energia para manter o comportamento do programa.

2.2.4 Problema

Vimos que cada refatoramento possui pre-condições para garantir o preservamento do comportamento do programa. Porém, estabelecê-las formalmente é um trabalho não trivial. Por exemplo, Tokuda [74] percebeu, ao implementar refatoramentos em C++, que as sete propriedades propostas por Opdyke não eram suficientes para preservar o comportamento do programa. Ele propôs mais três:

1. *implementação de funções virtuais puras.* Se uma classe é instanciada, ela não pode ter nenhuma função virtual-pura;

2. *manter objetos agregados.* Se um programa depende de uma propriedade agregada de um objeto, essa propriedade deve ser preservada;

3. *nenhum efeito-colateral de instanciação.* Se um refatoramento pode alterar a freqüência ou ordem na qual as classes são instanciadas, então os construtores não podem ter efeitos-colaterais além da inicialização dos objetos criados.

Com relação a Java, não existe nenhum trabalho que especifica de forma completa as pre-condições para refatoramentos. Por isso, a maioria das IDEs com suporte a refatoramentos, tais como Eclipse, NetBeans, JBuilder, não implementam todas as pre-condições. Essas ferramentas podem realizar transformações errôneas, gerando programas que não compilam ou com comportamento diferente do original.

O exemplo a seguir mostra um erro de compilação introduzido pelo Eclipse 3.4 ao aplicar o refatoramento *Encapsulate Field*. O Código Fonte 2.6 exibe um programa com a classe `A` e sua subclasse `B`. Em `A`, temos o atributo `i`, enquanto que em `B` temos o método `getI()`. Ekman et al. [19] identificaram que após aplicar o refatoramento *Encapsulate Fied* [21] em `i` utilizando o Eclipse 3.4, o código resultante não compila (Código Fonte 2.7). A IDE cria o método público `getI()` na classe `A`, porém, este não pode ser sobrescrito por um método com menor visibilidade (`B.getI()`), gerando um erro de compilação. Erros de compilação são mais fáceis de serem detectados; basta compilar o código resultante e checar se possui erros.

Por outro lado, o exemplo a seguir mostra um erro identificado por Ekman et al. [19] de uma ferramenta de refatoramento que introduz uma mudança comportamental no pro-

Figura 2.4: *Encapsulate Field* introduz erro de compilação

Código Fonte 2.6: Versão Original

```
public class A {
  public int i;
}
public  class B extends A {
  private int getI() {
    return 42;
  }
}
```

Código Fonte 2.7: Versão Refatorada

```
public class A {
  private int i;
  public int getI() {return i;}
  public void setI(int i) {
    this.i = i; }
 }
public  class B extends A {
  private int getI() {
    return 42; }
 }
```

grama. Suponha que desejamos alterar o nome do atributo `myPI` para `PI` na classe exibida no Código Fonte 2.8.

Código Fonte 2.8: Código que sobre uma mudança comportamental ao ser refatorado pelo Netbeans

```
import static java.lang.Math.PI;

public class Circle {
  static double myPI = 3.2;
  static double circle_area(double r) {
    return PI*r*r;
  }
}
```

O NetBeans permite aplicar o refatoramento *Rename Field* para realizar essa transformação. Porém, ao renomear esse atributo, o método `circle_area(double)` passa a chamá-lo, ao invés de chamar a variável estática `PI` importada da biblioteca `java.lang.Math`. Como `A.PI` e `java.lang.Math.PI` possuem valores diferentes (3.2 e aproximadamente 3.14, respectivamente), a classe refatorada não possui o mesmo comportamento da original. Por outro lado, o Eclipse não permite realizar esse refatora-

mento; ele detecta o problema. Transformações que podem ser aplicadas em uma ferramenta, mas não em outra nos mostram que as ferramentas de refatoramentos implementam pre-condições diferentes umas das outras.

A cada ano, os desenvolvedores do Eclipse recebem *bugs* reportados por usuários e removem alguns deles. Por exemplo, de acordo com o Bugzilla[1], foram reportados 80 *bugs* no refatoramento *Move* dessa IDE desde sua criação até o ano de 2009. A Figura 2.5 mostra um gráfico com a quantidade de *bugs* reportados de 2001 a 2009 nos refatoramentos: *Rename*, *Move*, *Extract Method*, *Push Down Method* e *Pull Up*. Os refatoramentos com mais *bugs* reportados são o *Move*, *Rename*, e *Extract Method* com 80, 77, e 39 *bugs*, respectivamente. Enquanto o *Pull Up* e o *Push Down* possuem 15 e 13 *bugs*, respectivamente.

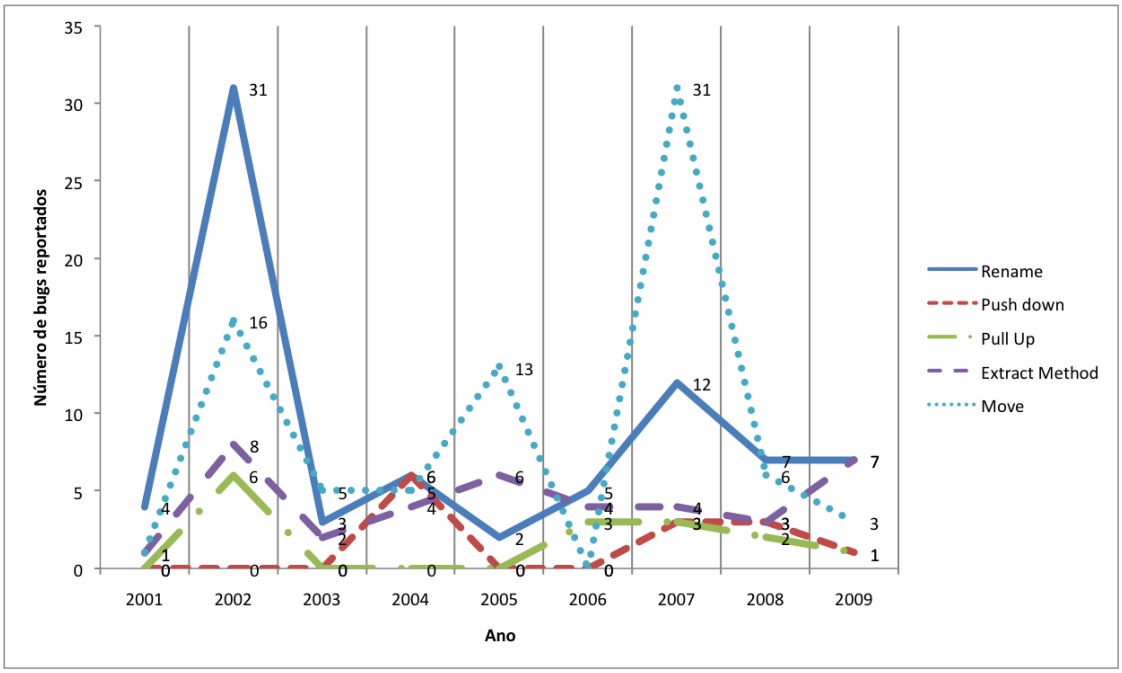

Figura 2.5: Gráfico representando a evolução do bugs reportados na ferramenta de refatoramento do Eclipse no período de 2003 à 2009

Daniel et al. [14] propuseram uma técnica para testar ferramentas refatoramentos de forma automatizada. O principal elemento da técnica é o gerador de programas Java ASTGen (descrito na Seção 2.4). Eles utilizam os programas gerados como entrada para a ferramenta de refatoramento. Para detectar refatoramentos incorretos, eles analisam sintaticamente o

[1]Bugzilla é o bug report do Eclipse. Pode ser acessado em: https://bugs.eclipse.org/bugs/

programa antes e depois da transformação. Como resultado, eles encontraram 21 *bugs* no Eclipse e 24 *bugs* no Netbeans. A maioria dos *bugs* encontrados foi relacionada à introdução de erros de compilação.

Utilizar ferramentas de refatoramentos torna a atividade mais segura, mas não acaba com os riscos de mudanças comportamentais. Fowler [21] afirma que a pre-condição essencial para refatorar é possuir uma coleção de testes confiáveis. Por isso, além das ferramentas de refatoramentos, os *frameworks* para testes de unidade, como o JUnit [46], são essenciais na atividade de refatoramento. Os desenvolvedores aplicam os refatoramentos utilizando ferramentas, como o Eclipse. Se não ocorrer nenhum problema, eles executam os testes para aumentar a confiança de que o refatoramento está correto. Caso algum teste falhe, eles desfazem o refatoramento utilizando a funcionalidade de *undo* das ferramentas.

Porém, alguns refatoramentos invalidam os testes mesmo sem alterar o comportamento do programa [47]. Isso acontece porque alguns testes dependem do código do programa, por isso, freqüentemente eles precisam ser refatorados para se adequarem ao novo código. Por exemplo, considere a classe `A` e sua subclasse `B` exibida no Código Fonte 2.9, e o teste de unidade para o método `k()` dessa classe (Código Fonte 2.10).

Código Fonte 2.9: Código que tem um teste invalidado ao aplicar o refatoramento Push Down Method

```
public class A {
  public int k() {
    return 10;
  }
}
class B extends A {
}
```

Código Fonte 2.10: Teste que não compila após o refatoramento Push Down Method

```
public void testK() {
  A a = new A();
  assertEquals(10, a.k());
}
```

Suponha que desejamos aplicar o refatoramento *Push Down* ao método `A.k()`. Se utilizarmos o Eclipse 3.4 para isso, a ferramenta move o método para a classe `B` e não faz nenhuma mudança no teste. Após esse refatoramento, nosso teste `testK()` não compila, pois ele faz uma chamada a `k` em `A`. O Eclipse modifica automaticamente os testes de unidade em alguns refatoramentos, como o *Rename Method*. Porém, da mesma forma que as ferramentas podem introduzir erros ao refatorar um programa, elas também podem introduzir erros na coleção de testes, fazendo com que a mudança comportamental no programa não seja detectada. Portanto, a prática atual de aplicar refatoramentos utilizando ferramentas para automatizá-los e testes não é suficiente para garantir a preservação do comportamento do programa.

2.2.5 Estado da Arte

Nessa seção, apresentamos alguns trabalhos relacionados a refatoramentos. Como descrito na Seção 2.2.4, mesmo refatoramentos aparentemente simples como o *Rename* são aplicados de forma incorreta por ferramentas de refatoramentos. O ideal seria que as pre-condições de cada refatoramento fossem especificadas formalmente de acordo com a semântica formal da linguagem.

Provar formalmente refatoramentos de programas foi proposto como um desafio [62]. Alguns trabalhos contribuíram nesta direção. Gheyi [26] formalizou a semântica, sistema de tipos e regras de boa formação de Alloy [36], uma linguagem formal de modelagem orientada a objetos. Além disso, ele propôs uma noção de equivalência [28] e codificou no PVS [56]. Baseado nestas formalizações, ele estabeleceu formalmente as pre-condições e provou em um provador de teoremas que vários refatoramentos propostos para Alloy são corretos [27].

Com relação a Java, a linguagem possui uma descrição completa de todos os seus aspectos [31]. Porém, ela não possui uma semântica formalmente definida. Borba et al. [4; 5; 13] propuseram um conjunto de refatoramentos para a linguagem Refinement Object-Oriented Language (ROOL), um subconjunto de Java seqüencial e com semântica de cópia. Estes refatoramentos são provados corretos com relação a uma semântica formal. Para exemplificar, mostramos um refatoramento proposto por eles. A regra a seguir formaliza o refatoramento *Pull Up Field* quando é aplicada da esquerda para direita e *Push Down Field* quando é da direita para a esquerda (Figura 2.6). Cada refatoramento consiste de dois *tem-*

plates de programas ROOL; um do lado esquerdo e outro do lado direito. O refatoramento pode ser aplicado se os programas casarem com os *templates*, ou seja, se todas as variáveis dos *templates* forem atribuídas para valores concretos. Cada refatoramento pode conter meta-variáveis. Por exemplo, cds, ads, e ops são meta-variáveis que definem um conjuto de classes, atributos e operações, respectivamente. Já a meta-variável c representa a função *main*. A noção de equivalência deles é baseada na comparação da função *main* das duas versões do programa, similar à noção de equivalência proposta por Opdyke [54]. A seta ($\rightarrow$) antes da condição indica que ela é requerida quando a regra é aplicada da esquerda para a direita. Da mesma forma, a seta ($\leftarrow$) indica uma condição na direção inversa. Adicionalmente, a seta ($\leftrightarrow$) indica uma condição necessária para ambas as direções. Nesse exemplo, vemos que para mover um atributo para a classe pai, não pode existir um atributo com o mesmo nome nela.

Figura 2.6: Regra de refatoramento para ROOL

ROOL Refactoring $\langle$*move atributo para superclasse*$\rangle$

class B **extends** A ads ops **end** **class** C **extends** B **pub** $a : T; ads'$ ops' **end**	$=_{cds,c}$	**class** B **extends** A **pub** $a : T; ads$ ops **end** **class** C **extends** B ads' ops' **end**

restrições

($\rightarrow$) O atributo de nome a não é declarado pelas subclasses de B em cds;

($\leftarrow$) $D.a$, para qualquer $D \leq B$ e $D \not\leq C$, não aparece em cds, c, ops, or ops'.

Silva et al. [64] adaptaram as leis propostas por Borba et al. [4; 5; 13], e propuseram novas leis para linguagens sequenciais orientadas a objetos levando em consideração semântica de referência. Eles provaram a corretude de cada uma das leis com relação à semântica de um linguagem que possui semântica de referência chamada rCOS. Porém, algumas fun-

cionalidades de linguagens como Java e C# não foram consideradas, como sobrecarga de método e *field hiding*. Além da dificuldade de formalizar e provar refatoramentos em linguagens complexas como Java, a introdução de novas construções a partir da evolução dessas linguagens obriga que todas as provas sejam refeitas.

Duarte [17] estende as leis de ROOL para o contexto de Java considerando boa parte dessa linguagem e paralelismo. Ele não prova essas leis com relação a uma semântica formal; por isso, não há garantias que elas não contêm erros. Por exemplo, a *Lei 10* (sentido esquerda para direita) especifica uma transformação que altera a visibilidade de um método de `private` para `public`. Eles não definiram nenhuma restrição para essa transformação. Porém, aumentar a visibilidade de um método pode habilitar a sobrescrita dele e alterar o comportamento do programa [67]. Além disso, ele considera apenas um subconjunto de Java, sem a estrutura de pacotes de Java, e sem algumas funcionalidades como sobrecarga de método. Por isso, as leis propostas por ele podem não ser suficientes para garantir a preservação do comportamento em refatoramentos de programas Java.

Alguns trabalhos propõem a aplicação de refatoramentos utilizando solucionadores de restrições [50]. Eles especificam o refatoramento (pre-condições e transformação) a partir de regras de inferência para derivar restrições, e utilizam um solucionador de restrições para encontrar a solução. A partir da solução encontrada, eles transformam o programa. Fuhrer et al.[73] propõem regras de inferência que garantem a preservação do comportamento com relação restrições de tipo. Steimann e Thies [70] formalizam restrições de visibilidade de Java, e propõem regras de inferência, relacionadas à visibilidade, para aplicar refatoramentos garantindo a preservação do *binding* estático e dinâmico das entidades.

Schäfer et al. [62] propõem abandonar o critério de preservação do comportamento baseado em pre-condições relacionadas com semântica da linguagem. Eles acreditam que é mais realístico provar individualmente cada refatoramento com relação a invariantes de propriedades específicas para o refatoramento. Dessa forma eles decompõem o problema de definir toda a semântica de uma linguagem complexa orientada a objeto, para definir propriedades mais simples. Eles apresentam uma solução para o *Rename* formalizando o *name lookup* de Java para garantir que após o refatoramento, todos os nomes no programa estejam vinculados estaticamente as entidades de antes [61]. Eles também propõem uma solução similar para o refatoramento *Extract Method* [63].

Dig e Johnson [15] analisam refatoramentos no contexto de reúso de software e API. Eles analisaram mudanças em três *frameworks* e uma biblioteca largamente utilizados. Como resultado, perceberam que mais de 80% das mudanças feitas nas API que provocam incompatibilidade com os clientes são refatoramentos. Henkel e Diwan [33] propõem uma técnica e uma ferramenta para permitir a evolução da API com base em refatoramentos. A ferramenta deles permite gravar o refatoramento aplicado na API para atualizar o código do cliente automaticamente baseado nesse refatoramento.

Alguns trabalhos vêm contribuindo para a popularização de refatoramentos na programação orientada a aspectos [39]. Monteiro e Fernandes [49] propõem uma coleção de refatoramentos orientados a aspectos; eles revisam os *bad smells* de orientação objetos no contexto de orientação a aspectos e também sugerem novos *smells* de aspectos. Cole e Borba [9] propõem trinta leis para AspectJ [40], uma linguagem de orientação a aspectos. Cada lei define duas transformações (inversas) de forma similar às leis de ROOL. Eles utilizam essas leis para provar refatoramentos previamente propostos. Rebêlo et al. [58] propõem leis similares às leis de Cole e Borba para otimizar um compilador de JML [42] (Java Modeling Language) utilizando refatoramentos orientados a aspectos.

Alguns trabalhos propõem refatoramentos para programação por contratos [48], uma metodologia que permite desenvolver programas orientados a objetos junto com especificações. A especificação faz parte de código na forma de invariante de classes e pré- e pós-condições de métodos. Os contratos e o código possuem dependências mútuas. De um lado, mudanças no código podem implicar em mudanças na especificação. Por exemplo, quando se renomeia um atributo pode ser preciso alterar a especificação. Por outro lado, os contratos implicam pre-condições para os refatoramentos. Goldstein et al. [30] propõem uma técnica e uma ferramenta para realizar refatoramentos em programas Java + JML. Freitas et al. [22] estendem as leis de ROOL para considerar escritas em JML. Além disso, eles propõem leis para invariantes em JML. Essas leis indicam se um refatoramento pode ser realizado; o código fonte Java precisa continuar em conformidade com a especificação JML após o refatoramento.

Murphy-Hill et al. [53] realizaram alguns experimentos para analisar como os desenvolvedores refatoram o código; eles analisaram históricos e repositórios referentes ao Eclipse. Entre as conclusões, eles afirmam que quase 90% dos refatoramentos são realizados manualmente. Eles propõem *designs* para melhorar a usabilidade das ferramentas de refatoramen-

tos [52].

2.3 Alloy

Nessa seção, mostramos uma visão geral da linguagem de especificação formal Alloy [36]. Ela é baseada em lógica de primeira ordem e tem expressividade suficiente para modelar vários tipos de aplicativos orientados a objetos [36]. Alloy é similar a Object Constraint Language (OCL) [76] e diagramas de classe UML [3], porém, ele possui sintaxe, sistema de tipos, e semântica mais simples, e permite a análise automática dos modelos. Além disso, Alloy é uma linguagem completamente declarativa, enquanto que OCL combina a abordagem declarativa com elementos operacionais. As principais características de Alloy são:

- checagem com escopo finito: a análise do modelo é feita para um escopo (tamanho) finito. Ela é segura (não gera falsos positivos), mas é incompleta, já que só analisa para um determinado escopo. Porém, podemos dizer que ela é completa para o escopo; ela gera todos os dados possíveis para um determinado escopo;

- declarativa: a modelagem declarativa especifica *o que* queremos gerar, enquanto que a imperativa especifica *como* deve ser feita a geração;

- análise automática: os modelos de Alloy podem ser analisados automaticamente. Podemos gerar exemplos do nosso modelo, ou contra-exemplos de asserções especificadas.

Uma especificação em Alloy consiste de uma seqüência de parágrafos de três tipos. Os parágrafos de *assinatura* são utilizados para definição de novos tipos. Já os parágrafos de *restrições* (fatos, predicados, e funções) são usados para declarar restrições e expressões. Por fim, os parágrafos de *análises* (asserções, execuções, e checagens) são utilizados para realizar as análises do modelo. Antes de darmos uma visão geral desses parágrafos, mostramos o exemplo que será utilizado no decorrer da seção.

2.3.1 Exemplo

Com o propósito de explicar as estruturas de Alloy, descrevemos um modelo de subconjunto de Java especificado por Daniel Jackson e distribuído com o Alloy Analyzer[2]. Esse modelo descreve os tipos *classe* e *interface* de Java. Ele não considera tipos primitivos e o tipo de referência *null*. Uma classe pode estender outra e implementar interfaces, além de possuir um conjunto de atributos. Cada atributo declara um tipo. O modelo de objetos [44] ilustrado na Figura 2.7 exibe esse subconjunto de Java especificado.

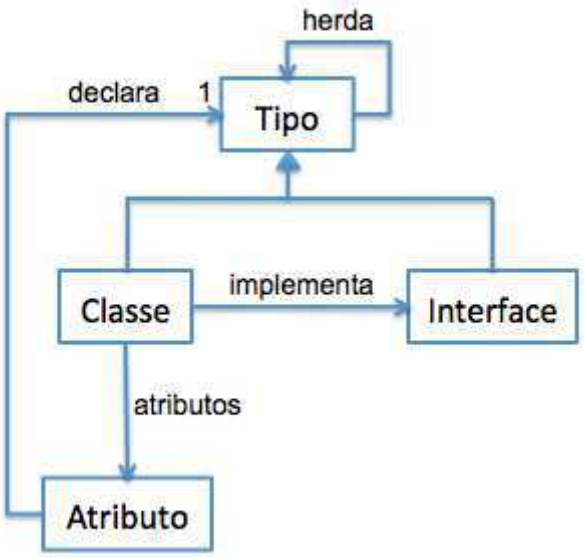

Figura 2.7: Modelo de objetos descrevendo um subconjunto de Java

Cada retângulo representa um conjunto de objetos. As setas representam relações entre dois conjuntos de objetos. Por exemplo, a seta de *Classe* para *Interface* identificada como *implementa* mostra que os objetos de *Classe* possuem uma relação com os objetos de *Interface*. As relações podem conter números em seus extremos indicando a multiplicidade. Quando esses números são omitidos, a multiplicidade é definida como zero ou mais. A seta com a ponta fechada indica uma relação de subconjunto. Nesse modelo, *Interface* e *Classe* são subconjuntos de *Tipo*. Esses subconjuntos compartilham a mesma seta, indicando que

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

são conjuntos disjuntos. A ponta da seta preenchida indica que os objetos desse subconjunto formam todos os objetos de *Tipo*. A seguir, descrevemos as estruturas da linguagem Alloy e mostramos como podemos modelar um meta-modelo similar a esse.

2.3.2 Assinaturas

Uma assinatura introduz um conjunto de objetos, similar aos retângulos do modelo de objetos. Ela pode conter um conjunto de relações desses objetos com outros objetos. As relações são declaradas como atributos da assinatura. O Código Fonte 2.11 exibe a assinatura que representa os objetos de *Tipo* do modelo de objetos da Figura 2.7. O atributo especificado define uma relação. Do lado esquerdo é definido o nome da relação, e do lado direito a expressão. As relações podem conter restrições de multiplicidade definidas pelas palavras-chaves: `set` (qualquer número), `one` (exatamente um), `lone` (zero ou um), e `some` (um ou mais). Quando a multiplicidade é omitida, ela é definida como exatamente um. A relação de nome `herda` descreve a herança entre tipos em Java. A palavra-chave `abstract` define que a assinatura não tem nenhum elemento além dos elementos de seus subconjuntos.

Código Fonte 2.11: Assinatura que especifica o elemento Tipo de Java

```
abstract sig Tipo {
  herda: set Tipo
}
```

Alloy permite que uma assinatura estenda outra, dessa forma a assinatura *estendida* é um subconjunto de outra assinatura, similar a seta com a ponta fechada do modelo de objetos. Uma assinatura estendida herda as relações definidas na assinatura pai. As assinaturas `Interface` e `Classe`, exibidas no Código Fonte 2.12, estendem a assinatura `Tipo`. Elas representam os objetos modelados com esses nomes no modelo de objetos. Na assinatura `Classe` são especificadas as relações `implementa` e `atributos`. Essa última relaciona cada objeto dessa assinatura com objetos de `Atributo`. Na assinatura `Atributo` é especificada uma relação `declara`, relacionando cada objeto dessa assinatura com um objeto de `Tipo`.

Além de declarar Assinaturas e relações, Alloy permite declarar invariantes para assegurar que o comportamento dos objetos modelados ocorrerá da maneira esperada. No nosso

exemplo, as invariantes garantirão que as instâncias do nosso meta-modelo Java estejam sintaticamente corretas. A seguir, mostramos como especificar essas restrições em Alloy.

Código Fonte 2.12: Assinatura que especifica os elementos Classe, Interface, e Atributo de Java

```
sig Interface extends Tipo {}
sig Classe extends Tipo {
  implementa: set Interface,
  atributos: set Atributo
  }
sig Atributo {
  declara: Tipo
  }
```

2.3.3 Fatos e Predicados

Um fato contém um conjunto de restrições que sempre são verdadeiras (invariantes). Cada assinatura pode conter fatos específicos para seus objetos (fatos de assinatura). Eles são declarados logo após a assinatura. Por exemplo, A assinatura `Interface` possui um fato com uma restrição especificando que seus objetos só estendem objetos de `Interface` (Código Fonte 2.13). O operador `in` representa uma relação de subconjunto. A assinatura `Classe` possui um fato com duas restrições. A primeira define a multiplicidade da relação *herda* como zero ou um. A segunda restrição define que uma classe só pode estender outra classe.

Código Fonte 2.13: Fato que especifica que uma classe não pode herdar ela mesma

```
sig Interface extends Tipo {}
  { herda in Interface }
sig Class extends Tipo {
  ...
} {
  lone herda
   herda in Class
}
```

O fato seguinte possui uma restrição especificando que um tipo não pode herdar ele mesmo, direta ou indiretamente (Código Fonte 2.14) . Do lado esquerdo da restrição temos

a variável `t` pertencente a `Tipo` e o quantificador `all` (para todo). Além desse quantificador, Alloy possui: `some` (para algum), `no` (para nenhum), `lone` (para até um), `one` (para exatamente um). Do lado direito temos uma restrição que será verdade para todo `t`. A palavra-chave `!` representa o operador lógico de negação. A Tabela 2.3 exibe outros operadores lógicos de Alloy. A palavra-chave `^` define o fecho transitivo de uma relação binária. Se considerarmos a relação `herda` como um grafo, o fecho transitivo `t.^herda` representa o conjunto de todos os vértices que podem ser atingidos por algum caminho iniciado a partir de `t.herda`.

Código Fonte 2.14: Fato que especifica que uma classe não pode herdar ela mesma

```
fact heranca {
  all t: Tipo | t !in t.^herda
}
```

Operadores Lógicos de Alloy	
not	negação
and	conjunção
or	disjunção
implies	implicação
else	alternativa
iff	bi-implicação

Tabela 2.3: Semântica de alguns operadores lógicos de Alloy

Alloy também possui o parágrafo de predicado (`pred`). Cada predicado contém restrições que são verdadeiras sempre que ele for utilizado. Ele pode ser utilizado para representar operações que descrevem transições de estado, especificando pré- e pós-condições.

Até o momento, explicamos como modelar elementos e relações em Alloy, e especificar restrições para garantir o comportamento correto desses objetos. A seguir, mostramos como é feita a análise dos modelos.

2.3.4 Análise

Alloy possui alguns parágrafos que são usados para realizar análises (asserções, execuções, e checagens) no modelo usando a ferramenta Alloy Analyzer [35]. Essa ferramenta pode

ser usada para achar soluções para uma especificação, ou checar se alguma propriedade do modelo é sempre verdadeira para um determinado escopo. O escopo define o número máximo de objetos para cada assinatura. A seguir, mostramos como utilizar o Alloy Analizer para gerar instâncias do nosso modelo de sistemas de arquivos.

Alloy possui o comando `run`. Este comando ordena que Alloy Analyzer busque todas as soluções que satisfaçam o modelo para um determinado escopo de objetos. Para usar esse comando, definimos um predicado `exibe` sem nenhuma restrição adicional, e o escopo com 3 objetos de `Tipo` e 4 objetos de `Atributo` (Código Fonte 2.15).

Código Fonte 2.15: Commando para executar o Alloy Analyzer e achar soluções para o modelo

```
pred exibe { }
run exibe for 3 Tipo, 4 Atributo
```

O Alloy Analyzer permite visualizar as soluções encontradas através de uma visualização gráfica, ou uma estrutura em árvore. A Figura 2.8 exibe a interface gráfica desse analisador mostrando uma instância gerada. A solução encontrada representa um programa Java com a classe `Classe0` e sua classe filha `Classe1`. A primeira implementa a interface `Interface`, e possui dois atributos (`Atributo0` e `Atributo1`) do tipo `Interface`. A classe `Classe1` declara os mesmos atributos. A partir das soluções encontradas, podemos analisar se é necessário adicionar mais restrições ao modelo, ou se especificamos restrições desnecessárias.

Alloy também possui o parágrafo de asserção (`assert`), nele especificamos algumas afirmações que devem ser verdadeiras segundo o comportamento do modelo. Podemos utilizar o comando `check` para o Alloy Analyzer buscar contra-exemplos em um determinado escopo. Por exemplo, o Código Fonte 2.16 exibe uma asserção para checar se existe alguma classe que herda uma interface. O Alloy Analyzer não achou nenhum contra exemplo para nossa asserção. Podemos aumentar o escopo de objetos para termos mais confiança de que a asserção é sempre válida. Alguns domínios de aplicação possuem números máximos de objetos. Nesses casos, podemos usar o Alloy Analyzer para fazer provas formais da corretude do sistema.

Nesse exemplo, mostramos uma especificação de um subconjunto de Java, e utilizamos o Alloy Analyzer para gerar instâncias representando programas Java a partir desse modelo.

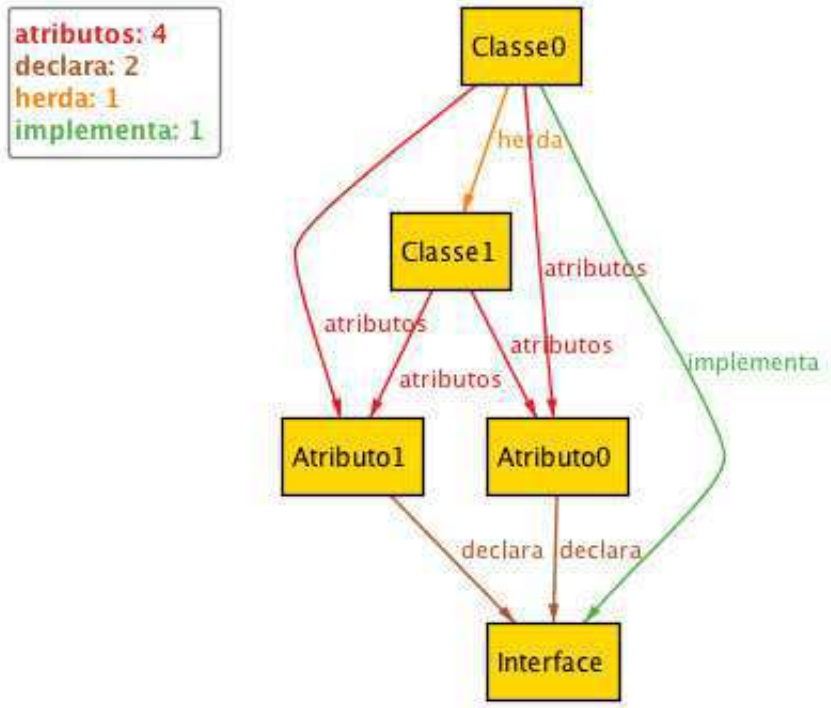

Figura 2.8: Interface gráfica do Alloy Analyzer

No Capítulo 4, mostramos como utilizamos Alloy para desenvolver um gerador de programas Java útil para gerar entradas para testar ferramentas de refatoramento.

Código Fonte 2.16: Commando para executar o Alloy Analyzer e checar asserções no modelo

```
assert Heranca {
  all c : Classe1 no i : Interface1 i in c.herda
}
check Heranca for 5 Tipo, 3 Atributo
```

2.4 ASTGen Framework

Nessa seção descrevemos o ASTGen, um *framework* para geração de árvores sintáticas abstratas (*AST*) de Java [14]. Ele foi utilizado para testar alguns refatoramentos implementados pelo Eclipse e Netbeans.

O ASTGen é um gerador baseado no paradigma de *programação imperativa*; ou seja,

o desenvolvedor precisa programar *como* deve ser feita a geração. Adicionalmente, sua geração segue a abordagem *bounded-exhaustive-testing* [45], ou seja, ela gera todas as combinações possíveis para um determinado escopo de complexidade ou tamanho.

O ASTGen foi desenvolvido em Java. Sua arquitetura foi modelada para permitir a composição de geradores a partir da combinação de geradores menores. Cada gerador é modelado utilizando o padrão de projeto *iterator* [25], ou seja, a geração é feita iterativamente.

Para exemplificar, mostramos como o ASTGen gera atributos de classe. Suponha que desejamos gerar declarações de atributos inteiros ou booleanos, com qualquer visibilidade e sem inicialização. Para isso, o ASTGen fornece o gerador `FieldDeclarationGen` (Código Fonte 2.17).

Código Fonte 2.17: Versão simplificada do gerador de atributos do ASTGen

```
class FieldDeclarationGen extends ASTNodeGenBase<FieldDeclaration> {
  IGenerator<Modifier> modifierGen;
  IGenerator<Type> typeGen;
  IGenerator<Identifier> idGen;

  ... (construtores e demais métodos)

  FieldDeclaration generateCurrent() {
    FieldDeclaration generated = new FieldDeclaration();
    generated.setModifier(modifierGen.current());
    generated.setType(typeGen.current());
    generated.setIdentifier(idGen.current());
    return generated;
  }
}
```

A classe `FieldDeclarationGen` estende `ASTNodeGenBase`, classe base para gerar os nós da AST. Cada nó é representando pelas classes da API Eclipse Core[3]. O nó `FieldDeclaration` possui três nós filhos: `Modifier` (visibilidade), `Type` (tipo que o atributo declara), `Identifier` (nome do atributo). O gerador `FieldDeclarationGen` é composto por três geradores responsáveis por gerar variações desses nós filhos:

[3]Java Model Tutorial: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_model.htm

`modifierGen`, `typeGen`, `idGen`. A cada iteração, o método `generateCurrent()` cria uma declaração de atributo combinando esses três geradores.

Para iniciarmos o gerador `FieldDeclarationGen`, precisamos primeiro instanciar os geradores `modifierGen`, `typeGen`, `idGen`, como é exibido no Código Fonte 2.18.

Código Fonte 2.18: Inicialização dos geradores que compõem o gerador FieldDeclarationGen

```
IGenerator<Modifier> modifierGen = new Chain<Modifier>(public, private,
    protected, default);
IGenerator<Type> typeGen = new Chain<Type>(int, boolean);
IGenerator<Identifier> idGen = new Chain<Identifier>(x);
```

Dessa forma, o atributo declarado poderá ter a visibilidade `public`, `private`, `protected`, ou `default`. Será do tipo `int`, ou `boolean`, e terá o nome `x`. Esses geradores são passados como parâmetros para inicializar o gerador `FieldDeclarationGen` (Código Fonte 2.19). Com esses parâmetros, o gerador produz um total de 8 declarações de atributos.

Código Fonte 2.19: Instanciação do gerador FieldDeclarationGen

```
FieldDeclarationGen fieldDeclGen =
  new FieldDeclarationGen(modifierGen, typeGen, idGen);
```

O ASTGen possui 43 geradores para elementos da sintaxe Java, tais como interfaces, classes, métodos, atributos, expressões de atribuição, estruturas de repetição (*while, for*), estruturas de controle de fluxo (*if, switch*). Entre esses geradores, temos:

- *ClassGenerator*. Gera uma classe. É composto por geradores de visibilidade, identificador, e corpo de classe. Pode gerar classes que implementam interfaces e estendem outras classes;
- *MethodDeclarationGenerator*. Gera um método. É composto por geradores de visibilidade, identificador, corpo de método, parâmetros, retorno;
- *MethodsInvocationGenerator*. Gera uma expressão de chamada de método. É composto por um gerador de declaração de método e por um gerador de tipo de chamada de método. Ele pode gerar chamadas simples ou utilizando `super`, `this`, e `new`;

- *ReturnStatementGenerator*. Gera uma instrução de retorno. É composto por um gerador de expressões. Ele gera uma instrução de retorno para cada expressão gerada;

- *IfStatementGenerator*. Gera as instruções de repetição `if` e `then`. É composto por dois geradores de instrução (cada um gera uma instrução para um dos fluxos) e um gerador de expressão responsável por gerar uma expressão booleana.

Além desses gerados, o ASTGen possui 8 geradores para construir expressões aninhadas e 3 gerados para gerar relações entre classes (herança e classes aninhadas).

A partir desses geradores de AST, Daniel el al. [14] desenvolveram 13 geradores de programas Java. Eles utilizaram esses geradores para gerar entradas para ferramentas de refatoramentos e testá-las. Como resultado, eles identificaram 21 *bugs* no Eclipse e 24 no NetBeans. Apesar disso, o esforço para escrever geradores para diferentes tipos de refatoramentos utilizando esse *framework* é alto, pois os testadores precisam programar como será feita a geração do programa. No Capítulo 4, mostramos como utilizamos o Alloy e o ASTGen para combinar a programação declarativa com a imperativa com o objetivo de diminuir o esforço necessário para gerar programas variados.

Capítulo 3

Uma Técnica para Aumentar a Segurança em Refatoramentos

Como vimos, mesmo as melhores ferramentas de refatoramentos realizam transformações que alteram o significado de um programa. Neste capítulo, descrevemos nossa técnica para detectar mudanças comportamentais em refatoramentos de programas. Ela pode ser aplicada em programas com as seguintes características:

- orientação a objetos;
- seqüenciais (sem concorrência);
- determinísticos.

Nossa técnica gera uma coleção de testes capaz de ser executada sem alteração no programa original e refatorado. O comportamento observável do programa antes e depois da transformação é exercitado com esses testes. Se detectarmos alguma mudança comportamental, a transformação não pode ser considerada um refatoramento.

Primeiramente, mostramos uma visão geral da nossa técnica (Seção 3.1). Nas Seções 3.2 a 3.6, descrevemos em detalhes cada etapa da técnica. Em seguida, exibimos um exemplo de como utilizá-la(Seção 3.7). Por fim, descrevemos nossa definição de noção de equivalência entre programas (Seção 3.8).

3.1 Visão Geral

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

A seguir, descrevemos o processo de nossa técnica para detectar mudanças comportamentais. Ele é dividido em cinco etapas sequenciais, ilustradas na Figura 3.1. Como pode ser observado, ele recebe como entrada as duas versões do programa (original e refatorada), e reporta quando é seguro aplicar o refatoramento. Na Etapa 1, uma análise estática identifica os *métodos públicos comuns* às versões original e refatorada do programa. Consideramos métodos comuns aqueles que possuem a mesma assinatura nas duas versões do programa (construtores também são considerados nessa etapa). Em seguida (Etapa 2), utilizamos o Randoop, um gerador automático de testes de unidade, para criar uma coleção de testes para os métodos identificados na Etapa 1. Os testes gerados podem ser executados nas duas versões do programa, pois eles exercitam a interface externa, comum às duas versões. Nas Etapas 3 e 4, os testes gerados são executados no programa original e refatorado, respectivamente. Ao final, a Etapa 5 analisa os resultados dos testes: se um teste passar no programa original e falhar no refatorado, é detectada uma mudança comportamental. Caso contrário, o programador tem mais confiança de que a transformação não altera o comportamento do programa.

Nossa abordagem é complementar à prática atual da utilização de testes de unidade para avaliar a corretude dos refatoramentos, pois o desenvolvedor pode continuar utilizando seus testes para aumentar a confiança na corretude da transformação. Ela pode ser utilizada para checar a segurança de um refatoramento que será realizado por uma ferramenta de refatoramento. Para realizar essa função, desenvolvemos o SAFEREFACTOR (Capítulo 6). Ele fornece um plugin para o Eclipse que permite ao usuário checar a segurança dos refatoramentos implementados por essa IDE. Adicionalmente, o SAFEREFACTOR fornece uma interface de linha de comando (Seção 6.2) que permite a aplicação de nossa técnica para refatoramentos aplicados manualmente.

A seguir, detalhamos cada etapa da técnica. Ao final, mostramos um exemplo prático de sua utilização para detectar uma mudança comportamental.

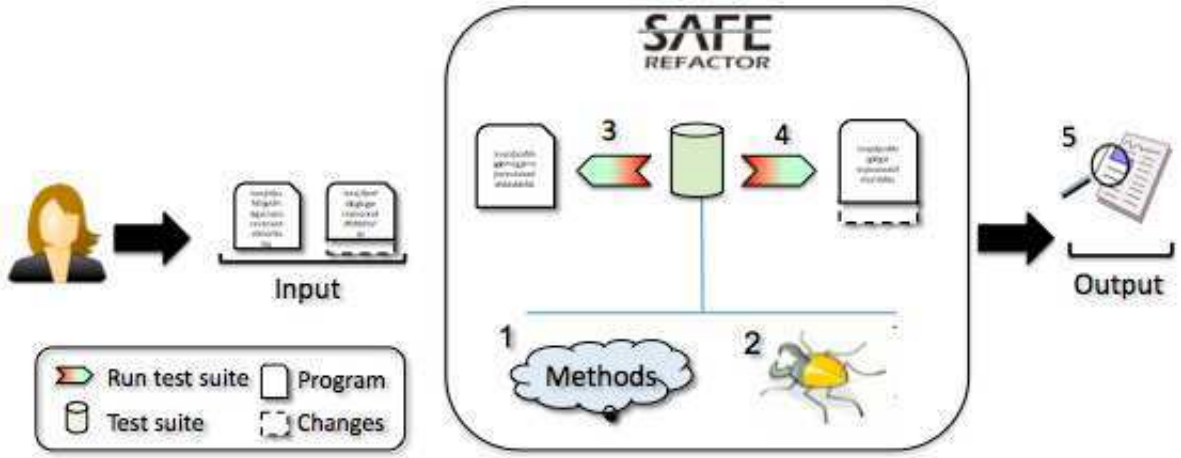

Figura 3.1: Técnica para detectar mudanças comportamentais; 1. Análise estática identifica os métodos públicos comuns, 2. Randoop modificado gera testes de unidade para os métodos identificados, 3. Os testes são executados no programa original, 4. Os testes são executados no programa refatorado, 5. a técnica analisa os resultados: se um teste passar no programa original e falhar no refatorado, ela detecta uma mudança comportamental.

3.2 Etapa 1: Análise Estática

Nossa abordagem tem o diferencial de gerar uma coleção de testes que pode ser executada tanto no programa original quanto refatorado sem que haja a necessidade de refatorá-la também. Para que isso ocorra, na Etapa 1, realizamos uma análise estática nas duas versões do programa (original e refatorada) com o objetivo de identificar os métodos públicos comuns. Nossa análise estática identifica os métodos tanto nas classes em que são declarados quanto nas subclasses que os herdam.

Os métodos identificados nessa etapa são os alvos dos testes que serão criados na Etapa 2. Note que apesar da nossa análise estática não incluir os métodos com assinaturas alteradas pelo refatoramento, o comportamento destes podem ser exercitados através dos métodos públicos comuns.

3.3 Etapa 2: Geração dos Testes de Unidade

Depois de identificar os métodos alvos dos testes, ocorre a segunda etapa, na qual a coleção de testes de unidade é gerada. Para essa tarefa, utilizamos o Randoop; ele é um gerador aleatório automático de testes de unidade que não necessita de *entradas* fornecidas pelo usuário. Esse gerador descobriu erros em aplicações *open-source* e comerciais largamente utilizadas, como a Sun JDK 1.5 [57]. A seguir, mostramos uma visão geral desse gerador.

Randoop

O Randoop utiliza uma geração aleatória guiada pelo *feedback* do comportamento do programa. Ele gera incrementalmente testes de unidade para um conjunto de classes passado como parâmetro. Cada teste consiste de uma seqüência de chamadas de métodos e construtores que criam e alteram objetos, seguido por `assertions` do JUnit.

O gerador seleciona aleatoriamente as chamadas de métodos e construtores para criar cada expressão que compõe o teste. Para compor os argumentos dos métodos, ele utiliza objetos de expressões anteriormente geradas, ou do tipo `String` e variáveis primitivas (`int, char, boolean`).

Logo após criar uma expressão, o Randoop a executa e recebe o *feedback* do comportamento do programa. Ele checa esse resultado com relação alguns contratos e filtros. Os filtros são utilizados para identificar quando uma expressão é ilegal, redundante, ou útil para gerar mais entradas. Por outro lado, os contratos estabelecem invariantes do programa, que ele utiliza para criar automaticamente os JUnit `assertions`. Por exemplo, o Randoop possui um contrato que especifica que um objeto tem que ser igual a ele mesmo. Então, ao capturar o valor retornado por um método de um objeto que está sendo testado, ele gera uma expressão `AssertEquals` que recebe como parâmetro o método executado e o valor retornado. Novos contratos podem ser criados implementando interfaces fornecidas pela API dele.

A seguir mostramos como o Randoop detectou um erro na API de Collection da JDK 1.5. O Código Fonte 3.1 exibe o teste gerado por ele que encontrou o erro. O teste instancia e altera um objeto do tipo `LinkedList`. Depois ele envolve esse objeto em outro do tipo `TreeSet`, e em seguida, exercita o método `Collections.unmodifiableSet`, que

retorna uma coleção que não pode ser modificada, passando esse objeto como parâmetro. O teste termina com uma `assertion` do JUnit, que exercita o método `equals`. Essa asserção falha, o que caracteriza um `bug` detectado (o método `equals` deveria ter retornado `true`).

Código Fonte 3.1: Teste que identificou um erro na API Collection da JDK 1.5.

```
public static void test1() {
  LinkedList l1 = new LinkedList();
  Object o1 = new Object();
  l1.addFirst(o1);
  TreeSet t1 = new TreeSet(l1);
  Set s1 = Collections.unmodifiableSet(t1);
  Assert.assertTrue(s1.equals(s1)); // falha
}
```

O tempo padrão para a geração de testes do randoop é de 120 segundos. A partir de um determinado instante, o Randoop pode começar a gerar testes redundantes. O usuário pode alterar o tempo para geração de testes via parâmetro.

Evolução do Randoop

Nós realizamos alterações no Randoop para possibilitar a geração de testes apenas para um conjunto de métodos específicos, pois nosso objetivo é gerar testes somente para os *métodos públicos comuns*. Adicionalmente, corrigimos alguns *bugs* que geravam testes com problemas de compilação.

O algoritmo original do Randoop possui quatro parâmetros: lista de classes alvos do teste, lista de contratos, lista de filtros, e o tempo para a geração de testes. Modificamos o Randoop para considerar parâmetros adicionais. Na versão original, ele gera testes para todos os métodos da lista de classes passada como parâmetro. Na versão modificada, é passada para ele uma lista contendo os métodos que devem ser testados. Para permitir isso, alteramos a classe `randoop.util.DefaultReflectionFilter`. Ela contém o método `canUse(Method m)`, este indica quando um método pode ser usado para a criação dos testes. Além dos filtros padrões, nós incluímos uma checagem adicional que verifica se o método está contido no conjunto de métodos passado como parâmetro.

Utilização do Randoop

Para compor a nossa técnica, utilizamos a versão do Randoop modificada. Como parâmetros, utilizamos o conjunto contendo as classes que possuem os métodos identificados na Etapa 1, e a lista desses métodos. Dessa forma, o Randoop gera uma coleção de testes especifica para o refatoramento em questão, por isso, conseguimos rodá-la no programa original e refatorado. Além disso, utilizamos os filtros e contratos padrões dele.

3.4 Etapa 3: Execução dos Testes no Programa Original

Nesta etapa, nós usamos o JUnit framework para *executar a coleção de testes gerada* na versão original do programa (Etapa 3). O objetivo é remover os testes que não são úteis para detectar mudanças comportamentais realizadas pelo refatoramento.

Eventualmente, um teste gerado pelo Randoop pode não passar no programa original. Isso pode demonstrar um *bug* no programa detectado por alguma invariante definida nos contratos do Randoop (como o exemplo mostrado na Seção 3.3), mas que não possui relação com a transformação que está sendo avaliada. Outro possível motivo é se programa possuir *concorrência*, o que está fora do escopo da nossa técnica. Portando, quando um teste não passa no programa original, nós não consideramos seu resultado para a avaliação (Etapa 5).

3.5 Etapa 4: Execução dos Testes no Programa Refatorado

Após executar os testes no programa original, executamos estes no programa refatorado. Como foi dito, os testes gerados por nossa técnica não precisam ser refatorados também. Dessa forma, conseguimos comparar o comportamento do programa com relação à mesma coleção de testes. Portanto, temos mais confiança no resultados de nossos testes.

3.6 Etapa 5: Avaliação dos Resultados dos Testes

A Etapa 5 analisa os resultados dos testes realizados nas Etapas 3 e 4. Se os testes passarem nas duas versões, aumenta a confiança dos desenvolvedores de que a transformação realizada no programa não alterou o comportamento dele. Por outro lado, se algum teste

passar no programa original e falhar no refatorado, é detectada uma mudança comportamental no programa depois da transformação. Portanto, ela não pode ser considerada um refatoramento.

É importante mencionarmos que nossa técnica não prova que o refatoramento está correto, mas aumenta a confiança de que nenhuma mudança comportamental foi introduzida. A seguir, demonstramos a utilização da nossa técnica através de um exemplo prático.

3.7 Exemplo de Aplicação da Técnica

Nesta seção, mostramos na prática como nossa técnica detecta uma mudança comportamental. O exemplo a seguir mostra uma transformação aplicada por algumas ferramentas de refatoramento (Eclipse, NetBeans, JBuilder, IntelligJ), mas que não preserva o comportamento do programa. Essa transformação foi manualmente catalogada por Ekman et al. [19].

O programa original é exibido no Código Fonte 3.2. Ele possui três classes `A`, `B` e `C`. Se utilizarmos uma das ferramentas citadas acima para aplicar o refatoramento *Push Down* ao método `m`, o programa refatorado será similar ao mostrado no Código Fonte 3.3. Essa transformação não preserva o comportamento, pois o valor retornado pelo método `test` foi alterado de `23` para `42`. O método `m` possui uma expressão de invocação de método utilizando o `super` para acessar a classe pai. A mudança comportamental ocorre porque o `super` acessa `A`, classe pai da classe em que ele foi declarado [31], porém, ao mover o método de `B` para `C`, ele passa a invocar `B.k()` ao invés de `A.k()`.

Para detectar essa mudança comportamental, nossa técnica primeiramente analisa a transformação realizada para identificar os *métodos públicos comuns* às duas versões do programa (Etapa 1). Ela identifica os seguintes métodos em comum às duas versões dos programas: `A.k()`, `B.k()`, e `C.teste()`. Esses métodos representam a interface externa do programa, utilizada para compararmos o comportamento dele. O método `B.m()` mudou de classe após a transformação. Como ele não possui o mesmo *qualified name* nas duas versões, ele não é incluído. Não podemos incluí-lo para a geração de testes porque o gerador poderá gerar um teste que não compila em uma das versões.

A Etapa 2 consiste na geração dos métodos identificados na etapa anterior. Com um tempo limite de 2 segundos, o RANDOOP gera uma coleção com 588 testes de unidade. Em

Figura 3.2: Tratamento incorreto de invocação de superclasse

Código Fonte 3.2: Versão Original

```
public class A {
  public int k() {
    return 23;
  }
}
public class B extends A {
  public int k() {
    return 42;
  }
  public int m() {
    return super.k();
  }
}
public class C extends B {
  public int test() {
    return m();
  }
}
```

Código Fonte 3.3: Versão Refatorada

```
public class A {
  public int k() {
    return 23;
  }
}
public class B extends A {
  public int k() {
    return 42;
  }
}
public class C extends B {
  public int m() {
    return super.k();
  }
  public int test() {
    return m();
  }
}
```

seguida, executamos os testes no programa original (Etapa 3); todos os testes passam nessa etapa. Porém, ao executar os testes na versão refatorada (Etapa 4), 554 testes falham. A técnica analisa os resultados dos testes e detecta uma mudança comportamental (Etapa 5), pois 554 testes que foram executados com sucesso no programa original, falharam depois da transformação. O Código Fonte 3.4 mostra um teste de unidade gerado que revela a mudança comportamental.

O Randoop consegue gerar testes para um método considerando as classes que herdam esse método. Por exemplo, o método `B.k()` é declarado na classe `B`, mas, o Randoop gera testes para ele na classe `C` também (Código fonte 3.5). Isso é importante porque alguns refatoramentos podem habilitar ou desabilitar a *sobrecarga* e *sobrescrita*. Ao testar o método em toda hierarquia, nossa técnica consegue capturar problemas relacionados a isso como será mostrado na Seção 5.3.

Alguns refatoramentos, tais como *Rename Field* e *Rename Variable*, não influenciam nossa análise estática. Neles, o conjunto de *métodos públicos comuns* é igual ao conjunto total dos métodos públicos do programa original e refatorado. Porém, outros refatoramentos, tais como *Rename Method* e *Push Down Method*, mudam classes ou métodos; isto diminui o conjunto dos métodos identificados na Etapa 3. Por exemplo, o método `B.m()` não é incluído porque um teste para ele compila no programa original, mas não no refatorado. Sendo assim este teste não é útil para identificarmos mudanças comportamentais. Entretanto, se a alteração desse método alterar o comportamento do programa através de sua interface externa (métodos públicos comuns), nós identificamos a mudança, como ocorre no exemplo da Figura 3.2.

Código Fonte 3.4: Teste de unidade que detecta erro no refatoramento

```
public void test0() {
    C var0 = new C();
    int var1 = var0.k();
    int var2 = var0.k();
    int var3 = var0.test();

    assertTrue(var1 == 42);
    assertTrue(var2 == 42);
    assertTrue(var3 == 23);
}
```

Código Fonte 3.5: Testes de unidade que detecta erro no refatoramento

```
public void test1() {
    C var0 = new C();
    int var1 = var0.k();
    assertTrue(var1 == 42);
}
```

3.8 Noção de Equivalência

As versões original e refatorada de um programa devem possuir o mesmo comportamento observável segundo uma noção de equivalência. Opdyke [54] define sua noção de equiva-

lência da seguinte forma:

> Seja a função *main* a interface externa do programa. Se a função *main* é chamada duas vezes (antes e depois do refatoramento) com o mesmo conjunto de entradas, a saída resultante deve ser a mesma. (p.40)

Nossa noção de equivalência é baseada na de Opdyke. Esta noção pode ser vista como uma aplicação da noção de refinamento de dados [32; 34], também utilizada em outros trabalhos da área [4]. Nossa técnica gera uma coleção de testes de unidade baseada nos métodos públicos comuns a dois conjuntos de classes (referentes aos programas originais e refatorados). Dessa forma, os testes gerados por ela conseguem ser executados nas classes do programa original ou refatorado, sem a necessidade de alterações. Cada teste possui uma seqüência de chamadas que exercitam esses métodos com diferentes entradas. Se o mesmo teste for executado nos dois conjuntos de classes (original e refatorado) e produzir resultados diferentes (por exemplo: sucesso e falha, sucesso e erro), caracterizamos uma mudança comportamental. Podemos dizer que nossa noção de equivalência compara dois conjuntos de classes com relação a `n` métodos `main` diferentes gerados aleatoriamente, onde `n` é o número de casos de teste gerados. Ou seja, nossa noção de equivalência é mais forte do que Opdyke pois nós comparamos as classes com relação a várias seqüências de chamadas de métodos (cada teste), e não só uma seqüência como o Opdyke (checa com relação a 1 main).

Capítulo 4

JDolly: Gerador de Programas Java

Neste capítulo, apresentamos o JDolly, um gerador de programas que permite gerar programas Java com certas características estruturais. Os programas gerados pelo JDolly podem ser utilizados como entradas no processo de testes de sistemas que manipulam programas, tais como: compiladores, interpretadores, *model checkers*, e ferramentas de refatoramento. Nosso objetivo é utilizá-lo para gerar entradas para ferramentas de refatoramento, e utilizar o SAFEREFACTOR para verificar a ocorrência de mudanças comportamentais em cada uma das transformações aplicadas. O JDolly recebe como entrada o número máximo de classes, atributos, e métodos que os programas devem possuir. Adicionalmente, caso o usuário deseje, pode especificar de maneira declarativa características estruturais para os programas. O JDolly gera automaticamente um conjunto de programas.

Primeiro, mostramos uma visão geral do JDolly (Seção 4.1). Em seguida, descrevemos o processo de geração dos programas (Seção 4.2 e Seção 4.3). Na Seção 4.4, mostramos um exemplo de como utilizá-lo. Por fim, discutimos suas vantagens e desvantagens e o comparamos com o ASTGen (Seção 4.5).

4.1 Visão Geral

O JDolly é capaz de gerar programas com classes contendo atributos e métodos e certas características estruturais como herança, sobrecarga e sobrescrita de método. Por exemplo, o Código Fonte 4.10 exibe um programa gerado pelo JDolly. O programa contém a classe `A` e sua classe filha `B`. `A` declara um atributo `x` que é inicializado com o valor 10, e um

método `k(int)` que retorna o valor de x. Em `B`, o método `k(int)` é sobrescrito e retornar o valor 20; além disso, é declarado o método `m()` que possui uma chamada a `k(int)` em sua instrução de retorno.

Código Fonte 4.1: Exemplo de um programa Java gerado pelo JDolly

```
public class A {
  int x = 10;
  public int k(int i) {
    return x;
  }
}

public class B extends A {
  public int k(int i) {
    return 20;
  }
  public int m() {
    return k(0);
  }
}
```

A geração de programas do JDolly é dividida em duas etapas: *geração da parte estrutural* e *geração da parte comportamental*. Na primeira etapa, ele gera a parte estrutural dos programas. Ela está relacionada com a declaração de classes, métodos e atributos. Por exemplo, no programa exibido no Código Fonte 4.10, ela gera as declarações das classes `A` e `B`, dos métodos `A.k(int)` e `B.k(int)`, e do atributo `x`. Essa etapa é baseada no paradigma declarativo de programação, e utiliza a linguagem de especificação formal Alloy [36] e seu analisador Alloy Analyzer [35]. O desenvolvedor especifica o escopo da geração (número máximo de classes, atributos, e métodos), e, se desejar, especifica em Alloy restrições que descrevem *qual* estrutura os programas gerados devem possuir. Na segunda etapa, o JDolly gera a parte comportamental dos programas. Ela está relacionada com a geração das instruções de inicialização de atributos e instruções existentes nos corpos dos métodos. No nosso exemplo (Código Fonte 4.10), ela gera a instrução de atribuição `x = 10`, e as instruções de retorno de cada método: `return x`, `return 20` e `return k(0)`. Essa etapa é baseada no paradigma imperativo de programação e é realizada usando o *framework* para

geração de ASTs de Java ASTGen [14]. Por utilizar esses dois paradigmas de programação (declarativo e imperativo), definimos a geração do JDolly como híbrida. As principais características do JDolly são:

- declarativo: o desenvolvedor especifica o número máximo de classes, atributos, e métodos dos programas gerados. Adicionalmente ele pode especificar características estruturais desejadas para os programas;
- imperativo: o JDolly programa como será realizada a geração da parte comportamental dos programas usando o ASTGen. Adicionalmente o desenvolvedor pode refinar a parte comportamental dos programas estendendo os geradores do ASTGen;
- *Bounded-exhaustive*: ele gera todos os programas para um determinado escopo [12]. O escopo da geração da parte estrutural é o conjunto de todas as soluções em Alloy que satisfazem nosso meta-modelo (descrito na Seção 4.2) Java para um número máximo de classes, atributos, e métodos. Na parte comportamental, o escopo é determinado pelos geradores utilizados para gerar o comportamento dos atributos e métodos do programa. O escopo do JDolly é definido por todas as combinações das gerações das partes estruturais e comportamentais;
- iterativo: a geração dos programas é feita iterativamente, um programa de cada vez. Dessa forma, ele consegue escalar com relação à quantidade de programas gerados, pois não é necessário um tempo para gerar todos os programas para depois poder utilizá-los, nem é necessário espaço para armazenar todos eles.

Os programas gerados pelo JDolly podem conter as entidades: classes, atributos e métodos. As classes podem herdar outras classes declaradas no programa. Os atributos podem ser de tipos primitivos, `String`, ou objetos de classes declaradas no programa. Caso um atributo seja primitivo ou `String`, ele pode ser inicializado com um valor aleatório; caso contrário, ele não é inicializado. Os métodos podem retornar os mesmos tipos dos atributos. A instrução de retorno pode conter um valor aleatório ou uma chamada a atributo ou método. Eles também podem possuir um parâmetro de um desses tipos. Atributos e métodos podem ter visibilidade `public`, `protected`, `private`, ou `package`. Entre as características que os programas gerados podem ter, estão: sobrecarga e sobrescrita de método; atributos

declarados com o mesmo nome na mesma hierarquia de classes; expressões de chamada de método e atributo usando o `super`, `this`, `new`. A seguir, descrevemos a geração da parte estrutural dos programas.

4.2 Geração da Parte Estrutural

A primeira etapa da geração de programas do JDolly consiste em gerar a parte estrutural do programa. Ela é realizada usando o Alloy [36] e o Alloy Analyzer [35]. Especificamos um meta-modelo da linguagem Java em Alloy, e utilizamos o Alloy Analyzer para gerar soluções para esse modelo. Essas soluções são transformadas em código Java. A seguir, descrevemos esse processo.

Meta-modelo de Java

Primeiramente, especificamos em Alloy um meta-modelo simplificado de um subconjunto de Java. Ele contém construções de Java relacionadas com a parte estrutural do programa. Modelamos os tipos em Java (primitivos, `String`, e classe), herança, membros de classe (atributo e método), e visibilidade (pública, protegida, privada, pacote). A partir desse meta-modelo, podemos gerar programas com as seguintes características estruturais:

- herança;
- sobrecarga de método;
- sobrescrita de método;
- re-declaração de atributo pela classe filha.

As assinaturas exibidas no Código Fonte 4.2 especificam os tipos primitivos `int`, `long`, `boolean`, o tipo `String`. A palavra-chave `one` define que a assinatura possui exatamente um elemento. Nesse caso, ela introduz uma enumeração dos tipos.

Código Fonte 4.2: Especificação de Tipos em Java

```
abstract sig Type {}
one sig Int,Long,Boolean extends Type {}
one sig String extends Type {}
```

A assinatura `Class` modela uma classe de Java (Código Fonte 4.3). A relação `extend` no corpo de `Class` especifica que uma classe pode estender no máximo uma classe, e as relações `fields` e `methods` declaram que cada classe contém um conjunto de atributos (`Field`) e métodos (`Method`). Em Java, um atributo possui um identificador, uma visibilidade e um tipo; enquanto que os métodos, além de possuir identificador e visibilidade, possuem um tipo de retorno e podem conter uma lista argumentos. Para simplificar, modelamos os métodos permitindo no máximo um parâmetro. O Código Fonte 4.3 exibe as assinaturas `Field` e `Method` referentes aos atributos e métodos de Java, respectivamente. As assinaturas `Id` e `Visibility` representam o identificador e a visibilidade dos elementos; sendo que a visibilidade pode ser `public`, `private`, `protected`, `package`.

Código Fonte 4.3: Especificação de classe de Java

```
sig Class extends Type {
  extend: lone Class,
  fields: set Field,
  methods: set Method
}

sig Id {}
abstract sig Visibility {}

one sig public, private, protected, package extends Visibility {}
sig Field {
  id : one Id,
  type: one Type,
  vis : one Visibility
}
sig Method {
  id : one Id,
  vis: one Visibility,
  return: one Type,
  arguments: lone Type
}
```

O JDolly utiliza o Alloy Analyzer para gerar as estruturas dos programas. Este analisador gera todas as soluções para o modelo em um determinado escopo de assinaturas. Ele

transforma as soluções em código Java. Para exemplificar, suponha que o usuário deseja gerar programas com no máximo duas classes, um atributo, e três métodos. Passamos para o JDolly o escopo de 2, 1, e 3 para `Class`, `Field` e `Method`, respectivamente. O Código Fonte 4.4 exibe a parte estrutural de um programa gerado pelo JDolly com essas características. Ela possui a classe `A` e sua classe filha `B`, um atributo `A.x`, e os métodos `A.k(int)`, `B.k(int)`, e `B.m()`.

Código Fonte 4.4: Parte estrutural de um programa gerado pelo JDolly

```
public class A {
  public int x;
  public int k(int i) {
  }
}
public class B extends A {
  public int k(int i) {
  }
  private int m() {
  }
}
```

Um programa Java deve satisfazer um número de regras de boa formação. Por exemplo, uma classe não pode estender a si mesma. A seguir, especificamos um conjunto de regras de boa formação.

Regras de boa formação

Para garantir a geração de programas bem formados, especificamos um fato `JavaConstraints` contendo restrições que descrevem algumas regras de boa formação de Java. A primeira delas especifica que uma classe não estende ela mesma direta ou indiretamente. A segunda e a terceira garantem que os métodos e atributos sejam declarados dentro das classes. As três primeiras são exibidas no Código Fonte 4.5.

Java não permite que uma classe possua dois atributos com o mesmo nome. Além disso, ela também não permite que uma classe possua dois métodos com o mesmo nome e mesmos argumentos. As duas restrições exibidas no Código Fonte 4.6 especificam essas regras.

Após especificar o subconjunto dos elementos de Java, as relações entre eles e regras

de boa formação, podemos utilizar o JDolly para gerar programas com a parte estrutural bem formada. Para um escopo de 2 classes, 2 métodos e 1 atributo, o JDolly gera 61.560 programas. O tempo total é de aproximadamente 255 segundos em um computador com processador dual-core de 2,2GHz e 4GB de RAM.

Código Fonte 4.5: Fato que declara algumas regras de boa formação de Java

```
fact JavaConstrains {
  no c:Class | c in c·^extend
  Method in Class·methods
  Field in Class·fields
```

Código Fonte 4.6: Regras de boa formação para atributos e métodos

```
no f1,f2: Field | some c: Class |
  f1 ≠ f2 &&
  f1 + f2 in c·fields &&
  f1·name == f2·name
no m1,m2: Method | some c: Class |
   m1 ≠ m2 &&
   m1 + m2 in c·methods &&
   m1·id == m2·id &&
   m1·arguments == m2·arguments
```

Geração de programas com determinadas características estruturais

Eventualmente, o usuário pode estar interessado apenas em programas com determinadas características estruturais, o que torna desnecessária a geração de certos programas. Levando isso em consideração, ele pode especificar fatos relacionados ao meta-modelo de Java mostrado anteriormente com restrições descrevendo características estruturais desejadas para os programas.

Por exemplo, suponha que o usuário tenha interesse apenas em programas que possuam pelo menos duas classes (classe pai e classe filho) e um método declarado na classe pai. O Código Fonte 4.7 mostra essa especificação feita em Alloy. A primeira linha especifica que deve existir uma classe `C1` e outra `C2`. O fato declara que `C2` estende `C1`, e que existe um método declarado em `C1`.

Código Fonte 4.7: Restrições para gerar programas com herança entre duas classes e um método na classe pai

```
one sig C1,C2 extends Class {}
fact Inheritance {
   C1 in C2·extend
   one m:Method | m in C1·methods
}
```

A partir dessa especificação, o JDolly gera 21.600 programas com essas características estruturais para o mesmo escopo de 2 classes, 2 métodos e 1 atributos. A execução é finalizada em aproximadamente 30 segundos na mesma máquina usada na execução anterior.

Implementação

A distribuição do Alloy fornece uma API Java [11] que permite manipular os comandos do Alloy Analyzer comandos e ter acesso as instâncias geradas em cada solução encontrada. Utilizamos essa API no JDolly para implementar a geração da parte estrutural. Cada instância gerada pelo Alloy é transformada em uma árvore sintática abstrata (AST) de Java. Utilizamos a API Eclipse Core [41] para representar a AST de Java.

Ao final dessa etapa, já temos um conjunto de programas gerados, porém com os corpos dos métodos vazios. A seguir, mostramos como é feita a geração da parte comportamental dos programas.

4.3 Geração da Parte Comportamental

Nessa seção, descrevemos a geração da parte comportamental do programa. Desenvolvemos geradores a partir do *framework* ASTGen [14] que são responsáveis por gerar instruções nos corpos dos métodos e nas declarações de atributos. O JDolly gera programas com as seguintes características comportamentais:

1. *atributos*. Cada atributo é inicializado com um valor aleatório que depende do seus tipo;

2. *métodos*. Cada método possui uma instrução de retorno. Essa instrução pode ter um valor aleatório baseado no tipo de retorno, ou uma expressão de chamada de método

para um dos métodos do programa (exceto ele mesmo) ou atributo (para um dos atributos do programa).

As chamadas de método e atributo do JDolly podem ser de quatro tipos:

1. *simples*. Por exemplo, `x();`
2. *this*. Por exemplo, `this.x();`
3. *qualified this*. Por exemplo, `A.this.x();`
4. *super*. Por exemplo, `super.x();`
5. *nova instância*. Por exemplo, `new A().x()`.

Na Seção 4.2, exibimos um exemplo da parte estrutural de um programa gerado pelo JDolly (Código Fonte 4.4). O programa completo, após o JDolly adicionar a parte comportamental, pode ser visto no Código Fonte 4.8. Na classe, `A` o JDolly criou uma instrução para o atributo `x` que inicializa-o com o valor `19`, e uma instrução para o método `k(int)` que retorna o valor 85. Em `B`, ele gerou no método `k(int)` uma instrução que retorna uma chamada a `x`, e em `m()` uma instrução de retorno usando o `super` com uma chamada a `k(int)` da classe pai.

Código Fonte 4.8: Exemplo de um programa gerado pelo JDolly

```
public class A {
  public int x = 19;
  public int k(int i) {
    return 85;
  }
}
public class B extends A {
  public int k(int i) {
    return x;
  }
  private int m() {
    return super.k(3);
  }
}
```

Mostramos uma visão geral do ASTGen na Seção 2.4. Ele possui possui 43 geradores para gerar elementos da sintaxe de Java. Utilizamos alguns desses geradores no JDolly para criar o gerador `BehavioralGenerator`, responsável por gerar a parte comportamental dos programas (Figura 4.1). Ele gera um número de programas com comportamentos distintos para cada parte estrutural gerada na etapa anterior. Esse gerador é composto por *n* geradores de inicialização de atributo (`AssignmentGenerator`), onde *n* corresponde ao número de atributos que o programa possui, e *m* geradores de corpo de método (`MethodBodyGenerator`), onde *m* corresponde ao número de métodos contidos no programa. A seguir, descrevemos esses geradores.

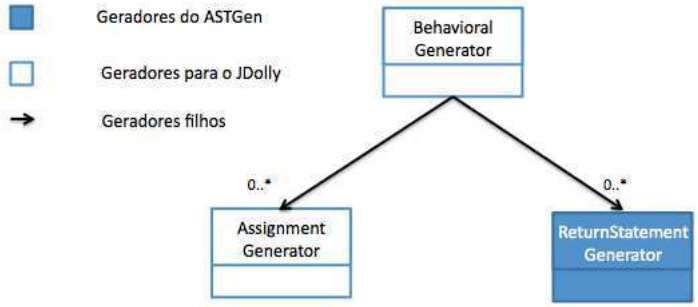

Figura 4.1: Diagrama de classes do gerador da parte comportamental

Gerador de comportamento de atributo

Utilizamos os geradores do ASTGen `StringLiteralGenerator`, `BooleanLiteralGenerator`, e `NumberLiteralGenerator` para compor o gerador `AssignmentGenerator`, responsável por inicializar o atributo com um valor aleatório. Este gerador só produz um valor dependendo do tipo do atributo. A versão atual do JDolly inicializa apenas atributos primitivos inteiros, booleanos ou *strings*. O diagrama de classes exibido na Figura 4.2 ilustra a relação entre esses geradores. O gerador `AssignmentGenerator` pode utilizar um dos três geradores do ASTGen para inicializar o atributo.

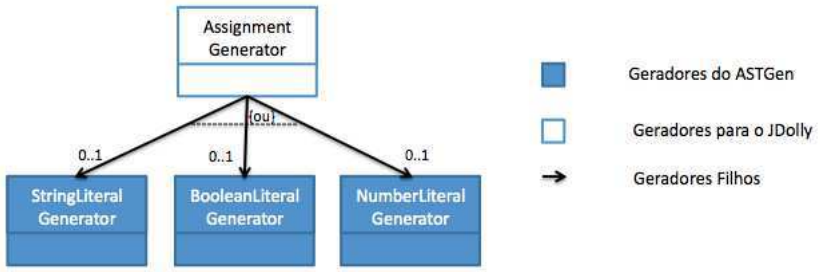

Figura 4.2: Diagrama de classes do gerador de inicialização de atributo

Gerador de comportamento de método

O gerador do ASTGen `ReturnStatementGenerator` cria instruções de retorno com expressões produzidas por um gerador filho. No JDolly, essas expressões são geradas pelo gerador `ExpressionGenerator` (Figura 4.3). Esse gerador cria 3 tipos de expressões:

1. chamadas de métodos para todos os métodos do programa (exceto o método que faz a chamada). Para gerar essas expressões, ele utiliza um gerador filho `MethodCallGenerator` para cada método que será chamado. Esse gerador gera até cinco tipos de expressões de chamadas para um determinado método;

2. chamadas de atributos para todos os atributos do programa. Ele utiliza um gerador filho `FieldCallGenerator` para cada atributo que será chamado. Esse gerador gera até cinco tipos de expressões de chamadas para um determinado atributo;

3. valores aleatórios baseados no tipo de retorno do método.

Os geradores `MethodCallGenerator` e `FieldCallGenerator` possuem alguns filtros filtros baseados nas regras de boa formação de Java para não gerar algumas expressões inválidas. Por exemplo, o gerador `MethodCallGenerator` não gera chamada com o `super` para um método que esteja na mesma classe do método que faz a chamada.

Adicionalmente, o desenvolvedor pode alterar os geradores de expressões para gerar programas com características comportamentais específicas. Por exemplo, suponha que ele deseja que não sejam geradas chamadas a atributos. Para isso, basta remover do gerador `ExpressionGenerator` os geradores filhos `FieldCallGenerator`. Além disso, ele pode criar novos geradores para aumentar a expressividade da parte comportamental do JDolly.

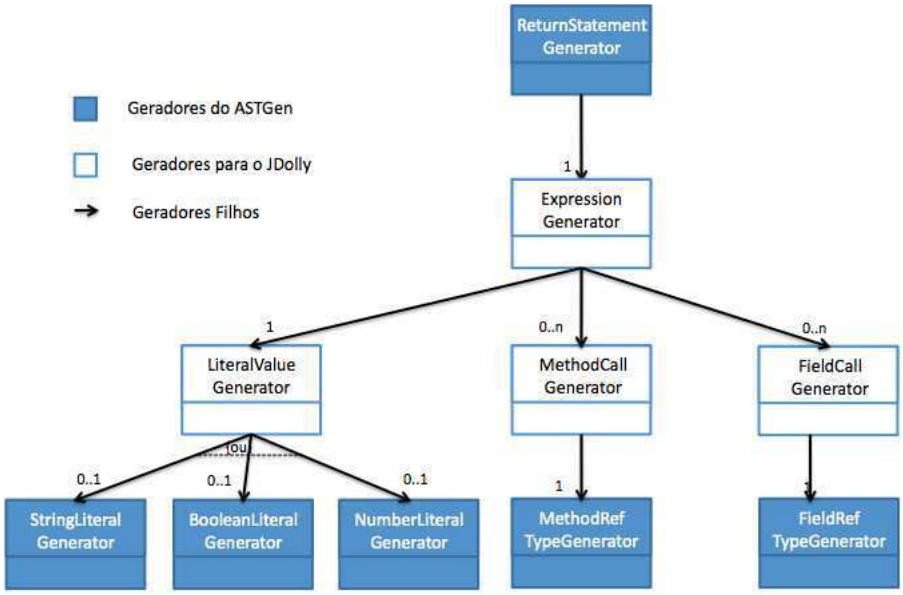

Figura 4.3: Diagrama de classes do gerador de instrução de retorno de método

Escopo da geração da parte comportamental

O gerador `BehaviorGenerator` produz todas as combinações dos geradores `AssignmentGenerator` e `ReturnStatementGenerator`. Por exemplo, para um programa com uma classe, dois métodos, e um atributo, o gerador `BehaviorGenerator` é instanciado com um gerador `AssignmentGenerator` (para o atributo), e dois geradores `ReturnStatementGenerator` (um para cada método). O gerador relacionado ao atributo produz um único valor aleatório. Cada gerador relacionado aos métodos produz 9 valores: 4 diferentes chamadas para o outro método, 4 diferentes chamadas para o atributo, e 1 valor aleatório. Nesse exemplo, o gerador `BehaviorGenerator` produzirá 81 programas com as mesmas características estruturais, mas comportamentos distintos.

4.4 Exemplo

Nessa seção, mostramos um exemplo de como gerar programas Java utilizando o JDolly. Suponha, que desejamos gerar programas com as seguintes características estruturais:

1. herança entre duas classes;

2. um atributo na classe filha;

3. atributos do tipo `int` ou `long`;

4. métodos com tipo de retorno `long`;

5. escopo de três classes, dois atributos, um método.

Especificamos essas características estruturais em arquivo, por exemplo "exemplo.als". O Código Fonte 4.9 exibe o conteúdo desse arquivo com as restrições adicionais que descrevem a estrutura desejada. Utilizamos o comando `open` para importar o meta-modelo e as restrições de boa formação que definimos de Java. A segunda linha modela as classes `C1` e `C2` que existirão nos programas. As duas primeiras restrições do fato `Design` especificam que `C2` estende `C1`, e que existe um atributo em `C2`, respectivamente. A terceira restrição declara que todos os atributos serão do tipo `int` ou `long`, e a quarta restrição declara que os métodos terão tipo de retorno `long`.

Código Fonte 4.9: Restrições para gerar programas com determinadas características

```
open java−meta−model
one sig C1,C2 extends Class {}
fact Design {
   C1 in C2·extend
   one f:Field | f in C2·fields
   all f:Field | f·type = Int || f·type = Long
   all m:Method | m·return = Long
}
```

Após definir a estrutura do programa em Alloy, é preciso inicializar o JDolly (Código Fonte 4.10). Instanciamos o construtor da classe `JDolly` passando os parâmetros: caminho para o arquivo "exemplo.als", número de classes (3), atributos (2) e métodos (1). Ao final, cada programa gerado é exibido na saída padrão de Java através do método `System.out.println`.

Ao executarmos o método `main`, em um computador com processador dual-core 2GHz e 4GB de RAM, o JDolly gera 487.296 programas em um tempo total de aproximadamente 79 segundos. O Código Fonte 4.11 exibe um dos programas gerados. A classe `C2` foi declarada estendendo `C1` e com um atributo `f` como especificado. Perceba que o que não

é restringido, o JDolly gera aleatoriamente, como a visibilidade dos métodos e atributos. Podemos observar ainda que as características desejadas para a geração são compatíveis com as características necessárias para aplicar o refatoramento *Push Down Field*. No Capítulo 5, mostramos como utilizamos nosso gerador para testar a implementação dos refatoramentos do Eclipse.

Código Fonte 4.10: Exemplo de como utilizar o JDolly para gerar programas na saída padrão

```
public static void main(String[] args) {
  int classes = 3;
  int atributos = 2;
  int metodos = 1;

  JDolly jdolly = new JDolly("exemplo.als", classes, atributos,metodos);
  for (CompilationUnit program : jdolly) {
      System.out.println(program);
    }
  }
}
```

Código Fonte 4.11: Programa gerado pelo JDolly com estrutura adequada para ser refatorado

```
public class C0 {
  protected int f = 10;
}
public class C1 extends C0 {}

public class C2 extends C1 {
  public int f = 20;
  private long m() {
    return super.f;
  }
}
```

4.5 Discussão

Nessa seção, discutimos algumas questões a respeito do JDolly, como a expressividade e variabilidade dos programas gerados, e a extensibilidade do gerador. Além disso, o comparamos com o ASTGen.

A principal motivação para desenvolvermos o JDolly foi criar um gerador de programas que fosse simples de especificar as características estruturais do programa, e que existisse uma boa variação das demais características nos programas gerados. Dessa forma, poderíamos gerar uma grande quantidade de programas para avaliar a eficiência do SAFEREFACTOR em detectar falhas em ferramentas de refatoramento. Nesse ponto, a abordagem declarativa do JDolly mostrou-se superior a abordagem imperativa do ASTgen, pois nela especificamos o que queremos que seja gerado, sendo que, o que não é especificado é gerado aleatoriamente.

Por exemplo, ao passarmos para o JDolly o escopo de X classes e Y atributos, para qualquer X e Y, sem mais nenhuma especificação, ele gera todas as combinações entre esses atributos e classes. Por outro lado, no ASTGen, precisamos especificar detalhadamente como será gerada toda a estrutura dos programas, tanto a estrutura que queremos manter fixa quanto a que queremos variar. Por exemplo, devemos programar as relações entre classes (herança). Portanto, o esforço necessário para gerar a parte estrutural dos programas em geral é menor no JDolly.

Para aumentar a expressividade da parte estrutural dos programas gerados pelo JDolly, é necessário estender o nosso meta-modelo. Por exemplo, suponha que desejamos adicionar pacotes e interfaces. Para isso, precisamos adicionar novas assinaturas representando-os, especificar relações destes com os demais elementos do modelo, e especificar novas regras de boa formação.

Com relação a geração da parte comportamental dos programas, optamos pela abordagem imperativa, utilizando o ASTGen. Instruções e expressões de Java possuem estruturas complexas que não são fáceis de serem modeladas em Alloy. Além disso, restrições complexas tornariam a análise do Alloy Analyzer mais lenta, o que comprometeria o desempenho do JDolly.

O ASTGen possui 13 geradores de programas, além dos 59 geradores de elementos de

Java. A expressividade estrutural desses geradores é inferior em relação ao JDolly. Por exemplo, 4 destes geradores geram programas com apenas uma classe, outros 7 geram programas com duas classes, e os últimos 2 geram programas com três classes. Além disso, a número de métodos e atributos também é fixo. No JDolly, a quantidade máxima desejada de entidades (classes, métodos, atributos) dos programas é passada como parâmetro, o que torna fácil o número de entidades de geração para geração.

Por outro lado, esses geradores possuem uma maior expressividade da parte comportamental dos programas em relação ao JDolly. A versão atual do JDolly gera métodos com apenas uma instrução de retorno, enquanto que os geradores do ASTGen geram métodos com estruturas de laço e repetição por exemplo. Pretendemos explorar mais os geradores da parte comportamental do ASTGen em trabalhos futuros.

Nosso meta-modelo em Alloy especifica um subconjunto de Java. Entre as características que os programas gerados podem ter, estão: sobrecarga e sobrescrita de método; atributos declarados com o mesmo nome na mesma hierarquia de classes; expressões de chamada de método e atributo usando o `super`. Como vimos no Capítulo 3, essas características são possíveis fontes de erros em refatoramentos. Na Seção 5.3, descrevemos como utilizamos o JDolly para testar os refatoramentos automatizados pelo Eclipse.

Capítulo 5

Avaliação

Neste capítulo, descrevemos os experimentos realizados para avaliar nossa técnica para detectar mudanças comportamentais após a atividade de refatoramento (Capítulo 3) e nosso gerador de programas JDolly (Capítulo 4). Primeiramente, mostramos um experimento com um conjunto contendo refatoramentos incorretos aplicados por ferramentas de refatoramento que foram catalogados na literatura (Seção 5.1). Essas transformações não preservam o comportamento dos programas. Nós avaliamos o desempenho de nossa técnica nessas mudanças comportamentais. Em seguida, descrevemos um experimento que realizamos envolvendo refatoramentos em sete sistemas reais em Java (Seção 5.2). Utilizamos nossa técnica para checar se as transformações preservavam o comportamento dos programas. Por último, descrevemos como utilizamos nossa técnica e o JDolly para testar o módulo de refatoramento do Eclipse 3.4 (Seção 5.3). Nesse experimento avaliamos o poder do SAFEREFACTOR para detectar mudanças comportamentais e erros de compilação.

5.1 Refatoramentos Problemáticos

Nesta seção, descrevemos o experimento que realizamos para avaliar a eficiência de nossa técnica em detectar mudanças comportamentais. Nós catalogamos na literatura 16 transformações [1], aplicados por ferramentas de refatoramento, que não preservam o comportamento, e avaliamos se nossa técnica era capaz de detectar essas mudanças.

[1] Todos os dados relacionados a este experimento podem ser encontrados em: http://dsc.ufcg.edu.br/~spg/saferefactor/experiments.htm

Id	Refatoramento	Descrição do Problema
1	Rename Class	Introduz shadowing
2	Rename Variable	Introduz shadowing por atributo
3	Rename Method	Introduz shadowing de método estático importado
4	Encapsulate Field	Habilita sobrescrita
5	Extract Method	Análise incorreta do fluxo de dados
6	Push Down Method	Tratamento incorreto do invocação de superclasse
7	Push Down Method	Tratamento incorreto da invocação de atributo
8	Push Down Method	Tratamento incorreto da invocação de classe
9	Move Class	Disabilita sobrescrita entre métodos em pacotes diferentes
10	Move Class	Disabilita sobrecarga entre métodos em pacotes diferentes
11	Change Method Signature	Habilita sobrescrita
12	Change Method Signature	Habilita sobrecarga
13	Change Method Signature	Habilita sobrescrita entre métodos em pacotes diferentes
14	Pull Up Method	Habilita sobrecarga
15	Pull Up Method	Habilita sobrescrita
16	Pull Up Method	Habilita sobrescrita entre métodos em pacotes diferentes

Tabela 5.1: Transformações utilizadas no experimento

Primeiro, descrevemos os refatoramentos avaliados, e em seguida, as configurações para o experimento. Por fim, apresentamos e discutimos os resultados obtidos.

5.1.1 Caracterização das Transformações

Os refatoramentos catalogados apresentam mudanças comportamentais que não são detectadas por pelo menos uma das seguintes IDEs que implementam refatoramentos: Eclipse 3.4.2, JBuilder 2007, Netbeans 6.0, JDeveloper 11g, e IntelliJ 8.0. Eles foram manualmente identificados por Ekman et al. [19], e Steimman e Thies [70]. A Tabela 5.1 exibe a lista dessas transformações e a descrição do problema de cada uma.

Os programas refatorados são compostos por um pequeno conjunto de classes (2-5) contendo métodos com argumentos e parâmetros de tipos primitivos e não primitivos. Cada método possuí uma instrução de retorno. Originalmente, os métodos possuíam tipo de retorno `void` e retornavam valores para a saída padrão de Java. Alteramos os métodos para que fosse possível testá-los através de testes de unidade do JUnit. Os programas possuem pelo menos um método público. Essa restrição não invalida nossos resultados, pois a prática de testes de unidade é utilizada em métodos públicos, e nossa noção de equivalência é base-

ada na observação dos métodos públicos. A seguir, mostramos 4 dessas 16 transformações.

Transformação 3. O Código Fonte 5.1 exibe um programa no qual é aplicado o refatoramento *Rename* ao método `m`, alterando seu nome para `valueOf` (Código Fonte 5.2). Essa transformação não preserva o comportamento, porque depois de ser aplicada, o método `test` retorna 42, ao invés de 23, valor retornado no programa original. Isso ocorre porque depois da transformação, o método `valueOf` da classe `A` é chamado ao invés do método `java.lang.String.valueOf`. A mudança comportamental não é detectada no Eclipse, JBuilder, Netbeans, e JDeveloper.

Figura 5.1: Refatoramento Rename Method introduz shadowing de método estático importado

Código Fonte 5.1: Versão Original

```
import static java.lang.*;
public class A {
  static String m(int i) {
    return "42";
  }
  public int test() {
    return Integer.parseInt(
           valueOf(23));
  }
}
```

Código Fonte 5.2: Versão Refatorada

```
import static java.lang.*;
public class A {
  static String valueOf(int i){
    return "42";
  }
  public int test() {
    return Integer.parseInt(
           valueOf(23));
  }
}
```

Transformação 10. O Código Fonte 5.3 exibe um programa com duas classes `A` e `B`. A classe `B` possui uma sobrecarga no método `k`. Esse método é invocado com um parâmetro do tipo `int` pelo método `test`, portanto, o método escolhido é o `k(int)`. Ao aplicar o refatoramento *Move Class* na classe `B`, movendo-a para o pacote `b`, o programa resultante fica similar ao apresentado no (Código Fonte 5.4). Essa transformação não preserva o comportamento, porque depois de ser aplicada, o método `test` retorna 42, ao invés de 23, valor retornado no programa original. Isso ocorre porque depois da transformação, o método `k(int)` não é visível no pacote `a`, pois ele é declarado com visibilidade `package`. Então, `test` passa a invocar `k` declarado na linha 9. Apesar da chamada ao método passar um

valor `int` para um parâmetro `long`, não ocorre erro de compilação. A partir do Java 1.5, a conversão de `int` para `long` é implícita (*Widening Primitive Conversion*) e checada em tempo de compilação, não havendo necessidade de usar um `cast` explícito [31].

Figura 5.2: Move Class desabilita sobrecarga

Código Fonte 5.3: Versão Original

```
package a;
public class A {
  public int test() {
    return (new B()).k(2);
  }
}
package a;
public class B {
  public int k(long l) {
    return 42;
  }
  int k(int i) {
    return 23
  }
}
```

Código Fonte 5.4: Versão Refatorada

```
package a;
public class A {
  public int test() {
    return (new B()).k(2);
  }
}
package b;
public class B {
  public int k(long l) {
    return 42;
  }
  int k(int i) {
    return 23
  }
}
```

Transformação 12. O Código Fonte 5.5 exibe um programa com a classe pai `A` e classe filha `B`. O método `test` invoca `m()` e retorna o valor 10. Note que apesar de `test` ter sido declarado em `B`, o método `m` chamado por ele invoca `k()` da classe A, mesmo existindo um `k()` em `B`. Isso ocorre porque não é possível sobrescrever um método privado. Ao aplicar o refatoramento *Change Method Signature* no método `A.k()`, aumentando a visibilidade dele para `package`, o programa resultante fica similar ao apresentado no (Código Fonte 5.6). Essa transformação não preserva o comportamento, pois depois de ser aplicada, o método `test` retorna 20, ao invés de 10, valor retornado no programa original. Isso ocorre porque depois ao aumentar a visibilidade do método `A.k()`, habilitamos a sobrescrita dele pelo método `B.k()`.

Transformação 14. O Código Fonte 5.7 exibe um programa com a classe pai `A` e sua classe

Figura 5.3: Change Visibility aumentando a visibilidade do método habilita sobrescrita

Código Fonte 5.5: Source Version

```
public class A {
  private int k() {
    return 10;
  }
  int m() {
    return k();
  }
}
public class B extends A {
  int k() {
    return 20;
  }
  int test() {
    return m();
  }
}
```

Código Fonte 5.6: Target Version

```
public class A {
  int k() {
    return 10;
  }
  int m() {
    return k();
  }
}
public class B extends A {
  int k() {
    return 20;
  }
  int test() {
    return m();
  }
}
```

filha `B`. O método `test()` faz uma chamada a `A.k(long)` e retorna o valor 10. Ao aplicar o refatoramento *Pull Up Method* no método `B.k(int)`, subindo-o para a classe `A`, o programa resultante fica similar ao apresentado na (Código Fonte 5.8). Essa transformação não preserva o comportamento, porque depois de ser aplicada, o valor retornado pelo método `test()` é alterado para 20. Isso ocorre porque depois da transformação, acontece uma sobrecarga no método `k`, então, `test()` passa a invocar `A.k(int)` ao invés de `A.k(long)`.

5.1.2 Configuração do Experimento

Nós executamos o experimento em um laptop Dell Vostro 1400 com processador dual-core 2.2GHz e 2 GB de RAM utilizando o Linux Ubuntu 9.04. Utilizamos a interface de linha de comando do SAFEREFACTOR (Seção 6.2) para aplicar nossa técnica nas transformações selecionadas. Ela fornece um método que recebe três parâmetros: o diretório do projeto com

Figura 5.4: Pull Up Method habilita sobrecarga

Código Fonte 5.7: Versão Original

```
public class A {
  public int k(long i) {
    return 10;
  }
}
public class B extends A {
  public int k(int i) {
    return 20;
  }
  public int test() {
    return new A().k(2);
  }
}
```

Código Fonte 5.8: Versão Refatorada

```
public class A {
  public int k(long i) {
    return 10;
  }
  public int k(int i) {
    return 20;
  }
}
public class B extends A {
  public int test() {
    return new A().k(2);
  }
}
```

o programa original, o diretório com o projeto do programa refatorado, e o tempo limite para a geração de testes. Nós utilizamos o tempo de apenas 2 segundos para geração dos testes, devido ao tamanho das transformações.

5.1.3 Resultados do Experimento e Discussão

Nesta seção, mostramos e discutimos os resultados do nosso experimento. Nossa técnica detectou todas mudanças comportamentais em menos de oito segundos para cada transformação. A Tabela 5.2 exibe os dados da execução do SAFEREFACTOR para cada transformação. Cada linha contém o número de testes gerados pela técnica (Coluna Testes), e o número de testes que passaram no programa original, mas falharam no refatorado (Coluna Falhas). A Coluna Detectou indica se a mudança comportamental foi detectada por nossa técnica.

As transformações avaliadas possuem métodos com instruções de retorno (`return`) que podem ser avaliadas por nossa técnica. Ela foi capaz de detectar mudanças em valores de tipo primitivo (`int`, `double`) e `Object`. A implementação corrente da nossa técnica não detecta mudanças comportamentais na saída padrão da linguagem Java (`System.out.println`). Nós planejamos usar algum padrão de testes que seja útil para

Id	Testes	Falhas	Detectou?
1	488	0	NÃO
2	102	95	SIM
3	494	492	SIM
4	93	91	SIM
5	474	464	SIM
6	558	554	SIM
7	486	404	SIM
8	78	75	SIM
9	101	99	SIM
10	101	99	SIM
11	79	77	SIM
12	214	40	SIM
13	79	76	SIM
14	121	40	SIM
15	101	99	SIM
16	170	88	SIM
17	167	163	SIM

Tabela 5.2: Resultados do Experimento; Testes = testes gerados; Falhas = testes que passaram na versão original e falharam na refatorada; Detectou? = indica se a mudança comportamental foi detectada

detectar alterações nesses tipos de mensagens e incorporar ao Randoop para que ele gere testes que identifiquem estas mudanças.

O refatoramento *Rename Method* pode habilitar ou desabilitar a sobrecarga ou sobrescrita de um método. Isso é uma potencial fonte de mudanças comportamentais [61]. Apesar desse refatoramento influenciar na análise estática realizada na terceira etapa (o método renomeado não é incluído na geração dos testes), nossa técnica conseguiu detectar o problema referente a esse refatoramento (Transformação 3), pois esse refatoramento afetou o comportamento dos métodos públicos comuns. Como vimos na Seção 3.2, estes métodos são os alvos dos testes gerados pela nossa técnica. Dessa forma, conseguimos exercitar indiretamente a mudança ocorrida. De maneira similar, nossa técnica consegue detectar problemas com *inner classes* e classes não públicas (Transformação 1).

Id	Programa	KLOC	Refatoramento
1	JHotDraw	23	Extract Exception Handler
2	CheckStylePlugin	20	Extract Exception Handler
3	Junit	3	Infer Generic Type Argument
4	Vpoker	4	Infer Generic Type Argument
5	ANTLR	32	Infer Generic Type Argument
6	Xtc	100	Infer Generic Type Argument
7	TextEditor	15	Replace Deprecated Code

Tabela 5.3: Transformações utilizadas no experimento; KLOC = linhas de código (excluindo linhas de comentários e em branco)

5.2 Refatoramentos em Programas Reais

Nesta seção, descrevemos como utilizamos nossa técnica para checar a existência de mudanças comportamentais em 7 refatoramentos realizados em programas reais com tamanhos entre 3 a 100 mil linhas de código.

Primeiro, descrevemos as transformações avaliadas. Em seguida, configurações do nosso experimento. Por fim, mostramos e discutimos os resultados obtidos.

5.2.1 Caracterização das Transformações

Nós utilizamos 7 transformações[2] nesse experimento, como é mostrado na Tabela 5.3. Elas foram extraídas de alguns estudos de casos e experimentos encontrados na literatura [72; 24; 53]. As transformações consistem de refatoramentos em sistemas reais desenvolvidos em Java. Esses refatoramentos foram aplicados manualmente ou com auxílio de ferramentas pelos desenvolvedores. A coluna "KLOC" exibe o tamanho de cada aplicação em linhas de código (excluindo comentários e linhas em branco). Nós utilizamos nossa técnica para avaliar se havia mudanças comportamentais entre os programas originais e refatorados. A seguir, descrevemos essas transformações.

[2]Todos os dados relacionados a este experimento podem ser encontrados em: http://dsc.ufcg.edu.br/~spg/saferefactor/experiments.htm

Transformações 1-2

JHotDraw é um framework Java para desenvolvimento de gráficos. Ele contém aproximadamente 23 KLOC, e mais de 400 classes e interfaces. CheckStylePlugin é um plugin para Eclipse que permite checar se o programa está em conformidade com um conjunto de padrões de código.

Desenvolvedores experientes realizaram um refatoramento no JHotDraw e no CheckStylePlugin para modularizar o código relacionado com tratamento de exceções [72]. O objetivo foi evitar duplicação de código ocasionada por tratamentos de exceção idênticos em diferentes partes do sistema, um problema já conhecido [6]. Oito desenvolvedores trabalhando em pares realizaram a mudança: eles extraíram o código contido nos blocos `try`, `catch`, e `finally` para métodos em classes específicas para o tratamento de exceções. Eles utilizaram ferramentas de refatoramentos, revisão em par, e teste de unidades para assegurar que o comportamento tinha sido preservado.

Transformações 3-6

A JDK 1.5 introduziu várias extensões à linguagem de programação Java, dentre elas a *Generics*. Esta permite ao desenvolvedor realizar abstração sobre o tipo do objeto. Seu uso mais comum é em estruturas de dados tais como as fornecidas pela API `Collections`. Na versão anterior a JDK 1.5, os objetos dessa API só retornavam objetos do tipo `Object`, então era preciso fazer um *cast* para recuperar o tipo do objeto. Quando ele era feito incorretamente, uma exceção do tipo `ClassCastExceptions` era lançada em tempo de execução. Já na JDK 1.5, `Generics` permite ao desenvolvedor declarar o tipo de objeto que será utilizado na estrutura de dados.

Fuhrer et al. [24] propuseram um algoritmo para substituir o uso de classes não genéricas por classes que utilizam *Generics*. Esse refatoramento é chamado de *Infer Generic Type Argument*, e é implementado no Eclipse. Eles avaliaram esse refatoramento em algumas aplicações Java, entre elas: JUnit, Vpoker, ANTLR, e Xtc.

Transformação 7

Murphy-Hill et al. [53] realizaram alguns experimentos para analisar como os desenvolvedores refatoram o código. Eles manualmente identificaram refatoramentos em um conjunto de 20 transformações de componentes do Eclipse extraídos do CVS dele. Entre essas transformações, foi identificado um refatoramento no módulo TextEditor do Eclipse. Os desenvolvedores alteraram o código para substituir métodos desatualizados por versões recentes.

5.2.2 Configurações do Experimento

Nós executamos o experimento em um laptop Dell Vostro 1400 com processador dual-core 2.2GHz e 2 GB de RAM utilizando o Linux Ubuntu 9.04. Assim como antes (Seção 5.1), utilizamos a interface de linha de comando do SAFEREFACTOR (Seção 6.2) para aplicar nossa técnica aos refatoramentos. Nós utilizamos o tempo de 90 segundos para geração dos testes. Para determinar o tempo, aplicamos o Randoop a um dos programas avaliados com diferentes limites de tempo: 30, 60, 90, 120, 150, 180. Notamos que a partir de 90 segundos a cobertura de testes tende a ficar estável.

5.2.3 Resultados do Experimento e Discussão

Nossa técnica detectou uma mudança comportamental em um refatoramento. Além disso, o SAFEREFACTOR reportou erros de compilação em outras duas transformações. O tempo total da técnica para cada transformação foi inferior a quatro minutos. A Tabela 5.4 exibe os dados da execução do SAFEREFACTOR para cada transformação. Cada linha contém o número de testes gerados pela técnica (Coluna Testes), e o número de testes que passaram no programa original, mas falharam no refatorado, caracterizando a mudança comportamental (Coluna Falhas).

A mudança comportamental detectada ocorreu no refatoramento realizado no JHotDraw (Transformação 1). Uma parte do código original dele pode ser vista no Código Fonte 5.9. A classe `Figure` implementa a interface `Serializable`. Os objetos dela podem ser codificados para fluxos de bytes, transmitidos de uma máquina virtual para outra ou armazenados em discos, e reconstruídos posteriormente. Desenvolvedores experientes extraíram o código referente a tratamento de exceções do método `clone` para uma classe específica para fa-

Id	Programa	Testes	Falhas	Tempo total(s)	Resultado
1	JHotDraw	2245	273	148	Mudança Comportamental
2	CheckStylePlugin	5864	0	235	-
3	Junit	1127	0	99	-
4	Vpoker	466	0	109	-
5	ANTLR	-	-	2	Erro de Compilação
6	Xtc	-	-	4	Erro de Compilação
7	TextEditor	16009	0	107	-

Tabela 5.4: Resultados do Experimento; Testes = número de testes gerados; Falhas = número de testes que passaram na versão original e falharam na refatorada; Resultado = problemas encontrados

zer esse tratamento (`ExceptionHandler`). No código refatorado (Código Fonte 5.10), eles adicionaram um objeto dessa classe como atributo da classe `Figure`. Porém, o novo atributo não é serializado. Então, quando o método `clone` tenta serializar o objeto, uma exceção é lançada. Portanto, o método refatorado `clone` tem um comportamento diferente da versão original dele. O Código Fonte 5.11 mostra um teste de unidade, gerado por nossa técnica, que revela a mudança comportamental.

O SAFEREFACTOR levou 148 segundos para detectar a mudança comportamental no JHotDraw: 5 segundos para identificar os métodos em comum (Etapa 1), 90 segundos para gerar os testes de unidade (Etapa 2), e 53 segundos para executar os testes nas duas versões do programa (Etapas 3 e 4).

Ferramentas de refatoramento podem introduzir um erro similar ao detectado no JHotDraw. Por exemplo, dada uma classe `A` que implementa `Serializable`. Ao aplicar, utilizando o Eclipse, o refatoramento *Extract Class* para extrair dois atributos dessa classe para uma nova classe `B`, essa nova classe criada pelo Eclipse não implementa `Serializable`. Essa transformação produz um problema similar ao detectado no JHotDraw.

Os refatoramentos aplicados pelo Eclipse ao ANTLR e Xtc geraram erros de compilação. Por isso, não pudemos aplicar nossa técnica nessas transformações. Nossa técnica não identificou problemas nas Transformações 2-4 e 7.

Com relação às Transformações 2-4 e 7, nossa técnica não detectou mudanças comportamentais. Os testes cobriram 52% do código do JUnit, e 14,40% do código do Vpoker, dessa forma, temos mais confiança de que essas transformações preservam o comportamento.

Figura 5.5: Refatoramento incorreto aplicado ao JHotDraw

Código Fonte 5.9: Versão Original

```
class Figure implements Serializable {
  Object clone() { ...
    try {
      ObjectOutput writer = new ObjectOutputStream(output);
      writer.writeObject(this); ...
    }
    catch (IOException e) {
      System.err.println("Class not found: " + e);
    } ...
  } ...
}
```

Código Fonte 5.10: Versão Refatorada

```
class Figure implements Serializable{
  ExceptionHander handler; ...
  Object clone() {
    try {
      ObjectOutput writer = new ObjectOutputStream(output);
      writer.writeObject(this); ...
    }
    catch (IOException e) {
      handler.handle(e);
    } ...
  } ...
}
class ExceptionHandler { ... }
```

Por outro lado, os testes gerados por nossa técnica só cobriram 7,2% e 11% do código do CheckStylePlugin e TextEditor, respectivamente. Plugins do Eclipse precisam de coleções de testes específicas para o ambiente do Eclipse, denominadas *Junit Plugin Test Suite* [3]. Acreditamos que esse foi o motivo da baixa cobertura do código pelos testes gerados.

Código Fonte 5.11: Teste de unidade que detecta a mudança comportamental

```
public void test0() {
  Figure var1 = Figure();
  int var3 = var1.getZValue();
  java.lang.Object var4 = var1.clone();
  assertTrue(var3 == 0);
}
```

5.3 Implementação de Refatoramentos do Eclipse

Nessa seção, mostramos como combinamos nossa técnica com o nosso gerador de programas Java para avaliar a corretude de 11 implementações de refatoramentos do Eclipse. Vimos que o desenvolvimento dessas ferramentas é complexo, pois propor o conjunto de pre-condições correto para um refatoramento é difícil. Além disso, essas ferramentas devem ser confiáveis, pois falhas em seu desenvolvimento podem silenciosamente introduzir mudanças comportamentais nos programas. Por isso, os desenvolvedores dessas ferramentas investem muito em testes. Para se ter uma idéia, o Eclipse 3.2 tem mais de 2.600 testes de unidade para o módulo de refatoramentos [14]. Porém, escrever esses testes manualmente é cansativo.

Na Seção 5.3.1, mostramos uma visão geral da nossa abordagem e um exemplo. Em seguida, descrevemos os refatoramentos do Eclipse avaliados (Seção 5.3.2), e detalhamos as configurações do experimento (Seção 5.3.3). Na Seção 5.3.4, mostramos os resultados obtidos no experimento. Por fim, discutimos nossos resultados e comparamos com trabalhos relacionados (Seção 5.3.6).

[3]Processo de testes de plugins do Eclipse: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.hyades.test.doc.user/concepts/cintro.htm

5.3.1 Visão Geral

Nessa seção, mostramos uma visão geral da nossa abordagem para testar ferramentas de refatoramento. O nosso alvo foi o módulo de refatoramento do Eclipse 3.4.2. A seguir, descrevemos a abordagem utilizada para testá-lo.

Para testar a implementação de um refatoramento, nossa abordagem recebe como entrada a especificação para geração de programas pelo JDolly (Etapa 1). O testador especifica a geração de programas aptos a serem refatorados. Na Etapa 2, o JDolly realiza a geração dos programas especificados. Para cada programa gerado, utilizamos a API de refatoramento do Eclipse, para refatorá-lo (Etapa 3). Em seguida, utilizamos o SAFEREFACTOR para checar a existência de erros de compilação ou mudanças comportamentais (Etapa 4). Após testar os refatoramentos em todos os programas gerados, reportamos para o testador os erros encontrados (Etapa 5). O processo da nossa abordagem é ilustrado na Figura 5.6.

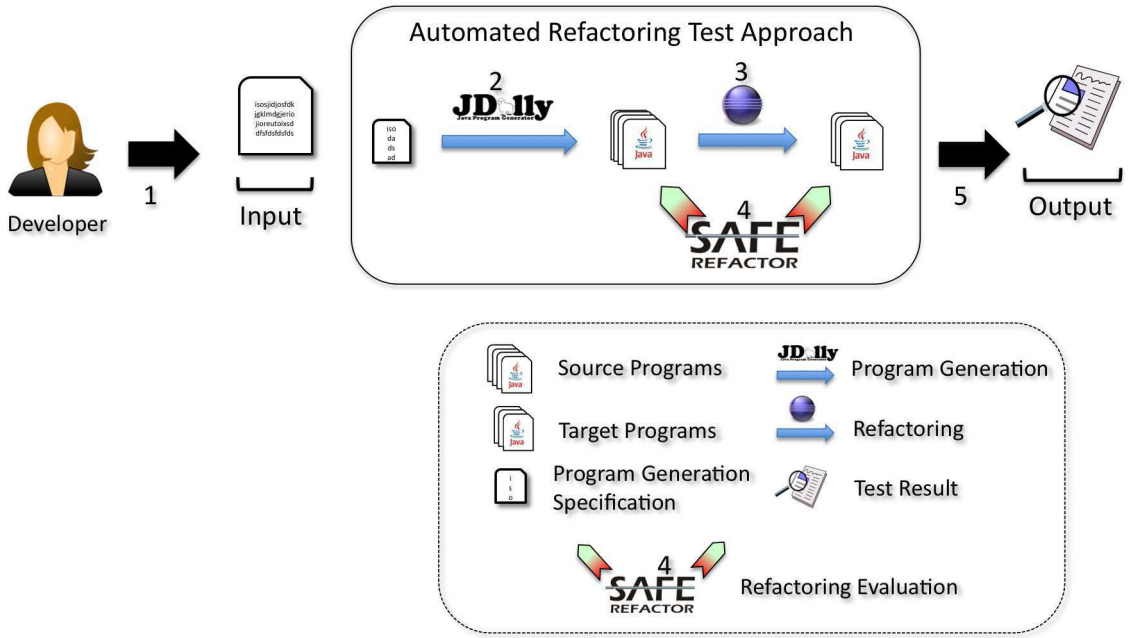

Figura 5.6: Abordagem para automatizar testes de ferramentas de refatoração; 1. testador especifica a geração de programas adequados para o refatoramento; 2. JDolly realiza a geração de programas; 3. utilizamos a API do Eclipse para refatorar os programas gerados pelo JDolly; 4. SafeRefactor avalia cada refatoramento para checar se a transformação gerou mudanças comportamentais; 5. Reportamos os resultados do SafeRefactor para o testador

Apesar de ter sido realizada para testar os refatoramentos implementados pelo Eclipse, essa abordagem pode ser utilizada para testar outras ferramentas de refatoramento para linguagens orientadas a objetos.

Exemplo

Nessa seção, mostramos um exemplo de como utilizar nossa abordagem. Suponha que desejamos testar a implementação do refatoramento `Push Down Method` do Eclipse. Primeiro, especificamos em Alloy a estrutura que os programas gerados pelo JDolly devem possuir. Para aplicar esse refatoramento, o programa deve possuir a seguinte estrutura:

- uma classe pai e uma filha;
- um método na classe pai.

Especificamos essas restrições em um arquivo chamado "pushdown.als" exibido no Código Fonte 5.12. A primeira linha importa o meta-modelo Java (Seção 4.2). Em seguida, definimos que existirão duas classes `C1` e `C2` e o método `M1`. As restrições do fato `PushDownMethod` estabelecem que `C2` estende `C1`, e que `M1` é declarado em `C1`, respectivamente.

Código Fonte 5.12: Restrições para o Push Down Method

```
open java−meta−model
 one sig C1,C2 extends Class {}
 one sig M1 extends Method {}
 fact PushDown{
    C1 in C2·extend
    M1 in C1·methods
}
...
```

É preciso especificar também algumas restrições para identificar o método que será refatorado utilizando a API do Eclipse. Sendo assim, especificamos que o método `M1` terá o identificador `N1` (Código Fonte 5.13). Além dessas restrições, é interessante que exista algum método no programa gerado que exercite o método a ser refatorado para podermos

avaliar o comportamento dos programa antes e depois do refatoramento. Por isso, especificamos também restrições para que seja gerado um método `test` sem parâmetro e público (Código Fonte 5.13). Sua única função é exercitar o método refatorado.

Código Fonte 5.13: Restrições adicionais para testar o refatoramento Push Down Method

```
...
one sig N1 extends Id {}
fact EclipseRefactoring {
  M1·id = N1
}
one sig Test extends Id {}
fact Extra {
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
\
```

Definimos a parte estrutural dos programas. O que não foi definido é gerado aleatoriamente de forma exaustiva em relação ao escopo. Em resumo, os programas gerados terão as seguintes características estruturais:

- uma classe pai `C1` e uma filha `C2`;
- um método `n1` na classe pai;
- um método público `test()`.

Código Fonte 5.14: Pseudo-código para testar a implementação do Push Down Method

```
JDolly jdolly = new JDolly("pshdwn.als", 3 , 0 , 4);
for (Program source : jdolly) {
  if (!source.compiles) {continue;}
  Refactoring r = new PushDownMethodRefactoring();
  r.setTargetMethod("n1");
  Program target =  r.performRefactoring(source);
  Report report = SafeRefactor.checkRefactoring(source,target);
  logger.logReport(report);
  }
}
```

Após especificar a geração dos programas (Etapa 1), inicializamos o JDolly passando como parâmetro o arquivo contendo as especificações da parte estrutural e o escopo de no máximo três classes e quatro métodos (Etapa 2). Para cada programa gerado pelo JDolly, checamos se ele compila e depois aplicamos o refatoramento (Etapa 3). Em seguida, enviamos para o SAFEREFACTOR o programa original e o refatorado para ele checar se o comportamento foi preservado (Etapa 4). Todos os resultados são armazenados para ao final serem disponibilizados para o usuário (Etapa 5). O Código Fonte 5.14 ilustra um pseudo-código para testar esse refatoramento do Eclipse.

A seguir, descrevemos os refatoramentos implementados pelo Eclipse que foram avaliados nesse experimento.

5.3.2 Caracterização dos Refatoramentos

Nós testamos 11 refatoramentos implementados pelo Eclipse 3.4.2. Abaixo, descrevemos esses refatoramentos:

- *Rename Method*: renomeia um método e atualiza as referências a este;
- *Rename Field*: renomeia um atributo e atualiza as referências a este;
- *Push Down Method*: move um método da classe pai para todas as classes filhas;
- *Push Down Field*: move um atributo da classe pai para todas as classes filhas;
- *Pull Up Method*: move o método de uma classe filha para a classe pai;
- *Pull Up Field*: move o atributo de uma classe filha para a classe pai;
- *Encapsulate Field*: torna um atributo privado e cria métodos de acesso para este. Adicionalmente, altera as referências ao atributo para os métodos de acesso;
- *Move Method*: move um método de uma classe para outra;
- *Change Method Signature (Add Parameter)*: adiciona um parâmetro a um método;
- *Change Method Signature (Remove Parameter)*: remove um parâmetro de um método;
- *Change Method Signature (Change Visibility)*: altera a visibilidade de um método.

O principal critério utilizado na escolha dos refatoramentos foi o grau de utilização deles pelos desenvolvedores no Eclipse. Para definir esse critério, nos baseamos na pesquisa conduzida por Murphy et al. [51]. Eles analisaram o uso da ferramenta de refatoramento do Eclipse por 41 desenvolvedores. Os cinco refatoramentos mais usados segundo essa pesquisa são: *Rename*, *Move*, *Extract Method*, *Pull Up*, e *Change Method Signature*. O Eclipse 3.4.2 possui 28 refatoramentos implementados (Tabela 2.2), portanto, 17 refatoramentos não foram testados, mas pretendemos testá-los em trabalhos futuros.

5.3.3 Configuração do Experimento

Nessa seção, mostramos como especificamos o JDolly para gerar programas úteis para testar os 11 refatoramentos selecionados. Além disso, descrevemos o *hardware* e *software* utilizados no experimento.

Para cada execução da nossa abordagem de testes, especificamos o *template* do programa a ser gerado, e o refatoramento a ser realizado. No JDolly, especificamos restrições na geração da parte estrutural referentes ao *template* dos programa baseado em um refatoramento e ao elemento que será refatorado pela API do Eclipse. Além disso, definimos um escopo fixo para o número de classes, atributos, e métodos para a geração dos programas.

Ao total, especificamos 14 execuções da nossa abordagem, como é exibido Tabela 5.5. Cada linha representa uma execução. A Coluna Escopo exibe o número exato de classes, atributos, e métodos dos programas gerados. Descrevemos as restrições da parte estrutural para cada refatoramento no Apêndice A.

Para os refatoramentos *Push Down Method* e *Pull Up Method* executamos os testes com mais de um escopo visando focar cada execução em uma possível fonte de problema. Por exemplo, na primeira execução referente ao *Pull Up Method*, geramos programas com três classes e quatro métodos; o nosso objetivo foi detectar alterações em chamadas de método. Por outro lado, na segunda execução, geramos programas com três classes, dois atributos, e dois métodos. Nessa execução o nosso objetivo foi detectar mudanças nas chamadas a atributos.

Nós executamos o experimento em um *notebook* Dell Vostro core 2 duo 2.2GHz com o sistema operacional Ubuntu 9.04. A versão testada do Eclipse foi a 3.4.2.

Refatoramento	Geração				Resultados		
	Escopo	TPG	Tempo	PC	NP	EC	MC
Rename Method	3 – 0 – 3	5184	02:33	4752	3942	216	0
Rename Field	3 – 2 – 1	4896	08:25	3168	432	0	0
Push Down Method	3 – 0 – 4	11804	10:27	4926	1232	521	122
	3 – 0 – 3	3240	05:34	2592	648	216	48
	3 – 2 – 2	2304	03:48	1848	432	200	342
Push Down Field	3 – 2 – 1	6624	07:08	4800	2256	448	128
Pull Up Method	3 – 0 – 4	7488	13:18	6243	2914	114	152
	3 – 2 – 2	4032	03:22	2808	1728	0	0
Pull Up Field	3 – 2 – 2	5760	09:46	4320	1080	192	571
Encapsulate Field	2 – 2 – 2	1008	00:29	900	444	380	18
Move Method	3 – 1 – 3	3408	11:43	3002	1120	85	1646
Add Parameter	3 – 0 – 3	4944	14:22	4279	1793	0	628
Remove Parameter	3 – 0 – 3	7207	17:32	6706	858	0	228
Change Visibility	3 – 0 – 3	7920	22:13	6522	897	0	852

Tabela 5.5: Resumo do experimento para testar o módulo de refatoramento do Eclipse; Escopo = número de classes, atributos, e métodos; TPG = total de programas gerados pelo JDolly; Tempo = Tempo total da execução do teste em horas:minutos; PC = total de programas compiláveis gerados pelo JDolly; NP = número de refatoramentos não permitidos pelo Eclipse; EC = refatoramentos que introduziram erros de compilação; MC = refatoramentos que introduziram mudanças comportamentais

5.3.4 Resultados do Experimento

Nossa abordagem conseguiu detectar falhas de mudança comportamental em 9 das 11 implementações de refatoramentos testadas. Além disso, em 7 dos 11 refatoramentos testados, detectamos refatoramentos que introduzirem erros de compilação.

A Tabela 5.5 exibe os dados do experimento. Os dados referentes à geração dos programas pelo JDolly podem ser observados na coluna central (Geração). Ela é dividida em 4 sub-colunas. O JDolly pode gerar programas que não compilam, a Coluna TPG mostra o número total de programas gerados, enquanto que a Coluna PC exibe a quantidade desses programas que são compiláveis. A Coluna Tempo exibe o tempo total para testar cada refatoramento. A maior parte do tempo gasto ocorre na execução do SAFEREFACTOR. Ele gastou em média 3 segundos para avaliar cada refatoramento, com um *timeout* de 2 segundos para gerar os testes. Utilizamos o mesmo *timeout* do experimento descrito na Seção 5.1, visto que os programas gerados pelo JDolly possuem tamanho similar ao avaliados naquele

experimento. O processo de refatoração aplicado pelo Eclipse também demanda tempo. Em nossos experimentos, ele conseguiu refatorar aproximadamente 100 programas por segundo. Enquanto que o JDolly gerou os programas a uma taxa aproximada de 4.000 programas por segundo.

Podemos observar, na terceira Coluna Resultados (Tabela 5.5), os resultados de cada execução de nossa abordagem para testar refatoramentos. Quando uma pre-condição não é satisfeita por uma transformação, o Eclipse não a aplica, e exibe um aviso com a pre-condição que não foi satisfeita (Coluna NP). Porém, como sabemos, o Eclipse não checa todas as pre-condições necessárias, por isso ele pode aplicar refatoramentos incorretos. A Coluna EC exibe o número de refatoramentos que geraram erros de compilação no programa refatorado; esses erros são simples de serem detectados. Por outro lado, a Coluna MC exibe o número de transformações que não preservaram o comportamento do programa. A Tabela 5.6 mostra o percentual (com relação ao programas compiláveis) de erros (EC e MC) e refatoramentos não permitidos pelo Eclipse (NP). Note que, em alguns refatoramentos, o percentual de falhas é muito alto. Analisamos as falhas manualmente, e percebemos que um dos motivos para isso ocorrer é que o mesmo tipo de erro pode acontecer em programas com estruturas distintas. Por exemplo, uma mudança comportamental ocasionada por sobrescrita de método pode acontecer em métodos com parâmetros e sem parâmetros. No nosso caso, o JDolly gera os dois programas, mas as falhas são semelhantes.

Nós analisamos manualmente as transformações que não preservaram o comportamento, e conseguimos identificar 35 categorias de *bugs* distintas nessas transformações, sendo que 8 deles já tinham sido catalogados anteriormente [19; 70]. Porém, eles foram catalogados sem ajuda de um gerador de programas, o que exige um conhecimento apurado da semântica de Java. Além de analisar as mudanças de comportamento, categorizamos os erros de compilação com relação às mensagens de erros geradas pelo compilador. Obtivemos um total de 15 categorias de erros de compilação. A Tabela 5.7 exibe a quantidade de erros encontrada em cada refatoramento. A seguir, nós mostramos alguns exemplos de transformações que geram mudanças comportamentais ou introduzem erros de compilação e que ainda não haviam sido catalogadas [67].

Refatoramento	Escopo	NP (%)	EC (%)	MC (%)
Rename Method	3 – 0 – 3	82,95	4,5	0
Rename Field	3 – 2 – 1	13,6	0	0
Push Down Method	3 – 0 – 4	25	10,6	2,5
	3 – 0 – 3	25	8,3	1,9
	3 – 2 – 2	23,4	10,8	18,5
Push Down Field	3 – 2 – 1	47	9,3	2,7
Pull Up Method	3 – 0 – 4	46,7	1,82	2,4
	3 – 2 – 2	61,5	0	0
Pull Up Field	3 – 2 – 2	25	4,4	13,2
Encapsulate Field	2 – 2 – 2	49,3	42,2	2
Move Method	3 – 1 – 3	37,3	2,8	54,8
Add Parameter	3 – 0 – 3	41,9	0	14,7
Remove Parameter	3 – 0 – 3	12,8	0	3,4
Change Visibility	3 – 0 – 3	13,8	0	13,06

Tabela 5.6: Resultado dos testes nos refatoramentos; Escopo = número de classes, atributos, e métodos; NP = número de refatoramentos não permitidos pelo Eclipse; EC = refatoramentos que introduziram erros de compilação; MC = refatoramentos que introduziram mudanças comportamentais

Refatoramento	EC	MC
Rename Method	1	0
Rename Field	0	0
Push Down Method	2	9
Push Down Field	2	3
Pull Up Method	2	5
Pull Up Field	3	5
Encapsulate Field	2	1
Move Method	3	4
Add Parameter	0	3
Remove Parameter	0	2
Change Visibility	0	3
Total	**15**	**35**

Tabela 5.7: Quantidade de bugs distintos encontrados no Eclipse; EC = bugs de erros de compilação; MC = bugs de mudanças comportamentais

Refatoramento	Categorias de Erros de Compilação
Rename	Rename method reduz a visibilidade de um método herdado
Push Down Method	Push down method inviabiliza acesso através do qualified this
	Push Down Method altera a chamada de atributo para um atributo sem visibilidde
Push Down FIeld	Chamada ao atributo refatorado usando new instance não compila após o Push Down Field
	Chamada ao atributo refatorado usando super não compila após o Push Down Field
Pull Up Method	Pull Up Method inviabiliza acesso através do qualified this
	Chamada a método usando new instance não compila após o Pull Up Method
Pull Up Field	Pull Up Field torna atributo inacessível através do super
	Pull Up Field torna atributo inacessível através do new Instance
	Pull Up Field torna atributo inacessível através do this
Encapsulate Field	Encapsulate field gera um erro de compilação por causa do tipo de retorno do método sobrescrito
	Encapsulate Field reduz a visibilidade de um método herdado
Move Method	Move Method torna o método inacessível através do super porque o método não existe na classe
	Move Method torna o método inacessível através do super porque o método tem visibilidade private na classe
	Move method reduz a visibilidade de um método herdado

Tabela 5.8: Categoria de Erros de Compilação [67]

5.3.5 Categorias de *Bugs*

Nessa seção, descrevemos os *bugs* identificados através de nossa abordagem nos refatoramentos implementados pelo Eclipse 3.4.2. Primeiro, mostramos os *bugs* relacionados a erros de compilação, e em seguida, os relacionados a mudanças comportamentais.

Erros de Compilação

As categorias de erros de compilação identificadas são exibidas na Tabela 5.8. Entre os motivos para os erros de compilação, temos: um método que sobrescreve outro não pode reduzir a visibilidade deste; uma chamada de método não pode ser realizada por causa da visibilidade do método; o atributo chamado não é declarado na classe. A seguir, mostramos dois exemplos de erros de compilação identificados [67].

Rename Method. O Código Fonte 5.15 mostra um programa gerado pelo JDolly. Ele possui uma classe `A` e sua classe filha `B`. `A` declara o método `k(long)` e `B` declara o método `n(long)`. Ao aplicar o refatoramento *Rename Method* usando o Eclipse 3.4.2 para renomear o método `n(long) para k(long)`, esse método passa a sobrescrever o método declarado em `A` (Código Fonte 5.16). Porém, Java não permite sobrescrever um método reduzindo a visibilidade dele. Por isso, essa transformação produz um erro de compilação.

Figura 5.7: Rename Method reduz a visibilidade de um método herdado

Código Fonte 5.15: Versão Original

```
public class A {
  protected long k(long a){
    return 25;
  }
}
class B extends A {
  private long n(long a){
    return 28;
  }
  public long m0(){
    return new B().n(2);
  }
}
```

Código Fonte 5.16: Versão Refatorada

```
public class A {
  protected long k(long a){
    return 25;
  }
}
class B extends A {
  private long k(long a){
    return 28;
  }
  public long m0(){
    return new B().k(2);
  }
}
```

Pull Up Field. Considere o programa exibido no Código Fonte 5.17. A classe `C` estende `B` que, por sua vez, estende `A`. O método `m()` faz uma chamada ao atributo `f` de `B` usando o `super` (`B` herda `f` de `A`). Ao aplicar o refatoramento *Pull Up Field* usando o Eclipse 3.4.2 no atributo `f` declarado em `C`, a IDE move esse atributo para `B` (Código Fonte 5.18). Após essa transformação, o atributo `f` de `B` passa a ter visibilidade `private`, por isso, o método `m()` não consegue acessá-lo, gerando um erro de compilação.

Mudanças Comportamentais

Durante esse trabalho, apresentamos dois exemplos de mudanças comportamentais introduzidas por refatoramentos automatizados que foram detectadas automaticamente através da

Figura 5.8: Pull Up Field torna atributo inacessível através do super

Código Fonte 5.17: Versão Original

```
public class A {
  protected int f=20;
}
public class B extends A {
}
public class C extends B {
  private int f=43;
  public long m(){
    return super.f;
  }
}
```

Código Fonte 5.18: Versão Refatorada

```
public class A {
  protected int f=20;
}
public class B extends A {
  private int f=43;
}
public class C extends B {
  public long m(){
    return super.f;
  }
}
```

nossa abordagem para testar o módulo de refatoramento do Eclipse (Figuras 1.2 e 6.1). A Tabela 5.9 exibe todas as categorias de *bugs* de mudança comportamental que identificamos nesse experimento. A seguir, mostramos mais dois exemplos de refatoramentos incorretos identificados [67].

Move Method. A implementação do refatoramento *Move Method* no Eclipse só permite mover um método de uma classe para outra que seja declarada como atributo da primeira. O Código Fonte 5.19 exibe um programa com essas características gerado pelo JDolly. O método `test()` retorna o valor 10. Ao aplicarmos esse refatoramento ao método `k(int)`, esse método é movido da classe `A` para `B`. Além de movê-lo, o Eclipse atualiza o acesso dele no método `test()` (Código Fonte 5.20). Porém, quando executamos este último após a transformação, é lançada a exceção `NullPointerException`, pois o atributo `b` não é inicializado. Portanto, essa transformação não preserva o comportamento.

Pull Up Field. O refatoramento `Pull Up Field` é útil quando duas ou mais classes filhas possuem o mesmo atributo. Esse refatoramento move o atributo para a classe pai. O Código Fonte 5.21 mostra um programa gerado pelo JDolly com essas características. Porém, apesar dos atributos terem o mesmo nome (`k`), eles são inicializados com valores diferentes. Ao aplicar esse refatoramento em `B.k`, o Eclipse move esse atributo para `A`, e remove o atributo de `C` (Código Fonte 5.22). Depois dessa transformação, o método `test` retorna o valor de

Categoria de Erros de Mudança Comportamental	
Push Down Method	**Pull Up Method**
Mudança na chamada de métodos sobrescritos usando o super	Habilita sobrecarga + widening primitive conversion
Mudança na chamada de métodos sobrecarregados usando o super	Habilita sobrecarga + widening primitive conversion + visibility
desabilida sobrescrita + super	Habilita sobrescrita
Desabilita sobrecarga + widening primitive conversion	Alteração incorreta de this para super
Habilita sobrecarga + visibilidade + widening primitive conversion	Habilita método privado + visibilidade
Desabilita field hiding	**Pull Up Field**
Desabilita field hiding + primitive widening conversion	Habilita field hiding + super
Desabilita field hiding + super	Habilita field hiding + super + chamada indireta
Desabilita field hiding + super + primitive widening conversion	Habilita field hiding + super + widening primitive conversion
Push Down Field	Altera o valor de um atributo da classe filha
Desabilita field hiding + super	Move atributos de níveis diferentes da hierarquia e altera o valor
Desabilita + super + chamada indireta	**Remove Parameter**
Desabilita field hiding + super + widening primitive conversion	Habilita sobrescrita
Change Visibility	Altera chamada de método e habilita sobrescrita
Habilita sobrescrita	**Add Parameter**
Habilita sobrecarga	Habilita sobrescrita
Desabilita sobrecarga	Altera chamada de método e habilita sobrescrita
Move Method	Habilita sobrecarga
Habilita sobrecarga	**Encapsulate Field**
Habilita sobrescrita	Habilita sobrescrita
Desabilita sobrecarga + super	
Causa NullPointerException	

Tabela 5.9: Categoria de *Bugs* de Mudança Comportamental [67]

Figura 5.9: Move Method pode causar NullPointerException

Código Fonte 5.19: Versão Original

```
public class A {
  public B b;
  protected long k(int a){
    return 10;
  }
  protected long k(long a){
    return 20;
  }
  public long test(){
    return k(2);
  }
}
public class B {
}
```

Código Fonte 5.20: Versão Refatorada

```
public class A {
  public B b;
  protected long k(long a){
    return 20;
  }
  public long test(){
    return b.k(2);
  }
}
public class B {
  protected long k(int a){
    return 10;
  }
}
```

10, ao invés de 20, valor retornado no programa original.

5.3.6 Discussão

A abordagem utilizada mostrou-se útil para testar a ferramenta de refatoramento do Eclipse. Ela conseguiu detectar mudanças comportamentais em todos os refatoramentos, com exceção do `Rename`. Conseguimos identificar 50 categorias de *bugs* distintas nos refatoramentos implementados pelo Eclipse, sendo 35 com relação a mudanças comportamentais, e 15 relacionadas a erros de compilação. Adicionalmente, testamos manualmente o Netbeans 6.7.1 com relação aos *bugs* de mudança comportamental e identificamos que 22 deles também ocorrem nessa IDE (os *bugs* dos refatoramentos *Move Method* e *Chance Visibility* não foram testados, pois o Netbeans não automatiza esses refatoramentos).

Um problema chave na geração de programas para testes é conseguir escalá-la de forma a produzir programas significativos para os testes [12]. A geração de programas do JDolly segue a abordagem *bounded exhaustive testing* [12]. Ela gera todos os possíveis programas para um determinado escopo de complexidade ou tamanho. Dessa forma, conseguimos testar

Figura 5.10: Pull Up Field pode alterar o valor de um dos atributos

Código Fonte 5.21: Versão Original

```
public class A {
}
public class B extends A {
  public int k = 10;
}
public class C extends A {
  public int k = 20;
  public long test() {
    return k;
  }
}
```

Código Fonte 5.22: Versão Refatorada

```
public class A {
  public int k = 10;
}
public class B extends A {
}
public class C extends A {
  public long test() {
    return k;
  }

}
```

todos os *corner cases* desse escopo, incluindo aqueles que a geração puramente randômica de programas pode perder [14]. Essa abordagem fundamenta-se na hipótese de que, na prática, uma falha que ocorre em um programa de tamanho grande, ocorre também em programas menores [14; 36]. O JDolly mostrou-se útil para gerar programas para testar os refatoramentos avaliados. Ele conseguiu gerar programas aleatórios interessantes para detectar tanto falhas de mudança comportamental, quanto erros de compilação. A abordagem declarativa permitiu especificar de forma simples as características estruturais que o programa deveria ter para ser refatorado. Porém, outros refatoramentos não avaliados são aplicados na parte comportamental do programa. Por exemplo, o *Extract Method* e o *Rename Variable* são aplicados no corpo do método. A versão atual do JDolly possui pouca expressividade para gerar a parte comportamental dos programas. Com relação aos métodos, ela só é capaz de gerar instruções de retornos. Por isso, não testamos esses refatoramentos, mas pretendemos testá-los em trabalhos futuros.

Apesar de não termos identificados mudanças comportamentais na implementação do Eclipse do refatoramento *Rename*, vimos no primeiro experimento (Seção 5.1) que nossa técnica consegue detectar mudanças comportamentais catalogadas na literatura [19] nesse refatoramento. Porém, nosso meta-modelo atual de Java em Alloy não possui expressividade suficiente para detectarmos os problemas previamente encontrados [19] nesse refatoramento,

e em alguns outros [70]. Para o JDolly gerar programas capazes de expor esses *bugs* conhecidos, é necessário incluir no meta-modelo os elementos Java: *package*, *import*, *static*, *inner class*.

Além disso, podemos observar na Tabela 5.6 que quase 83% das transformações aplicadas pelo refatoramento *Rename Refactoring* não satisfizeram as pre-condições implementadas pelo Eclipse. Para identificar porque tantas transformações não foram realizadas, analisamos essas pre-condições. A sobrecarga de métodos com o mesmo número de parâmetros é uma possível fonte de mudanças comportamentais em refatoramentos. Acreditamos que por esse motivo, os desenvolvedores do Eclipse preferiram implementar como pre-condição que não pode haver sobrecarga com o mesmo número de parâmetros no método a ser refatorado. Apesar dessa pre-condição evitar que muitos erros ocorram, ela impede a realização de muitos refatoramentos válidos. Nossa técnica pode ser útil para detectar os casos que realmente causam mudanças comportamentais, ajudando na especificação de um conjunto menor de pre-condições que garantam a preservação do comportamento.

Daniel et al. [14] propuseram um abordagem para testar ferramentas de refatoramentos utilizando o ASTGen. Porém, enquanto nosso foco foi identificar falhas de mudança comportamental utilizando nossa técnica, a maioria dos *bugs* identificados por Daniel et al. [14] foram relacionados a erros de compilação. No total, eles identificaram 21 *bugs* no Eclipse 3.2, sendo 17 destes, erros de compilação, e apenas um erro de comportamento. Os outros 3 *bugs* são relacionados com transformações que não foram completamente realizadas, mas que não alteraram o comportamento ou geraram erros de compilação. Nós também identificamos erros de compilação com o JDolly. A Tabela 5.10 exibe a quantidade de *bugs* identificados usando o JDolly e o ASTGen para cada refatoramento. Identificamos erros de compilação em todos refatoramentos que Daniel et al. [14] identificou com o ASTGen, exceto o *Member to Top*, pois não o testamos. Por outro lado, eles testaram e não encontraram nenhum erro no *Rename Method*, enquanto nós identificamos dois tipos de *bugs* com o JDolly. Com relação aos *bugs* de mudança comportamental, o ASTGen identificou apenas um na implementação do refatoramento *Encapsulate Field*.

Corrigir esses *bugs* encontrados é um trabalho difícil. Note que Daniel et al. [14] detectou 21 problemas na versão 3.2.2 do Eclipse. Nós avaliamos o Eclipse 3.4.2 e identificamos 50 problemas (35 *bugs* de mudança de comportamento e 15 *bugs* de erros de compilação).

Refatoramento	EC		MC	
	JDolly	ASTGen	JDolly	ASTGen
Rename Method	2	0	0	0
Rename Field	0	0	0	0
Push Down Method	2	2	9	0
Push Down Field	1	2	3	0
Pull Up Method	3	2	5	0
Pull Up Field	1	3	5	0
Encapsulate Field	2	7	1	1
Move Method	4	-	4	-
Add Parameter	0	-	3	-
Remove Parameter	0	0	2	0
Change Visibility	0	-	3	-
Change Return Type	-	0	-	0
Member to Top	-	1	-	0

Tabela 5.10: Comparação entre os resultados do JDolly e ASTGen; EC = refatoramentos que introduziram erros de compilação; MC = refatoramentos que introduziram mudanças comportamentais

Ekman et al [19], e Steimann e Thies [70] identificaram manualmente outros *bugs* nos refatoramentos do Eclipse e Netbeans. Nesse trabalho, propomos uma técnica e uma ferramenta (SAFEREFACTOR) que pode ser utilizada tanto por desenvolvedores de ferramentas de refatoramento, ajudando a identificar *bugs*, quanto por desenvolvedores que precisam refatorar seus programas, aumentando a confiança de que as transformações estão corretas.

Capítulo 6

SAFEREFACTOR

Nesse capítulo, apresentamos o SAFEREFACTOR, uma ferramenta para detectar mudanças comportamentais em refatoramentos de programas Java sequenciais. Ele pode ser utilizado através de um plugin para o Eclipse ou através de sua interface de linha de comando. Na Seção 6.1, descrevemos o Eclipse plugin que utiliza o SAFEREFACTOR para checar a segurança dos refatoramentos automatizados por essa IDE. Na Seção 6.2, descrevemos como utilizar o SAFEREFACTOR através de sua interface de linha de comando. Essa interface permite checar se um refatoramento (manual ou automatizado) introduz mudanças comportamentais no programa.

6.1 Eclipse Plugin

Vimos que os refatoramentos implementados pelo Eclipse podem realizar transformações que alteraram o comportamento do programa. Desenvolvemos o SAFEREFACTOR para permitir que os desenvolvedores possam utilizar nossa técnica para aumentar a confiança nas transformações realizadas por essa IDE. Nós disponibilizamos esse plugin para *download* através de seu *site* [1]. Ele é distribuído sob a licença *Eclipse plugin License (EPL)*. No *site*, incluímos também um vídeo demonstrando como utilizá-lo e instruções de como instalá-lo.

Nessa seção, descrevemos as funcionalidades dessa ferramenta e como utilizá-lo (Seção 6.1.1). Além disso, mostramos uma visão geral de sua arquitetura (Seção 6.1.2).

[1] Site do SafeRefactor: http://dsc.ufcg.edu.br/~spg/saferefactor

6.1.1 Exemplo

Nessa seção, mostramos como utilizar o SAFEREFACTOR na prática. O exemplo a seguir mostra uma transformação (Figura 6.1) aplicada pelo Eclipse 3.4, mas que não preserva o comportamento do programa. Essa transformação foi automaticamente detectada pela nossa abordagem para testar ferramentas de refatoramentos (Capítulo 4).

Figura 6.1: Pull Up Method altera a expressão de chamada de método e causa uma mudança comportamental

Código Fonte 6.1: Versão Original

```
public class A {
  public int k() {
    return 10;
  }
}
public class B extends A {
  public int k() {
    return 20;
  }
  public int m() {
    return super.k();
  }
  public int test() {
    return m();
  }
}
```

Código Fonte 6.2: Versão Refatorada

```
public class A {
  public int k() {
    return 10;
  }
  public int m() {
    return this.k();
  }
}
public class B extends A {
  public int k() {
    return 20;
  }
  public int test() {
    return m();
  }
}
```

O programa original é exibido no Código Fonte 6.1. O método `k()` é declarado em `A` e sobrescrito por `B`. O método `test()` retorna o valor 10. Ele faz uma chamada a `m()` que por sua vez chama o método `k()` da classe `A`.

Suponha que desejamos aplicar o refatoramento *Pull Up Method* em `m()`. Os passos iniciais são: (1) seleciona o método `m()` no editor Java; (2) acessa o menu de refatoramentos e seleciona *Pull Up* (Figura 6.2); (3) o Eclipse abre uma janela para configurar o refatoramento.

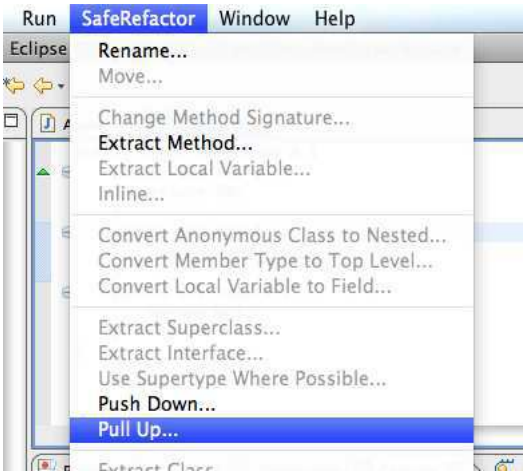

Figura 6.2: Menu de seleção dos refatoramentos

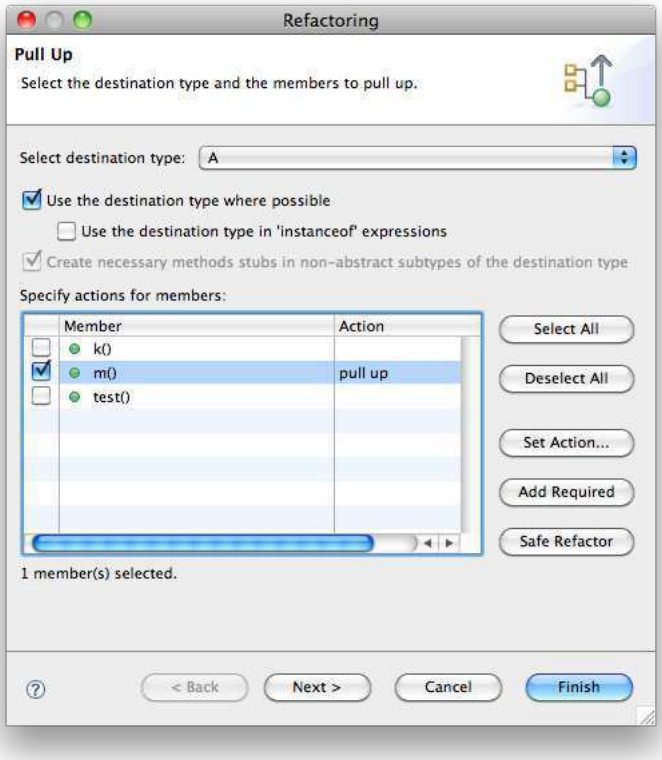

Figura 6.3: Tela com o botão que habilita a checagem do SAFEREFACTOR

Após configurar o refatoramento (escolha do método e da classe de destino), o Eclipse permite visualizar uma prévia da transformação, ou aplicá-lo. Adicionalmente, o SAFEREFACTOR permite avaliar a segurança da transformação. Para isso, o desenvolvedor aciona o botão "Safe Refactor", ilustrado na Figura 6.3. Outra janela é aberta (Figura 6.4) e o plugin

inicia as etapas da nossa técnica para checar a segurança do refatoramento.

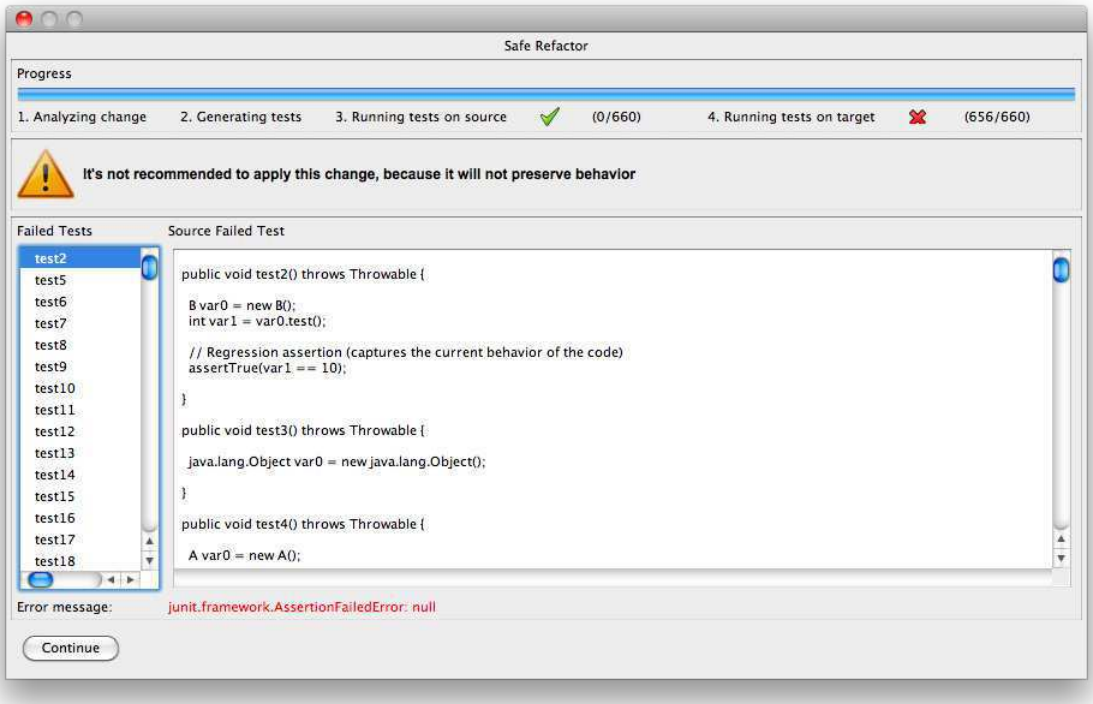

Figura 6.4: Tela principal do SAFEREFACTOR

Em um notebook Vostro Dual Core 2,2 GHz com 2GB de RAM, o processo leva 8 segundos para ser concluído, e gera 660 testes de unidade para avaliar a transformação da Figura 6.1. O número de testes gerados é proporcional ao tempo alocado para a geração. O tempo padrão é de 2 segundos, podendo ser alterado nas configurações do plugin (Figura 6.5). É recomendável aumentar o tempo para programas maiores, pois quanto mais testes, maior as chances de exercitar todas as mudanças realizadas.

A janela do SAFEREFACTOR ilustrado na Figura 6.4 mostra que 656 dos 660 testes falharam no programa refatorado, portanto, a transformação não preserva o comportamento do programa original. Uma mensagem é exibida na janela avisando ao desenvolvedor que não é recomendável aplicar o refatoramento. Além disso, o SAFEREFACTOR exibe a lista de testes que falharam; isso ajuda o desenvolvedor a identificar em que parte do programa ocorreu a mudança comportamental. Podemos observar na Figura 6.4 um teste de unidade `test2()` que revela a mudança comportamental: o método `B.test()` retorna 10 no programa original, mas após o transformação ele retorna 20. O Código Fonte 6.2 exibe a transformação que

Figura 6.5: Tela de configuração SAFEREFACTOR permite alterar o tempo para geração de testes

seria realizada pelo Eclipse sem o SAFEREFACTOR. Além de mover o método `m()` para a classe `A`, a IDE altera a expressão no corpo desse método para evitar um erro de compilação. Ela substitui o `super` pelo `this` na chamada de método a `k()`. Porém, enquanto que o `super` é calculado em tempo de compilação, o `this` é calculado em tempo de execução. Ao executar o método `test()`, `this` refere-se a `B`; então, `m()` chama `k()` declarado em `B`, ao invés de `k()` declarado em `A`, como no programa original.

6.1.2 Arquitetura

O Eclipse possui uma arquitetura extensível que permite desenvolver plugins para adicionar novas funcionalidades a essa IDE [8]. O SAFEREFACTOR foi desenvolvido com base nessa arquitetura de plugins.

O SAFEREFACTOR contém dois módulos principais: `Core` e `GUI` (Figure 6.6). O primeiro é responsável por implementar o processo da nossa técnica. Enquanto que o segundo é responsável por integrar a implementação da técnica à interface gráfica utilizada para aplicar os refatoramentos do Eclipse. A Figura 6.6 ilustra a arquitetura do nosso plugin. A seguir, detalhamos esses dois módulos.

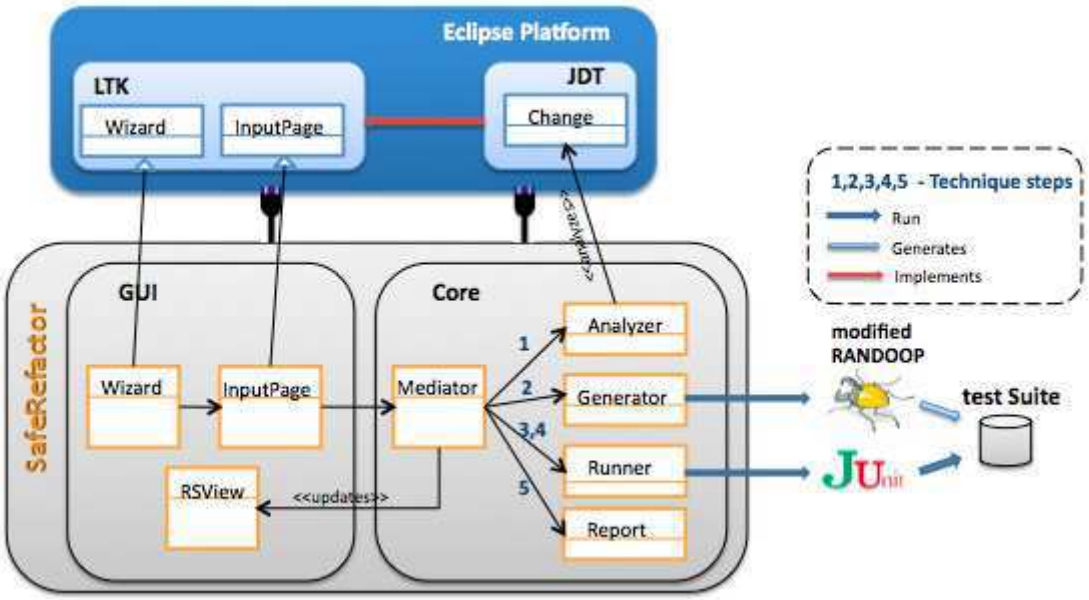

Figura 6.6: Arquitetura do SAFEREFACTOR

Módulo Core

Nossa técnica para detectar mudanças comportamentais possui cinco etapas, como mostramos na Seção 3.1. A primeira etapa consiste em identificar os métodos comuns às versões original e refatorada do programa. Ela é implementada na classe `Analyzer` (Figura 6.6). Essa classe utiliza a API Eclipse Core [2] para gerar uma AST do programa original. Ela identifica todos os métodos do programa. O Eclipse possui uma API de automatização de refatoramentos LTK [77] que foi implementada no módulo JDT para criar a ferramenta de refatoração dessa IDE. Essa API contém a classe `Change` que armazena as informações sobre as modificações que serão realizadas no código pelo refatoramento. A classe `Analyzer` acessa essas informações para checar quais métodos permanecerão em comum entre as versões original e refatorada do programa. A classe `Generator` realiza a segunda etapa da nossa técnica. Ela gera uma coleção de testes utilizando nossa versão do Randoop para os métodos identificados pela classe `Analyzer`. As Etapas 3 e 4 são realizadas pela classe `Runner`. Ela utiliza a API do Junit para executar os testes na versão original e refatorada do programa. Por fim, a classe `Report` analisa os resultados dos testes e reporta se houve

[2] Java Model Tutorial: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_model.htm

ou não mudança comportamental. Todo o processo da técnica é modelado utilizando o padrão de projeto *Mediator* [25]. A classe `Mediator` é responsável por invocar cada uma das classes descritas acima para concluir o processo, e atualizar a interface gráfica com os resultados.

Módulo GUI

A atividade de refatoramento no Eclipse é realizada através da interface gráfica dessa IDE, como vimos na Seção 2.2.2. Cada refatoramento possui um conjunto de janelas conectadas (*wizard*) que guia o desenvolvedor na aplicação do refatoramento. Para implementar os *wizards*, o Eclipse estende a classe `RefactoringWizard` da API LTK. A janela inicial possui os controles para aplicar ou pré-visualizar o refatoramento, e é implementada através da classe `UserInputPage`. Essas duas classes foram implementadas pelo Eclipse de forma a não permitir sua extensão por outros desenvolvedores. Por outro lado, o manuseio dos refatoramentos pode ser realizado programaticamente através de sua API.

Para habilita nossa técnica, criamos novos *wizards* para cada refatoramendo, adicionando a opção de checar a segurança do refatoramento usando o SAFEREFACTOR. Note que foi necessário re-implementar apenas a interface gráfica dos refatoramentos. Estas aplicam os refatoramentos implementados pelo Eclipse. O módulo GUI contém um par de classes filhas de `RefactoringWizard` e `UserInputPage` para cada refatoramento.

Refatoramentos Implementados

A implementação atual do SAFEREFACTOR habilita nossa técnica para 6 refatoramentos do Eclipse, o que corresponde a uma cobertura de aproximadamente 20% dos refatoramentos implementados por essa IDE (Tabela 2.2). Os refatoramentos implementados são:

- *Rename Method, Field, Variable*;
- *Extract Method.*
- *Pull Up Method*;
- *Push Down Method*;

Decidimos implementar esses refatoramentos de início baseado na pesquisa conduzida por Murphy et al. [51] que avalia os refatoramentos mais utilizados no Eclipse. O *Rename*, *Extract Method*, e *Pull Up* estão entre os quatro tipos de refatoramentos mais utilizados segundo essa pesquisa. Além disso, foram catalogadas anteriormente [19] mudanças comportamentais devido ao refatoramento `Push Down Method`, o que nos motivou a habilitar nossa técnica para esse refatoramento.

Adição de Refatoramentos

A implementação atual de nossa ferramenta permite utilizar nossa técnica para alguns refatoramentos usados pelos usuários do Eclipse. Como podemos ver na sua arquitetura ilustrada na Figura 6.6, a técnica é desacoplada dos refatoramentos do Eclipse. Dessa forma, desenvolvedores podem estender nosso plugin adicionando novos refatoramentos. Para adicionar um novo refatoramento ao SAFEREFACTOR é necessário seguir os passos:

1. criar uma classe que estende a classe `UserInputPage` da API LTK. Essa classe é responsável por criar os controles para configuração e finalização do refatoramento. É nessa classe que adicionamos o botão do SAFEREFACTOR;

2. criar uma classe que estende a classe `RefactoringWizard` da API LTK. Essa classe contém as janelas que formam o *wizard* do refatoramento, como a janela com os controles para configurar o refatoramento e a janela com o *preview* dele;

3. criar uma classe que implementa a interface `IWorkbenchWindowActionDelegate` da API Eclipse UI. Essa classe é responsável por disparar a ação que inicia o *wizard* do refatoramento;

4. adicionar a entrada do novo refatoramento no arquivo plugin.xml.

As classes criadas a partir de `RefactoringWizard` e `IWorkbenchWindowActionDelegate` possuem em média 25 e 100 linhas de código, respectivamente. A classe que estende `UserInputPage` é maior, pois é nela que criamos os controles da interface gráfica do refatoramento. Seu tamanho depende do número de componentes gráficos implementados. A implementação dessa classe similar à

classe original do refatoramento *Pull Up Method* com a adição do botão para habilitar o SAFEREFACTOR possui 865 linhas de código. O trecho de código exibido no Código Fonte 6.3 mostra os comandos que é necessário adicionar no método `createButtonComposite` dessa classe para habilitar nossa técnica para esse refatoramento. Adicionamos um botão na interface gráfica com uma ação que chama o método `Mediator.checkSafety` passando como parâmetro o refatoramento que será realizado. Esse método abre a janela principal do SAFEREFACTOR (Figura 6.4) e inicia o processo para checar se o refatoramento produz uma mudança comportamental no código. Widmer [77] descreve detalhadamente como criar a interface gráfica dos refatoramentos.

Código Fonte 6.3: Comandos necessários para adicionar nossa técnica a janela de refatoramento

```
final Composite composite = new Composite(parent, SWT.NONE);
final Button safeButton = new Button(composite, SWT.PUSH);
safeButton.setText(Constants.TOOL_NAME);
safeButton.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
SWTUtil.setButtonDimensionHint(safeRefactorButton);
checkSafe.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(final SelectionEvent event) {
    initializeRefactoring();
    Refactoring ref = getRefactoring();
    Mediator.checkSafety(ref,composite.getShell());
  }
});
```

6.2 Interface de Linha de Comando

A distribuição do SAFEREFACTOR fornece uma interface de linha de comando. Essa interface permite utilizar nossa técnica para checar a ocorrência de mudanças comportamentais em refatoramentos manuais ou automatizados de programas Java sequenciais. Ela foi utilizada nos experimentos descritos no Capítulo 5. A seguir, descrevemos como suas funcionalidades e como utilizá-las.

6.2.1 Funcionalidades

Nessa seção, mostramos as funcionalidades que o SAFEREFACTOR fornece através da interface de linha de comando. Elas podem ser acessadas através dos métodos da classe `saferefactor.CommandLine`. A implementação atual dessa classe fornece dois métodos: `isRefactoring` e `checkRefactoring` (Código Fonte 6.4).

Código Fonte 6.4: Métodos do SAFEREFACTOR

```
public static boolean isRefactoring(String original,String refactored,
    int timeout)
public static Report checkRefactoring(String original,String refactored
    ,int timeout)
```

Os métodos recebem como parâmetro o caminho para o diretório do programa original, o caminho para o diretório do programa refatorado, e o tempo para a geração dos testes. O primeiro método (`isRefactoring`) retorna `true` se não for detectada nenhuma mudança comportamental entre as duas versões do sistema. Enquanto que o segundo, `checkRefactoring`, retorna um objeto do tipo `Report`. Esse objeto contém os dados do resultado de nossa técnica: número de testes gerados; testes que passaram no programa original e falharam no programa refatorado; tempo total; se foi detectada uma mudança comportamental.

6.2.2 Exemplo

A seguir, mostramos como utilizamos o SAFEREFACTOR para detectar uma mudança comportamental no refatoramento aplicado ao JHotDraw (Seção 5.2). O método `main` exibido no Código Fonte 6.5 mostra os comandos para executar a nossa técnica utilizando o SAFEREFACTOR. Esse programa recebe como argumento o caminho para o programa original, o caminho para o programa refatorado, e o tempo para geração de testes. Ele imprime os resultados armazenados no objeto `report` na saída padrão de Java. A Figura 6.7 exibe o resultado da execução desse programa para o refatoramento aplicado ao JHotDraw, com um tempo de 60 segundos para geração de testes. Esse refatoramento não preservou o comportamento do programas, pois 273 dos 2245 testes gerados passaram na versão original, mas falharam na versão refatorada. Na Seção 5.2.3, descrevemos essa mudança comportamental

encontrada no JHotDraw.

Código Fonte 6.5: Métodos do SAFEREFACTOR

```
public static void main(String[] args) {
  String original = args[0];
  String refactored = args[1];
  int timeout = args[2];
  Report report = new Report();
  report = SRCommandLine.checkRefactoring(original, refactored, timeout);
  System.out.println("Tempo Total: " + report.getTotalTime());
  System.out.println("Número total de testes gerados: " + report.
      getTotalTests());
  System.out.println("Número de testes que falharam: " + report.
      getFailedTests());
  System.out.println("É um refatoramento? " + report.isRefactoring());
}
```

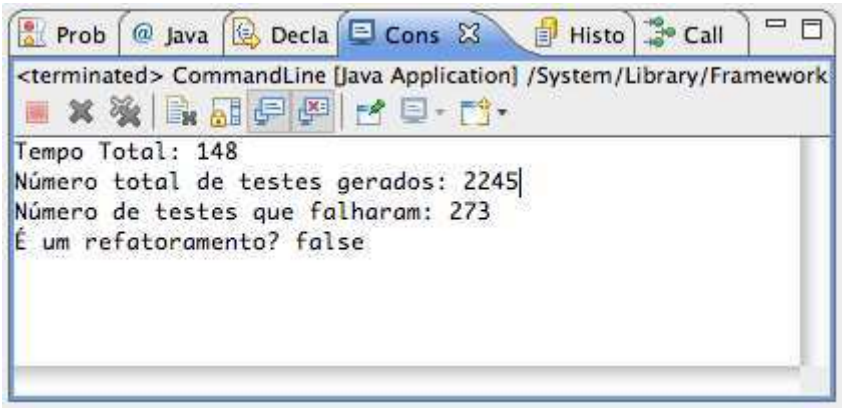

Figura 6.7: Resultado da avaliação SAFEREFACTOR para o refatoramento aplicado ao JHotDraw

Capítulo 7

Conclusões

Nesse trabalho, nós propomos uma abordagem para aumentar a segurança em refatoramentos de programas sequenciais orientados a objetos. Atualmente, as IDEs que automatizam refatoramentos, tais como Eclipse, Netbeans, IntelliJ, ajudam a eliminar vários erros causados pela aplicação manual do refatoramento. Entretanto, elas podem realizar transformações que introduzem erros de compilação ou mudanças comportamentais nos programas. Nossa técnica detecta automaticamente mudanças comportamentais em refatoramentos de programas sequenciais orientados a objetos. Primeiro, ela realiza uma análise estática no programa original e no refatorado para identificar os métodos comuns às duas versões. Em seguida, ela gera uma coleção de testes de unidade para esses métodos. Executamos a mesma coleção de testes no programa antes e depois de ser refatorado. Se algum teste passar em uma versão e falhar na outra, nós caracterizamos uma mudança comportamental. Caso contrário, aumentamos confiança de que a transformação está correta. Implementamos nossa técnica em uma ferramenta chamada SAFEREFACTOR.

Propomos também um gerador de programas Java (JDolly) utilizando a linguagem de especificação formal Alloy [36], e o ASTGen [14], um *framework* para geração de ASTs Java. O JDolly pode ser utilizado para testar ferramentas que recebem programas como entrada, como por exemplo, ferramentas de refatoramento. O usuário especifica o número máximo de classes, métodos, e atributos que os programas devem possuir. Além disso, ele pode especificar restrições descrevendo as características estruturais desejadas para os programas. O JDolly gera automaticamente um número de programas.

Avaliamos nossa técnica em três experimentos. Primeiro, ela foi utilizada em um con-

junto de 16 transformações aplicadas por IDEs (Eclipse, Netbeans, JBuilder, IntelliJ) que alteram o comportamento do programa. Nossa técnica detectou todas as mudanças comportamentais. Em seguida, utilizamos nossa técnica para checar a segurança de refatoramentos aplicados em sete programas reais (3-100 KLOC). Os desenvolvedores que implementaram essas transformações acreditavam que elas eram corretas. Nossa técnica levou aproximadamente dois minutos para detectar uma mudança comportamental em um refatoramento aplicado ao JHotDraw (23 KLOC). Além disso, dois refatoramentos aplicados com o Eclipse introduziram erros de compilação no código. Por último, utilizamos o SAFEREFACTOR e o JDolly para testar 11 implementações de refatoramentos do Eclipse 3.4. Para cada implementação, utilizamos o JDolly para gerar um número de programas com características específicas que permitisse o refatoramento. Por exemplo, o JDolly gerou 5.760 programas, cada um contendo pelo menos uma classe pai e uma classe filha com um atributo, para testar a implementação do *Pull Up Field*. Aplicamos o refatoramento nesses programas utilizando a API do Eclipse, e ao final, utilizamos o SAFEREFACTOR para checar a ocorrência de erros de compilação e mudanças comportamentais. Como resultado, identificamos 50 *bugs* distintos, dos quais 35 produzem mudanças comportamentais nos programas refatorados, e 15 introduzem erros de compilação.

Nossa abordagem não garante a corretude do refatoramento. Como foi dito por Dijkstra [16], testes podem revelar a presença de erros, mas não sua ausência. O ideal seria que as pre-condições de cada refatoramento fossem especificadas formalmente de acordo com a semântica formal da linguagem. Porém, linguagens complexas, como Java e C#, não possuem uma semântica formal definida para todas as suas construções. Provar refatoramentos para essas linguagens ainda é considerado um desafio [62]. Nesse trabalho, propomos uma abordagem mais prática que foca em detectar automaticamente mudanças comportamentais.

Nossa técnica não pode comparar diretamente o comportamento de um método que tem sua assinatura alterada por um refatoramento, tal qual: *Rename Method*, *Push Down Method*, *Change Method Signature*. Porém, podemos checar esse comportamento indiretamente através dos métodos comuns às versões original e refatorada do programa. Nossa noção de equivalência de programas é baseada baseada na noção de equivalência proposta por Opdyke [54]. Ele compara o comportamento de dois programas com relação a um método `main` (comum às duas versões). Se esse método for chamado duas vezes com as mesmas

entradas, as saídas devem ser as mesmas. Nossa técnica avalia o comportamento observável dos programas com relação a uma coleção de testes de unidade. Cada teste contém uma seqüência, gerada aleatoriamente, de chamadas de métodos e construtores. Essa seqüência só contém métodos comuns às duas versões do programa (como o `main` de Opdyke). Se o resultado desses métodos no programa original e no refatorado forem diferentes para as mesmas entradas, esses programas possuem comportamentos diferentes.

É importante ressaltar que nossa abordagem pode ser utilizada de forma complementar à abordagem tradicional de refatoramentos (Seção 2.2). Os desenvolvedores podem continuar usando suas coleções de testes, e utilizar nossa técnica para aumentar a confiança de que a transformação está correta.

Com relação a abordagem proposta para testar ferramentas de refatoramento, é importante mencionar que módulo de refatoramento do Eclipse possui mais de 2.400 testes de unidade [14]. Esses testes não foram capazes de revelar os *bugs* que identificamos. Além disso, a abordagem de testes de ferramentas de refatoramento proposta por Daniel et al. [14] foca em detectar erros de compilação. Nossa abordagem conseguiu detectar tanto erros de compilação quanto erros de mudança comportamental. Ela pode ser utilizada para ajudar no desenvolvimento de ferramentas mais confiáveis.

O JDolly gerou programas variados com no máximo três classes, quatro métodos com apenas uma instrução, e dois atributos. Esses programas foram úteis para detectar 50 *bugs* distintos no Eclipse. Nossa abordagem fortalece a hipótese proposta por Jackson [36] de que, na prática, uma falha que ocorre em um programa de tamanho grande, ocorre também em programas menores [14; 36].

7.1 Trabalhos Relacionados

Nosso trabalho é relacionado com trabalhos que propõem soluções formais para garantir a corretude dos refatoramentos, e trabalhos sobre geração automática de entradas para testes e testes em ferramentas de refatoramento. A seguir, relacionamos essas abordagens com a nossa.

7.1.1 Preservação do Comportamento

Ekman et al. [19] catalogaram manualmente alguns refatoramentos problemáticos. Eles propõem avaliar a corretude de cada refatoramento baseado em invariantes específicos. Por exemplo, para o *Rename*, eles especificam os invariantes: apenas nomes devem ser afetados pelo refatoramento; cada nome refere-se à mesma entidade antes e depois da transformação. Como solução, eles propõem uma solução para garantir que após o refatoramento, todos os nomes no programa estejam vinculados estaticamente as entidades de antes [61]. Eles também propõem uma solução similar para o *Extract Method* [63]. Essa abordagem tem a vantagem de não precisar formalizar toda a linguagem, apenas a parte necessária para analisar se o invariante especificado é preservado. Ela é complementar à nossa, visto que ela aumenta a segurança de um refatoramento específico, enquanto que nós propomos uma solução que pode ser usada em todos os refatoramentos. Além disso, podemos utilizar nossa abordagem de testes de ferramentas de refatoramento para testar essa implementação do *Rename* proposta por eles.

Em Java, a alteração dos modificadores de visibilidade (`public`, `protected`, `package`, `private`) pode alterar o *binding* estático e dinâmico das entidades, alterando o comportamento do programa. Steimann e Thies [70] mostram mudanças comportamentais relacionadas à visibilidade de classes e métodos em transformações aplicadas por ferramentas de refatoramento. Eles formalizam a visibilidade de Java, e propõem restrições relacionadas à visibilidade para garantir a preservação do *binding* estático e dinâmico das entidades. Eles não provaram formalmente essas restrições, mas implementaram uma ferramenta para testar os refatoramentos com as restrições propostas. Essa ferramenta automatiza os refatoramentos, checa se o programa refatorado compila, e se passa nos testes originais. A abordagem proposta nesse trabalho pode ajudar a aprimorar as ferramentas de refatoramento com relação às restrições de visibilidade. Nosso diferencial é detectar automaticamente mudanças comportamentais independente do refatoramento e da causa da mudança.

Em nosso experimento exibido na Seção 5.1, utilizamos os *bugs* de mudanças comportamentais catalogados por Ekman et al. [19] e Steimann e Thies [70]. O SAFEREFACTOR não detecta mudanças comportamentais relacionadas com valores exibidos na saída padrão de Java (`System.out.println`). Por isso, incluímos nos exemplos catalogados por eles instruções de retorno nos métodos com os valores exibidos na tela. Preten-

demos incluir no Randoop algum padrão de testes que seja útil para detectar mudanças no `System.out.println`. Nossa técnica detectou todos os *bugs* catalogados. Por outro lado, no experimento descrito na Seção 5.3, o JDolly não conseguiu gerar programas que revelassem alguns desses *bugs* previamente catalogados. Pretendemos incluir outras construções de Java, como *package*, *import*, *static*, e *inner class*, no meta-modelo Java do JDolly para gerar programas com expressividade suficiente para revelar esses *bugs* e outros novos *bugs*.

Borba et al. [4; 5; 13] propõem um conjunto de refatoramentos para a linguagem Refinement Object-Oriented Language (ROOL), um subconjunto de Java seqüencial com semântica de cópia. Eles provaram a corretude dos refatoramentos com relação a uma semântica formal. Esse trabalho pode ajudar a identificar pre-condições para refatoramentos em Java. Porém, provar refatoramentos considerando toda a linguagem ainda é considerado um desafio [62]. Além disso, o número de refatoramentos é extenso. Por exemplo, no catálogo de Fowler [59] existem mais de 90 refatoramentos. Formalizar todos os refatoramentos é custoso, e uma evolução da linguagem pode obrigar a refazer as provas. Nosso trabalho é complementar a esse. Como mencionamos antes, as pré-condições de cada refatoramento não são formalmente definidas. Nosso trabalho propõe uma maneira simples e prática de detectar mudanças comportamentais.

Silva et al. [64] propõem um conjunto de leis para linguagens sequenciais orientadas a objetos levando em consideração semântica de referência. Eles provaram a corretude de cada uma das leis com relação a semântica de um linguagem que possui semântica de referência chamada rCOS. Essas leis podem ser utilizadas no contexto de Java, visto que essa linguagem possui semântica de referência. Porém, eles não consideram algumas funcionalidades da linguagem, como sobrecarga de método e *field hiding*. Como vimos nesse trabalho, nossa abordagem detectou refatoramentos automatizados que alteraram o comportamento de programas que utilizam essas funcionalidades. Nosso trabalho é complementar ao de Silva et al. [64], pois nossa técnica consegue detectar mudanças comportamentais não abordadas no trabalho deles.

Duarte [17] estendeu as leis de transformação de ROOL para a linguagem Java, considerando boa parte de sua estrutura e paralelismo. Porém, ele não prova essas leis com relação a uma semântica formal; por isso, elas podem conter erros. Na verdade, inspecionamos ma-

nualmente algumas destas leis e verificamos que elas podem alterar o comportamento do programa. Por exemplo, a *Lei 10* (sentido esquerda para direita) especifica uma transformação que altera a visibilidade de um método de `private` para `public`. Eles não definiram nenhuma restrição para essa transformação. Porém, como vimos nos Códigos Fonte 5.5 e 5.6, aumentar a visibilidade de um método pode habilitar a sobrescrita dele e alterar o comportamento do programa. Podemos utilizar nossa abordagem para avaliar as leis propostas por Duarte, ajudando a aprimorar essas leis e aumentar a confiança de que estão corretas.

7.1.2 Geração Automática de Entradas para Testes

Marinov e Khurshid [45] propuseram o TESTERA, um gerador de dados para testes baseado na abordagem *bounded exhaustive testing*. Esse gerador utiliza Alloy e Alloy Analyzer para fazer uma geração declarativa, ou seja, os desenvolvedores especificam as restrições de que tipo de dado deve ser gerado, e o gerador busca no espaço de estados os dados que satisfazem essas restrições. Por outro lado, Daniel et al. [14] desenvolveram um framework imperativo para gerar programas Java utilizando a mesma abordagem *bounded exhaustive testing*. Na geração imperativa, o desenvolvedor especifica *como* o programa deve ser gerado, ao invés de *o que* deve ser gerado (geração declarativa). Nosso trabalho é relacionado com esses trabalhos anteriores, mas diferencia-se por combinar a abordagem declarativa e imperativa para gerar os programas. Dessa forma, o usuário pode especificar declarativamente as características estruturais dos programas. Por outro lado, para a parte comportamental, implementamos geradores a partir do ASTGen que inicializam os atributos e geram os corpos dos métodos.

7.1.3 Testes de Ferramentas de Refatoramento

Daniel et al. [14; 37] propõem uma técnica para automatizar os testes em ferramentas de refatoramento. O principal elemento da técnica é o gerador de programas Java ASTGen. Eles utilizam os programas gerados como entrada para o módulo de refatoramento do Eclipse. Para detectar refatoramentos incorretos, eles utilizam oráculos que analisam sintaticamente o código resultante. Eles detectaram 21 *bugs* no Eclipse e 24 *bugs* no Netbeans. A maioria dos *bugs* introduz erros de compilação no programa refatorado. Essa abordagem é complementar a nossa. Nós focamos em detectar mudanças comportamentais (mais difíceis de

serem detectadas). Porém, também identificamos 15 *bugs* que introduzem erros de compilação. Nós também checamos manualmente o Netbeans com relação ao *bugs* que encontramos. Descobrimos que 22 dos 35 *bugs* de mudança comportamental também ocorrem nessa IDE. Inicialmente, tentamos adotar o ASTGen para testar o Eclipse utilizando nossa técnica. Porém, notamos que os geradores implementados no ASTGen geram programas com pouca expressividade estrutural. Por exemplo, os programas não possuem sobrecarga e sobrescrita de método. Tentamos implementar geradores com uma maior expressividade estrutural no ASTGen, porém, percebemos que o esforço era muito alto para criar programas com uma boa variação estrutural. Sendo assim, decidimos avaliar se a abordagem declarativa do Alloy seria mais interessante para a geração da parte estrutural dos programas, e desenvolvemos o JDolly.

7.2 Trabalhos Futuros

Em trabalhos futuros, pretendemos aprimorar nossa técnica para detectar mudanças comportamentais em refatoramentos, aumentar a expressividade do nosso gerador de programas JDolly, e aplicar nossa abordagem de testes automatizados usando nossa técnica e o JDolly em novos experimentos. A seguir, descrevemos como pretendemos alcançar esses objetivos.

7.2.1 Técnica para Detectar Mudanças Comportamentais

A implementação corrente da nossa técnica não detecta mudanças comportamentais na saída padrão da linguagem Java (`System.out.println`). Nós planejamos usar algum padrão de testes que seja útil para detectar alterações nesses tipos de mensagens e incorporar ao Randoop para que ele gere testes que identifiquem estas mudanças.

Wloka et al. [78] propõem uma ferramenta chamada JUnitMX que informa ao desenvolvedor se as mudanças aplicadas ao código estão sendo cobertas pelos testes do programa. Pretendemos incorporar essa funcionalidade do JunitMX ao SAFEREFACTOR para aumentar a confiança de que os testes gerados estão exercitando as mudanças feitas no código.

7.2.2 JDolly

Na implementação atual do JDolly, os métodos gerados possuem apenas uma instrução de retorno. Essa limitação nos impossibilitou de testar refatoramentos relacionados com o corpo do método, como o *Extract Method*. O JDolly utiliza o ASTGen para gerar o corpo dos métodos; esse *framework* possibilita gerar seqüências de instruções. Pretendemos aprimorar o gerador comportamental do JDolly para gerar métodos com mais instruções. Com relação a parte estrutural dos programas, nosso meta-modelo em Alloy especifica classes, métodos, e atributos, além de tipos primitivos. Pretendemos adicionar ao nosso modelo mais elementos da linguagem Java, como *package*, *import*, *static*.

7.2.3 Testes Automatizados

Nós utilizamos nossa técnica de detectar mudanças comportamentais e o JDolly em uma abordagem para testar o módulo de refatoramento do Eclipse. Pretendemos utilizar a mesma abordagem para avaliar outras IDEs utilizadas para automatizar refatoramentos, como o Netbeans. Além disso, pretendemos avaliar implementações de refatoramentos propostas em trabalhos relacionados [61; 70]. Nossa abordagem também pode ser útil para avaliar leis de transformação propostas para Java, como as propostas por Duarte [17].

Apêndice A

Especificação da Geração de Programas Para testar Refatoramentos

Nessa seção, descrevemos as especificações para o JDolly gerar programas para cada refatoramento. Essas especificações importam o meta-modelo Java descrito na Seção 4.2. Eles possuem dois fatos. O primeiro declara restrições que descrevem as características que o programa deve possuir para ser refatorado pelo Eclipse. O segundo especifica restrições adicionais utilizadas no experimento descrito na Seção 5.3.

A.1 Rename Method

O refatoramento *Rename Method* pode ser aplicado em programas que possuam pelo menos um método. O Código Fonte A.1 exibe as especificações para esse refatoramento. Adicionalmente, especificamos que deve existir duas classes (pai e filha) no programa, um método chamado `K` (nome que o método N terá após o refatoramento) com o mesmo número de argumentos do método a ser refatorado, e um método chamado `Test`.

Código Fonte A.1: Especificação do template para o Rename Method

```
open java−meta−model
one sig N extends Id {}
one sig M1 extends Method {}
 fact RenameMethod {
   M1·id = N
}
one sig Test, K extends Id {}
one sig M2 extends Method {}
one sig C1,C2 extends Class {}
fact Extra {
   C1 in C2·extend
   M2·id = K
   #M1·arguments = #M2·arguments
   one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.2 Rename Field

O refatoramento *Rename Method* pode ser aplicado em programas que possuam pelo menos um atributo. O Código Fonte A.2 mostra as especificações para esse refatoramento. Especificamos também que deve existir duas classes (pai e filha) no programa, um atributo chamado `K` (nome que o atributo N terá após o refatoramento), e um método chamado `Test`.

Código Fonte A.2: Especificação do template para o Rename Field

```
open java−meta−model
one sig N extends Id {}
one sig F1 extends Field {}
 fact RenameField {
   F1·id = N
}
one sig Test, K extends Id {}
one sig F2 extends Field {}
one sig C1,C2 extends Class {}
fact Extra {
   C1 in C2·extend
   F2·id = K
   one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.3 Push Down Method

O refatoramento *Push Down Method* move um método da classe pai para a filha. O *template* especificado no Código Fonte A.3 declara que deve existir uma classe pai com um método e uma classe filha no programa. Adicionalmente, especificamos que deve existir um método chamado `Test`.

Código Fonte A.3: Especificação do template para o Push Down Method

```
open java−meta−model
one sig C1,C2 extends Class {}
one sig M extends Id {}
fact PushDownMethod {
  C1 in C2·extend
  one m:Method | m·id = M && m in C1·methods
}
one sig Test extends Id {}
fact Extra {
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.4 Push Down Field

O refatoramento *Push Down Field* move um atributo da classe pai para a filha. O *template* especificado no Código Fonte A.4 declara que deve existir uma classe pai com um atributo e uma classe filha no programa. Adicionalmente, especificamos que deve existir outro atributo com o mesmo nome do atributo refatorado, e um método chamado `Test`.

Código Fonte A.4: Especificação do template para o Push Down Field

```
open java−meta−model
one sig C1,C2 extends Class {}
one sig N extends Id {}
one sig F1 extends Field {}
fact PushDownField {
  C1 in C2·extend
  F1·id = N
  F1 in C1·fields
}
one sig F2 extends Field {}
fact Extra {
  no f: Field | f in C2·fields
  F1·id = F2·id
}
```

A.5 Pull Up Method

O refatoramento *Pull Up Method* move um método da classe filha para a classe pai. O *template* especificado no Código Fonte A.5 declara que deve existir no programa uma classe pai e uma classe filha com um método. Adicionalmente, especificamos que deve existir um método chamado `Test` no programa.

Código Fonte A.5: Especificação do template para o Pull Up Method

```
open java−meta−model
one sig C1,C2 extends Class {}
one sig M extends Id {}
fact PullUpMethod {
  C1 in C2·extend
  one m:Method | m·id = M && m in C2·methods
}
one sig Test extends Id {}
fact Extra {
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.6 Pull Up Field

O refatoramento *Pull Up Field* move um atributo da classe filha para a classe pai. O *template* especificado no Código Fonte A.5 declara que deve existir no programa uma classe pai e uma classe filha com um atributo. Adicionalmente, especificamos que deve existir no programa um método chamado `Test` e um atributo com o mesmo nome do atributo refatorado.

Código Fonte A.6: Especificação do template para o Pull Up Field

```
open java−meta−model
one sig C1,C2extends Class {}
one sig N extends Id {}
one sig F1 extends Field {}
fact PullUpField {
  C1 in C2·extend
  F1·id = N
  F1 in C2·fields
}
one sig F2 extends Field {}
fact Extra {
  F1·id = F2·id
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.7 Encapsulate Field

O refatoramento *Encapsulate Field* é aplicado em um atributo público. O Código Fonte A.7 mostra as especificações para esse refatoramento. Especificamos também que deve existir duas classes (pai e filha) no programa, um método com o mesmo nome do atributo, e um método chamado `Test`.

Código Fonte A.7: Especificação do template para o Encapsulate Field

```
open java−meta−model
one sig N extends Id {}
one sig F1 extends Field {}
fact EncapsulateField {
  F1·id = N
  F1·vis = public
}
one sig M1 extends Method {}
one sig C1,C2 extends Class {}
fact Extra {
  C1 in C2·extend
  M1·id = N
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.8 Add Parameter

O refatoramento *Add Parameter* adiciona um parâmetro a um método. Nosso meta-modelo só permite que o método tenha no máximo um parâmetro. Por isso, para aplicar esse refatoramento, o programa deve ter um método sem parâmetro. O Código Fonte A.8 exibe as especificações para esse refatoramento. Especificamos também que deve existir no programa duas classes (pai e filha), um método chamado `Test`, e um método com o mesmo nome do método refatorado e com o número de parâmetros diferente do método refatorado.

Código Fonte A.8: Especificação do template para o Add Parameter

```
open java−meta−model
one sig K extends Id {}
one sig M1 extends Method {}
fact AddParameter {
  #M1·arguments = 0
  M1·id = K
}
one sig Test extends Id {}
one sig M2 extends Method {}
one sig C1,C2 extends Class {}
fact Extra {
  C1 in C2·extend
  M2·id = K
  M1·arguments ≠ M2·arguments
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.9 Remove Parameter

O refatoramento *Remove Parameter* remove um parâmetro de método. Nosso meta-modelo só permite que o método tenha no máximo um parâmetro. O Código Fonte A.9 exibe as especificações para esse refatoramento. Especificamos também que deve existir no programa duas classes (pai e filha) , um método chamado `Test`, e um método com o mesmo nome do método refatorado e com o número de parâmetros diferente do método refatorado.

Código Fonte A.9: Especificação do template para o Remove Parameter

```
open java−meta−model
one sig K extends Id {}
one sig M1 extends Method {}
fact RemoveParameter {
  #M1·arguments = 1
  M1·id = K
}
one sig Test extends Id {}
one sig M2 extends Method {}
one sig C1,C2 extends Class {}
fact Extra {
  C1 in C2·extend
  M2·id = K
  M1·arguments ≠ M2·arguments
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.10 Chance Visibility

O refatoramento *Change Visibility* pode ser aplicado em programas que possuam pelo menos um método. O Código Fonte A.10 exibe as especificações para esse refatoramento. Adicionalmente, especificamos que deve existir duas classes (pai e filha) no programa, um método com o mesmo nome e número de parâmetros do método refatorado, e um método chamado `Test`.

Código Fonte A.10: Especificação do template para o Change Visibility

```
open java−meta−model
one sig K extends Id {}
one sig M1 extends Method {}
fact ChangeVisibility {
  M1·id = K
}
one sig Test extends Id {}
one sig M2 extends Method {}
one sig C1,C2 extends Class {}
fact Extra {
  C1 in C2·extend
  M2·id = K
  #M1·arguments = #M2·arguments
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

A.11 Move Method

O refatoramento *Move Method* move um método de uma classe para a outra. O Eclipse só permite mover o método para classes que sejam declaradas como atributos da classe em que o método está declarado. O Código Fonte A.11 exibe as especificações para aplicar o refatoramento, movendo o método `k()` da classe `C1` para `C2`. Adicionalmente, especificamos que deve existir um método com o mesmo nome e número de argumentos do método a ser refatorado, e um método chamado `Test`.

Código Fonte A.11: Especificação do template para o Move Method

```
open java−meta−model
one sig C1,C2 extends Class {}
one sig K extends Id {}
one sig M1 extends Method {}
fact MoveMethod {
 one f:Field | f·type = C2 && f·id = F && f in C1·fields
 M1 in C1·methods
 M1·id = K
}
one sig Test extends Id {}
one sig M2 extends Method {}
fact Extra {
  M2·id = K
  #M1·arguments = #M2·arguments
  one m:Method | m·id = Test && m·vis = public && #m·arguments = 0
}
```

Bibliografia

[1] IEEE Std 1219-1998. Ieee standard for software maintenance. IEEE, 1998.

[2] ISO/IEC 14764:1999. Software engineering - software maintenance. ISO and IEC, 1999.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[4] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, October 2004.

[5] P. Borba, A. Sampaio, and M. Cornélio. A refinement algebra for object-oriented programming. In *European Conference on Object-Oriented Programming*, pages 457–482, 2003.

[6] B. Cabral and P. Marques. Exception handling: A field study in java and .net. In *21st European Conference on Object Oriented Programming*, pages 151–175, 2007.

[7] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[8] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-quality Plug-ins*. Addison-Wesley, 2004.

[9] Leonardo Cole and Paulo Borba. Deriving refactorings for aspectj. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 123–134, New York, NY, USA, 2005. ACM.

[10] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[11] Alloy Community. Accessing alloy4 using java api. At http://alloy.mit.edu/alloy4/api.html, 2009.

[12] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31:328–339, 2005.

[13] Márcio Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Federal University of Pernambuco, 2004.

[14] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Foundations of Software Engineering*, pages 185–194, 2007.

[15] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[17] Rafael Machado Duarte. Parallelizing java programs using transformation laws, 2008. Dissertação de Mestrado da Universidade Federal de Pernambuco.

[18] Eclipse.org. Eclipse project. At http://www.eclipse.org, 2009.

[19] Torbjorn Ekman, Ran Ettinger, Max Schafer, and Mathieu Verbaere. Refactoring bugs in eclipse, idea and visual studio. At http://progtools.comlab.ox.ac.uk/refactoring/bugreports, 2008.

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

[21] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[22] Gabriel Freitas, Marcio Cornélio, Tiago Massoni, and Rohit Gheyi. Object-oriented programming laws for annotated java programs. In *International Workshop on Rule-Based Programming*, Brasília, Brazil, 2009.

[23] Robert Fuhrer, Adam Kiezun, and Markus Keller. Refactoring in the eclipse jdt: Past, present, and future. In *Workshop on Refactoring Tools at ECOOP*, 2007.

[24] Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently refactoring java applications to use generic libraries. In *19th European Conference on Object Oriented Programming*, pages 71–96, 2005.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.

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

[27] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A Rigorous Approach for Proving Model Refactorings. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 372–375, Long Beach, United States, 2005.

[28] Rohit Gheyi, Tiago Massoni, and Paulo Borba. An Abstract Equivalence Notion for Object Models. *Elsevier's Electronic Notes in Theoretical Computer Science, Proceedings of Brazilian Symposium on Formal Methods 2004*, 130:3–21, 2005.

[29] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[30] Maayan Goldstein, Yishai A. Feldman, and Shmuel Tyszberowicz. Refactoring with contracts. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 53–64, Washington, DC, USA, 2006. IEEE Computer Society.

[31] James Gosling, Bill Joy, Guy L. Steele Jr, and Gilad Bracha. *The Java Language Specification*. Sun microsystems, 2005.

[32] J He, C A R Hoare, and J W Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[33] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[34] C. A. R. Hoare. Proof of correctness of data representations. pages 385–396, 2002.

[35] D. Jackson, I. Schechter, and H. Shlyahter. Alcoa: the alloy constraint analyzer. In *22nd International Conference on Software Engineering*, pages 730–733. ACM Press, 2000.

[36] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT press, 2006.

[37] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE 2009: Fundamental Approaches to Software Engineering*, March 2009.

[38] Inc Jet Brains. Intellij idea. At http://www.intellij.com/idea/, 2009.

[39] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, page 154.

[40] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[41] Thomas Kuhn and Olivier Thomann. Abstract Syntax Tree. At http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2006.

[42] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2004.

[43] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[44] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[45] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[47] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[48] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[49] Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA, 2005. ACM.

[50] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, 2001. ACM.

[51] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[52] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[53] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *International Conference on Software Engineering*, pages 287–296, 2009.

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

[55] William Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Object-Oriented Programming emphasizing Practical Applications*, pages 145–160, 1990.

[56] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Germany, 1998. Springer-Verlag.

[57] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[58] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Mota, and César Oliveira. Optimizing jml feature compilation in ajmlc using aspect-oriented refactorings. In *Brazilian Symposium on Programming Languages (SBLP)*, pages 117–130, Gramado, Brazil, 2009.

[59] Refactoring.com. Alpha list of refactorings. At http://refactoring.com/catalog/index.html, 2010.

[60] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[61] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for java. In *Object-oriented programing, systems, languages, and applications*, pages 277–294, 2008.

[62] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: Verification of refactorings. In *Programming Languages meets Program Verification*, pages 67–72, 2009.

[63] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. Stepping stones over the refactoring rubicon. In *Genoa: Proceedings of the 23rd European Conference*

on ECOOP 2009 — Object-Oriented Programming, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] Leila Silva, Augusto Sampaio, and Zhiming Liu. Laws of object-orientation with reference semantics. *Software Engineering and Formal Methods, International Conference on*, pages 217–226, 2008.

[65] Gustavo Soares. Making program refactoring safer. In *Student Research Competition at ICSE '10: 32nd International Conference on Software Engineering Proceedings*, Cape Town, South Africa, 2010. ACM.

[66] Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Marcio Cornélio. Saferefactor - tool for checking refactoring safety. In *Tools Session at Brazilian Symposium on Software Engineering*, pages 49–54, Fortaleza, Brazil, 2009.

[67] Gustavo Soares and Rohit Gheyi. A catalog of refactoring bugs in eclipse. Technical Report TR-UFCG-DSC-200911107. Federal University of Campina Grande, Brazil, 2009.

[68] Gustavo Soares, Rohit Gheyi, Tiago Massoni, Marcio Cornelio, and Diego Cavalcanti. Generating unit tests for checking refactoring safety. In *Brazilian Symposium on Programming Languages (SBLP)*, pages 159–172, Gramado, Brazil, 2009.

[69] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. In *IEEE Software*. IEEE, Jul/Aug 2010.

[70] Friedrich Steimann and Andreas Thies. From public to private to absent: Refactoring java programs under constrained accessibility. In *23st European Conference on Object Oriented Programming*, pages 419–443, 2009.

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

[72] Júlio Cesar Taveira, Cristiane Queiroz, Rômulo Lima, Juliana Saraiva, Sérgio Soares, Hítalo Oliveira, Nathalia Temudo, Amanda Araújo, Jefferson Amorim, Fernando Castor, and Emanoel Barreiros. Assessing intra-application exception handling reuse with aspects. In *23rd Brazilian Symposium on Software Engineering*, pages 22–31, 2009.

[73] F. Tip, A. Kiezun, and D. Baumer. Refactoring for Generalization Using Type Constraints. In *Object-oriented programing, systems, languages, and applications*, pages 13–26, 2003.

[74] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, 2001.

[75] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.

[76] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[77] Tobias Widmer. Unleashing the Power of Refactoring. At http://www.eclipse.org/resources/resource.php?id=321, 2007.

[78] Jan Wloka, Barbara G. Ryder, and Frank Tip. Junitmx - a change-aware unit testing tool. In *31st International Conference on Software Engineering*, pages 567–570, Washington, DC, USA, 2009. IEEE Computer Society.