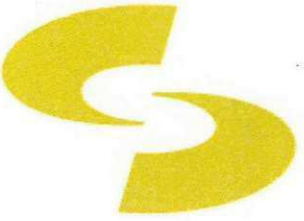


UFCG CEEI Departamento de Sistemas e Computação

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Sistemas e Computação

Uma abordagem para avaliar refatoramentos utilizando análise de impacto e geração automática de testes

Melina Mongiovi

Relatório de Estágio

Orientador Acadêmico: Prof. Rohit Gheyi

Supervisor Técnico: Ghustavo Cunha Lima Sabino

Campina Grande, 2011

Dedico este trabalho à minha família.

Uma abordagem para avaliar refatoramentos utilizando análise de impacto e geração automática de testes

APROVADO EM: .....

BANCA EXAMINADORA

.....
Prof. Dr. Rohit Gheyi
ORIENTADOR ACADÊMICO

.....
Prof. Dr. Tiago Massoni
MEMBRO DA BANCA

.....
Profa. Dra. Joseana Fechine
MEMBRO DA BANCA

Biblioteca Setorial do CDSA. Maio de 2021.

Sumé - PB

Agradecimentos

Agradeço primeiramente aos meus pais, Giuseppe Mongiovi (*In Memoriam*) e Graça Mongiovi, que sempre me deram inspiração e que me apoiaram durante toda minha vida permitindo que eu chegasse até aqui.

Ao meu marido, Gusthavo Sabino, minha filha Camilla Sabino, e todos os meus familiares, por estarem sempre presentes, dando-me forças para continuar.

Ao meu orientador acadêmico, Rohit Gheyi, pela dedicação, empenho, paciência e disponibilidade de orientar meu estágio, como também pela oportunidade de realizar este trabalho.

À professora Joseana Fechine, pela disponibilidade e acompanhamento do estágio.

Aos professores Tiago Massoni (UFCG) e Augusto Sampaio (UFPE), por me ajudarem com sugestões e comentários relevantes.

Aos amigos e companheiros de trabalho, especialmente Gustavo Soares e Catuxe Varjão, que me deram auxílio durante as atividades e tornaram o ambiente de trabalho mais agradável.

A todos meus sinceros agradecimentos.

Melina Mongiovi

Apresentação

Como parte das exigências do curso de Ciência da Computação, da Universidade Federal de Campina Grande, para cumprimento da disciplina de estágio integrado, apresenta-se o relatório de estágio, Uma abordagem para avaliar refatoramentos utilizando análise de impacto e geração automática de testes.

O estágio foi realizado no laboratório SPG (Software Productivity Laboratory) localizado no bloco CN da Universidade Federal de Campina Grande.

O conteúdo do relatório está distribuído conforme descrição a seguir:

- Capítulo 1 – Introdução
- Capítulo 2 – Ambiente de Estágio
- Capítulo 3 – Fundamentação Teórica e Tecnologias Utilizadas
- Capítulo 4 – Atividades do Estágio
- Capítulo 5 – Considerações Finais
- Bibliografia
- Apêndice

Resumo

Na prática, desenvolvedores se baseiam em compilação, testes e ferramentas para garantir a preservação de comportamento durante refatoramentos. Entretanto, as ferramentas de refatoramentos não realizam todas as atividades necessárias para garantir que o refatoramento preserve o comportamento porque não existe nenhuma teoria estabelecendo-as formalmente. Esta atividade é considerada um grande desafio especialmente para linguagens como Java, que possui uma semântica não-trivial. Neste estágio foi proposta uma abordagem para avaliar se uma transformação preserva o comportamento baseada na geração automática de testes apenas para as entidades que possam ter sido impactadas pela mudança. Para isso, foi implementada uma ferramenta chamada Safira, que possui um plugin para o Eclipse. Safira permite aumentar a confiança durante a aplicação de refatoramentos. Além disso, a ferramenta foi avaliada em alguns refatoramentos aplicados a estudos de caso reais de até 20 KLOC. Por fim, a ferramenta foi comparada com outra proposta na literatura com relação a dois requisitos não funcionais: corretude (detecção ou não de mudanças comportamentais) e eficiência (rapidez na avaliação da transformação e tamanho da coleção de testes gerados).

Palavras-chave: refatoramentos, análise de impacto, geração de testes.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Capítulo 1

Introdução

O Estágio Integrado é um elemento acadêmico de fundamental importância, pois nele é possível ter a oportunidade de observar na prática os conhecimentos adquiridos ao longo do curso, além de possibilitar o aprendizado de novas tecnologias e aprofundar em temas que não são contemplados durante o curso, mas que são muito utilizadas na vida profissional fora da academia.

1.1 Delimitação do estágio

O estágio foi realizado no Laboratório SPG (Software Productivity Laboratory), localizado na Universidade Federal de Campina Grande, tendo início em fevereiro de 2011 e finalizado no mês de junho do mesmo ano, período correspondente ao período letivo de 2011.1. O horário utilizado para realizar o trabalho do estágio foi das 14h às 18h de segunda à sexta, durante todo o período.

1.2 Contexto

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, ou seja, aplicando refatoramentos.

1.3 Problema e relevância

Aplicar transformações que preservem o comportamento, principalmente em sistemas de grande porte, não é trivial. Refatoramentos podem ser aplicados manualmente, o que são propensos a erros e consomem bastante tempo, ou com a ajuda de ferramentas que implementam um conjunto de refatoramentos, como o Eclipse e o Netbeans. Entretanto, as ferramentas de refatoramentos não implementam todas as pré-condições necessárias

para garantir a preservação de comportamento para cada tipo de refatoramento, pois, não existe nenhuma teoria estabelecendo-as formalmente. Não é trivial defini-las para linguagens complexas como Java. Com isso, as ferramentas possuem *bugs*.

Na prática, desenvolvedores executam uma coleção de testes antes e depois da transformação para avaliar se houve mudança de comportamento. Contudo, esta coleção pode ser grande, consumindo muito tempo para executá-la, já que os testes não focam em exercitar apenas a mudança. Além disso, em geral, as coleções não focam em exercitar apenas as entidades impactadas pela mudança. Logo, os desenvolvedores necessitam de uma ferramenta melhor que auxilie nesta atividade, dando uma maior confiabilidade e consumindo pouco tempo nessa tarefa.

1.4 Objetivos

A seguir, estão detalhados os principais objetivos do estágio.

1.4.1 Objetivo geral

O objetivo geral do estágio é desenvolver uma ferramenta, chamada Safira, para auxiliar desenvolvedores em atividades de refatoramentos. A ferramenta será um *plugin* para o Eclipse. A partir de uma transformação, Safira irá construir automaticamente uma coleção de testes que exercitará apenas para os métodos impactados pela mudança (métodos que podem mudar o comportamento). Para avaliar se o comportamento foi preservado, a coleção de testes gerada será executada tanto no programa original (antes do refatoramento) e quanto no programa modificado pela transformação. Se os resultados forem iguais, aumenta-se a confiança de que a transformação preserva comportamento. Caso contrário, Safira indicará um conjunto de casos de teste exibindo a mudança comportamental.

1.4.2 Objetivos específicos

- Fazer um levantamento do estado da arte das melhores ferramentas existentes com o intuito de identificar os pontos fortes e limitações;
- Estudar as tecnologias que serão utilizadas;
- Implementar uma ferramenta;
- Comparar a ferramenta proposta com outras da literatura.

Capítulo 2

Ambiente de Estágio

2.1 A Empresa

A construtora São Bernardo é uma empresa atuante no segmento da construção civil em Campina Grande. A empresa é dedicada ao segmento residencial, englobando construções de prédios e casas destinados à moradia. A São Bernardo está situada na Rua Marques do Herval, 58, 2^o andar, sala 02, Centro, Campina Grande.

2.2 Carga horária

Abaixo segue a lista de atividades com suas respectivas cargas horárias.

- Levantamento do estado da arte. Realizada durante os meses de fevereiro e março, totalizando 60 horas.
- Proposta de uma nova abordagem para avaliar refatoramentos utilizando análise de impacto. Realizada durante os meses de fevereiro e março, totalizando 70 horas.
- Planejamento. Realizada entre os dias 14 e 25 de março, totalizando 20 horas.
- Levantamento de requisitos. Realizada entre os dias 25 de março e 5 de abril, totalizando 30 horas.
- Estudo das tecnologias utilizadas. Realizada entre os dias 5 de abril e 22 de abril, totalizando 30 horas.
- Modelagem do sistema. Realizada durante o mês de abril, totalizando 20 horas.
- Implementação. Realizada durante os meses de abril e maio, totalizando 60 horas.
- Validação. Realizada durante os meses de maio e junho, totalizando 40 horas.

2.3 Infraestrutura

2.3.1 Infraestrutura de hardware

- Intel Core i5 2.3 GHz;
- RAM de 4GB;
- HD de 320 GB.

2.3.2 Infraestrutura de software

- Mac OS X 10.6;
- Eclipse 3.6;
- JUnit 4.

2.4 Orientação acadêmica

Nome: Rohit Gheyi
Endereço: Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Avenida Aprígio Veloso, 882 — CEP: 58.429-900
Bodocongó, Campina Grande, PB — Brasil.
Email: rohit@dsc.ufcg.edu.br

2.5 Supervisão técnica

Nome: Gusthavo Cunha Lima Sabino
Endereço:
Belmiro Pinto Brandão, 55
Mirante, Campina Grande, PB — Brasil.
Email: gusthavoclsabino@hotmail.com

Capítulo 3

Fundamentação Teórica

Este capítulo será descrito o processo de software utilizado para o desenvolvimento do Safira. Além disso, serão apresentados também as principais tecnologias utilizadas durante o estágio.

3.1 Processo de desenvolvimento de software

O desenvolvimento da ferramenta foi realizado de maneira iterativa e incremental apoiado por pequenos e freqüentes *releases* do sistema, seguindo algumas práticas das metodologias ágeis Scrum [Schwaber 2004] e XP [Beck e Andres 2004], tais como:

- Reuniões semanais ou diárias. Para acompanhar melhor o projeto, minimizar os riscos e problemas, e tornar mais rápido todo o processo, foram realizadas reuniões semanais ou diárias em algumas etapas mais críticas.
- Desenvolvimento incremental. Primeiramente, foi feito um protótipo da ferramenta. Posteriormente, foi sendo incrementada, assegurando que o programa continuou funcionando corretamente (especialmente durante a implementação do analisador de impacto).
- Desenvolvimento orientado a testes. A cada novo incremento feito na ferramenta, foram realizados testes manuais e comparativos com a ferramenta SafeRefactor.

As próximas subseções descrevem os conceitos de cada etapa do processo de desenvolvimento da ferramenta proposta no estágio.

3.1.1 Planejamento e gerenciamento

A etapa de planejamento e gerenciamento deve assegurar que o projeto de software atenda às restrições de orçamento e de cronograma e que esteja dentro dos padrões de qualidade exigidos. Além disso, é necessário prever todos os possíveis riscos que possam comprometer o projeto.

3.1.2 Engenharia de requisitos

Estudo de viabilidade

O estudo de viabilidade é utilizado para avaliar se o sistema contribui para os objetivos gerais da organização. É feita também uma avaliação para verificar se as necessidades dos usuários identificadas podem ser satisfeitas por meio das tecnologias atuais de software e hardware. O resultado deve fornecer informações para a tomada de decisão quanto a prosseguir para uma análise mais detalhada.

Elicitação e análise de requisitos

É o processo de coleta de requisitos de sistema pela observação de sistemas existentes, discussões com usuários e análise de tarefas. Isto pode envolver o desenvolvimento de um ou mais modelos de sistema e protótipos. Eles ajudam a compreensão do sistema a ser especificado.

Especificação de requisitos

É o processo que reúne informações sobre o sistema proposto e os existentes para obter os requisitos de usuário e de sistema com base nessas informações. Durante esta fase, as fontes de informações, incluem documentação, possíveis usuários do sistema e especificações de sistemas similares.

Validação de requisitos

A validação de requisitos dedica-se a mostrar que os requisitos realmente definem o sistema que o usuário deseja. Esta atividade está relacionada à descoberta de problemas com os requisitos.

3.1.3 Modelagem

A modelagem do sistema consiste em especificar um modelo de domínio e um o projeto arquitetural, um documento identificando os subsistemas constituintes do sistema e os seus relacionamentos, e o diagrama de classes, que representa a estrutura do sistema, recorrendo ao conceito de classes e suas relações. Outras modelagens em diferentes níveis de abstrações também podem ser feitas.

3.1.4 Implementação do sistema

O estágio de implementação do software consiste em desenvolver o sistema executável com base nos requisitos coletados e documentados. Esta atividade pode envolver o desenvolvimento de vários modelos do sistema em diferentes níveis de abstração.

3.1.5 Validação

A validação de software destina-se a mostrar que um sistema está em conformidade com sua especificação. Isso envolve processos de verificação, tais como testes, inspeções

e revisões, a cada estágio do processo de software, desde a definição de requisitos de usuário até o desenvolvimento do programa.

3.2 Ferramentas e tecnologias

3.2.1 JDT (Java Development Tools)

O projeto JDT do Eclipse fornece um conjunto de APIs para acessar e manipular um código fonte em Java. Ele permite acessar, modificar e ler os projetos existentes na área de trabalho, assim como também criar novos projetos. O JDT permite o acesso ao código fonte Java de duas maneiras diferentes: Java Model e Abstract Syntax Tree (AST). O JDT foi utilizado no Safira para identificar as entidades impactadas por uma mudança. Para detectar as mudanças e posteriormente as entidades impactadas pela mudança, foi realizada uma análise estática no código. O acesso aos elementos do projeto foram feitos utilizando as APIs fornecidas pelo Java Model e AST.

Java Model

Cada projeto Java no Eclipse é representado como um modelo Java. O modelo Java do Eclipse é uma representação leve e tolerante a falhas do projeto Java. Ele não fornece muitas informações como Abstract Syntax Tree (por exemplo, ele não fornece informações do corpo de um método), mas é mais rápido de ser recriado em caso de alterações no projeto, ou seja, as informações do projeto são rapidamente atualizadas. Na Tabela 3.1, são ilustrados alguns elementos do projeto e seus respectivos elementos Java Model juntamente com as descrições.

Abstract Syntax Tree (AST)

A AST fornece uma representação em árvore sintática do código em Java. Cada elemento de um arquivo Java é representado como uma subclasse de `ASTNode`. Cada nó `AST` fornece informações específicas sobre o objeto que ele representa. Por exemplo, `MethodDeclaration` representa métodos. Já `VariableDeclarationFragment` representa declarações de variáveis, enquanto que `SimpleName` representa qualquer palavra que não seja uma palavra-chave em Java.

3.2.2 Randoop

O Randoop [Pacheco et al. 2007] é uma ferramenta que gera aleatoriamente uma coleção de testes de unidade para um conjunto de classes passado como parâmetro. Cada teste consiste de uma sequência de chamadas de métodos e construtores que criam e alteram objetos, seguidos por asserções 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`).

Elemento do Projeto	Elemento Java Model	Descrição
Projeto Java	IJavaProject	O Projeto Java que contém todos os outros elementos.
Diretórios src/bin/lib	IPackageFragmentRoot	Código fonte, arquivos binários ou bibliotecas externas.
Cada pacote	IPackageFragment	Subpacotes de IPackageFragmentRoot.
Arquivos .java	ICompilationUnit	Os arquivos Java estão sempre abaixo do nó Package.
Tipos, atributos e métodos	IType/IField/IMethod	Tipos, atributos e métodos.

Tabela 3.1: Elementos do projeto e do Java Model.

Logo após criar uma expressão, o Randoop a executa e recebe o *feedback* do comportamento do programa. Ele checa esse resultado com relação a 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 as asserções do JUnit. Por exemplo, o Randoop possui um contrato que especifica que se um objeto não for nulo, ele 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.

Para compor a nossa técnica, utilizamos como parâmetros para o Randoop, o conjunto contendo os métodos que foram impactados pela mudança. O objetivo é avaliar apenas as partes do código que possam mudar o comportamento devido a transformação. Além disso, só geramos testes para métodos que estão presentes no programa antes e depois do refatoramento. Dessa forma, conseguimos rodá-la no programa original e refatorado.

Capítulo 4

Atividades realizadas

As atividades realizadas foram:

1. Levantamento do estado da arte. Consiste na pesquisa do tema do trabalho de modo a entender algumas técnicas já utilizadas em outros projetos para realizar a análise de impacto. As abordagens estudadas foram aprimoradas e adaptadas para o desenvolvimento da ferramenta. Realizada durante os meses de fevereiro e março, totalizando 60 horas.

2. Propor uma nova abordagem para avaliar refatoramentos utilizando análise de impacto. Essa atividade inclui a formalização de leis que descrevem o impacto no código de algumas transformações primitivas. Essas leis serão utilizadas para implementação do módulo de análise de impacto. Realizada durante os meses de fevereiro e março, totalizando 70 horas.

3. Planejamento. Esta atividade inclui o planejamento de tempo, custo e qualidade do software, assim como também o levantamento de possíveis riscos. Realizada entre os dias 14 e 25 de março, totalizando 20 horas.

4. Levantamento de requisitos, que incluem o estudo de viabilidade, elicitação e análise, especificação e validação de requisitos. Realizada entre os dias 25 de março e 5 de abril, totalizando 30 horas.

5. Estudo das tecnologias que foram utilizadas para desenvolver a ferramenta. Realizada entre os dias 5 de abril e 22 de abril, totalizando 30 horas.

6. Modelagem do sistema. Esta atividade especificou o projeto arquitetural e o diagrama de classes. Realizada durante o mês de abril, totalizando 20 horas.

7. Implementação da ferramenta. Realizada durante os meses de abril e maio, totalizando 60 horas.

8. Validação. Realização de testes comparando com outras ferramentas e eliminando alguns bugs que foram detectados nesta fase. Realizada durante os meses de maio e junho, totalizando 40 horas.

9. Escrita do relatório. Realizada durante os meses de abril, maio e junho.

4.1 Planejamento de Atividades

4.1.1 Cronograma

O cronograma inicial proposto no plano de estágio foi modificado devido a mudanças de necessidades no decorrer do estágio. O cronograma atualizado está descrito no Quadro 4.1.

4.1.2 Reuniões

Foram realizadas reuniões semanais com o orientador acadêmico para acompanhamento das atividades, discussão sobre técnicas e abordagem a serem utilizadas e apresentação dos resultados ao longo do estágio.

4.2 Pesquisa

A pesquisa foi realizada na fase inicial do estágio. Foi feita a partir de trabalhos atuais na área de refatoramentos, análise de impacto, e testes. Algumas abordagens foram propostas na literatura para definir algumas condições e implementar refatoramentos para um subconjunto de Java [Schaefer e Moor 2010, Schäfer et al. 2009, Steimann e Thies 2009]. Entretanto esta não é uma tarefa fácil, já que provar refatoramentos com relação a uma semântica formal foi proposto como um desafio [Schäfer, Ekman e Moor 2008]. Algumas abordagens propuseram uma análise de impacto de mudança para Java [Law e Rothermel 2003, Ren et al. 2004]. Essas abordagens focam em evolução de software em geral e não avaliam a preservação de comportamento de uma transformação.

Para melhorar este problema, foi proposto o SafeRefactor [Soares et al. 2010], uma ferramenta que também analisa o comportamento de uma transformação a partir da geração automática de testes pelo Randoop [Pacheco et al. 2007]. Entretanto, o SafeRefactor considera todos os métodos em comum para geração de testes tornando. Com isso, esta análise pode ser custosa em sistemas grandes, já que os testes gerados não focam em exercitar apenas a mudança.

4.3 Abordagem

A seguir, será mostrado um exemplo motivante para relevar o problema abordado, e em seguida será detalhada a solução proposta.

4.3.1 Exemplo motivante

A Listagem 4.1 descreve um programa em Java composto pela classe A e sua subclasse B. Neste programa o método *teste* retorna 10. Quando aplicado o refatoramento *Pull Up Method* do Eclipse 3.4.2 movendo o método m para a classe A é obtido o programa destino descrito pela Listagem 4.2. Observe que o método *teste* no programa destino

Tarefa	Fev	Mar	Abr	Mai	Jun
Pesquisa sobre o tema abordado	X	X			
Proposta de uma nova abordagem	X	X			
Planejamento		X			
Levantamento de requisitos		X	X		
Estudo das tecnologias			X		
Modelagem			X	X	
Implementação			X	X	
Validação				X	
Escrita de relatório técnico				X	X
Defesa do Estágio					X

Quadro 4.1: Cronograma de atividades do estágio.

retorna 20 diferentemente do programa original. Ou seja, o comportamento foi modificado. Portanto, esta transformação aplicada pelo Eclipse não é um refatoramento. Mesmo as ferramentas recentemente propostas que estabelecem formalmente algumas pré-condições, possuem *bugs* [Schaefer e Moor 2010]. Estabelecer todas as pré-condições com relação a uma semântica formal de Java é não-trivial e muito custoso.

Na prática, uma forma de evitar este cenário e aumentar a confiança de ferramentas, é executando uma coleção de testes no programa antes e depois da transformação e avaliando os resultados. Entretanto, nesse cenário a coleção de testes pode ser modificada também pelo refatoramento, como o *Rename Class*. Como as ferramentas possuem *bugs*, elas podem mudar o comportamento da coleção de testes. Dessa forma, não é o ideal comparar programas com coleções de teste diferentes. O SafeRefactor [Soares et al. 2010], que é uma ferramenta que avalia a preservação de comportamento de transformações, avalia os programas gerando uma mesma coleção de testes. Entretanto a mesma pode ser grande, o que pode consumir tempo, já que a coleção não leva em consideração apenas as entidades impactadas pela mudança.

4.3.2 Solução proposta

A seguir, será detalhada a abordagem proposta. De maneira geral, essa analisa a transformação e gera testes guiados pelo impacto da mudança. O objetivo é aumentar a confiança na avaliação de refatoramentos.

Para ilustrar em mais detalhes a abordagem, considere a transformação apresentada na Figura 4.1. O primeiro passo consiste em identificar as entidades impactadas (construtores e métodos) pela transformação. Para isso é preciso primeiramente caracterizar a

Figura 4.1: O refatoramento Pull Up Method aplicado pelo Eclipse não preserva comportamento.

Listing 4.1: Programa Original

```
public class A {
  public int k(long l) {
    return 10;
  }
  private int k(int l) {
    return 20;
  }
}
public class B extends A {
  public int m() {
    return k(2);
  }
  public int teste() {
    return m();
  }
}
```

Listing 4.2: Programa Refatorado

```
public class A {
  public int k(long l) {
    return 10;
  }
  private int k(int l) {
    return 20;
  }
  public int m() {
    return k(2);
  }
}
public class B extends A {
  public int teste() {
    return m();
  }
}
```

transformação. A transformação foi decomposta em um conjunto de *transformações primitivas*. No exemplo, a transformação pode ser decomposta em quatro transformações primitivas: modificar corpo do método m da classe B, remover o método vazio m da classe B, adicionar o método vazio m na classe A e modificar o corpo do método m da classe A. Para cada transformação primitiva, foi formalizado o conjunto de entidades impactadas pela mesma. Por exemplo, modificar o corpo do método m da classe B impactará ele mesmo, e $B.teste$. O conjunto de todas as entidades impactadas será a união dos conjuntos impactados por cada transformação primitiva. Além disso, serão impactados todos os métodos e construtores que exercitam direta ou indiretamente alguma entidade deste conjunto. No exemplo como não existem métodos que exercitam indiretamente entidades impactadas, então, o conjunto impactado será composto apenas pelos métodos: $B.m$, $B.teste$ e $A.m$. Por fim, foram identificados os métodos e construtores públicos em comuns que pertencem ao conjunto impactado. Neste exemplo os métodos em comum e públicos são $A.k(long)$, $B.k(long)$, $B.m$ e $B.teste$. Entretanto apenas $B.m$ e $B.teste$ pertencem ao conjunto impactado. Então, automaticamente uma coleção de testes que exercite apenas estes dois métodos foi gerada. Os testes foram executados e com base no resultado, foi identificado que a transformação não preserva comportamento.

De maneira geral, o usuário seleciona um refatoramento a ser aplicado no programa origem (Passo 1) e então será gerado um programa destino com base na transformação desejada (Passo 2). A partir das duas versões do programa será feita uma análise de impacto, que identificará as transformações primitivas (Passo 3) e o conjunto de entidades impactadas pela transformação (Passo 4). Posteriormente, serão identificados os métodos em comuns e públicos que exercitam as entidades impactadas (Passo 5). Um

método m é comum ao programa origem e destino se possui a mesma assinatura nos dois programas. No Passo 6, serão gerados automaticamente uma coleção de testes considerando apenas o conjunto de métodos identificados no Passo 5. Os testes serão executados no programa origem (Passo 7) e no programa destino (Passo 8). Por fim, os resultados serão comparados para informar ao usuário se houve mudança de comportamento no Passo 9. A abordagem está descrita na Figura 4.2.

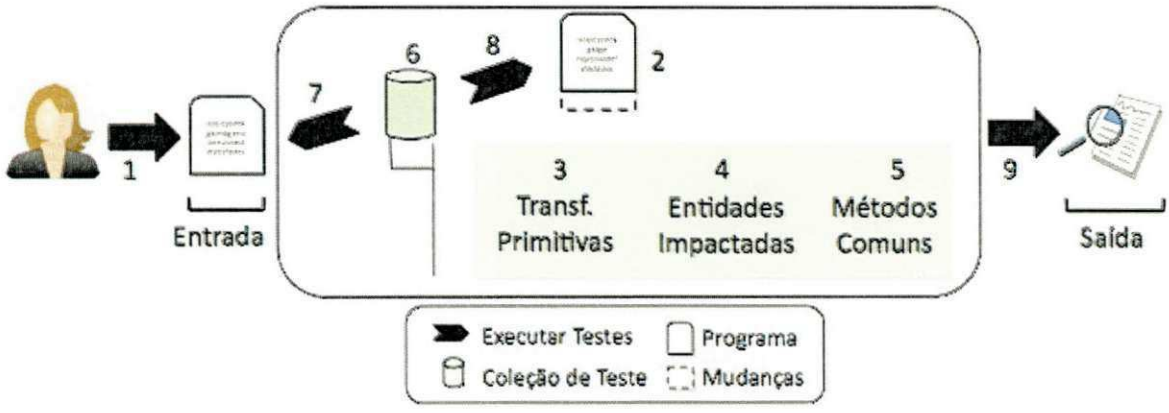


Figura 4.2: Avaliação de refatoramentos baseada no impacto da mudança.

As seções seguintes descrevem em mais detalhes como é realizada a análise do impacto da mudança, e a geração automática de testes guiada pela mudança.

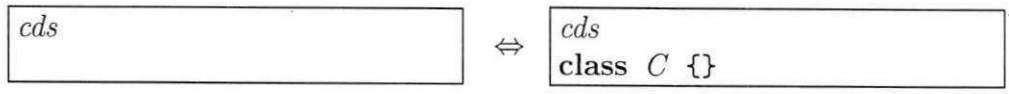
Análise de impacto da transformação

O primeiro passo da análise de impacto é caracterizar a transformação decompondo-a em um conjunto de transformações primitivas. Foram consideradas as seguintes transformações: adicionar e remover classe, adicionar e remover atributo, adicionar e remover método, adicionar e remover herança, e modificar corpo de método. Para cada transformação primitiva, foi formalizado o conjunto de métodos e construtores impactados. O impacto da transformação será dado pela união do impacto de cada transformação primitiva.

A seguir foi descrita a formalização de algumas transformações primitivas usando leis. Cada lei define duas transformações primitivas (da esquerda para a direita, e vice-versa) e declara dois *templates* de um programa. As meta-variáveis *cds*, *ads* e *ops* definem um conjunto de classes, atributos e operações, respectivamente. Por fim, a lei define o conjunto de entidades impactadas ao aplicar a mesma em cada uma das direções. Por exemplo, a Lei 1 define uma transformação adicionar classe vazia, quando aplicada da esquerda para a direita. Como a nova classe não possui métodos, construtores e atributos, ela não impacta nenhuma entidade. A remoção de uma classe vazia também não causa nenhum impacto seguindo um raciocínio similar.

A Lei 2 adiciona um método vazio m em uma classe C quando aplicada da esquerda para a direita. Se a classe C não pertencer a uma hierarquia, o conjunto impactado será formado apenas por $C.m$. Se existir algum ancestral de C na hierarquia que implemente

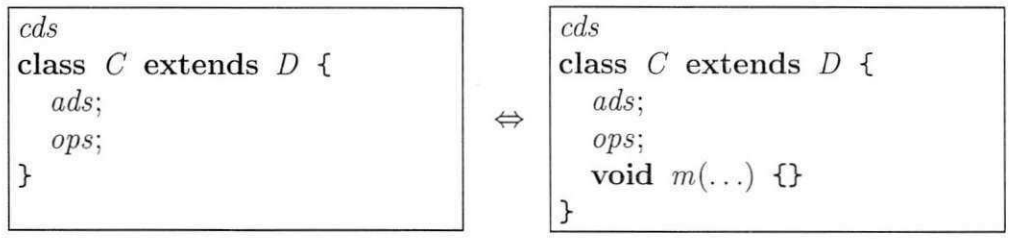
Lei 1 ⟨adicionar classe⟩



(↔) O conjunto de métodos impactados é vazio.

o método m, pode-se ter um impacto devido a sobrecarga e sobrescrita. Nesta caso, serão impactados $C.m$ e todos os métodos que chamam $C.m$ ou qualquer método m herdado de C. Seguindo um raciocínio similar, a remoção de um método pode ter impacto na sobrecarga e sobrescrita de métodos.

Lei 2 ⟨adicionar método⟩



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

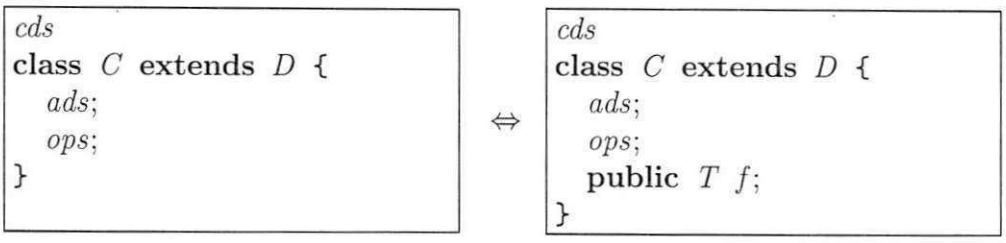
O operador $<^*$ define um relacionamento direto ou indireto de herança. A Lei 3 descreve a transformação adicionar um atributo f do tipo T em uma classe C, quando aplicada da esquerda para a direita. Se a classe C não pertencer a uma hierarquia, o conjunto impactado será vazio. Entretanto, se existir algum ancestral de C na hierarquia que declare algum atributo visível f, pode-se ter um impacto devido a sobrescrita. Serão impactados todos os métodos que usem $C.f$ ou qualquer atributo f herdado de C. De maneira similar, foi definido o impacto causado pela remoção de um atributo.

As outras transformações primitivas foram definidas de maneira similar. O conjunto de todas as entidades impactadas será a união de todos os conjuntos de métodos impactados por cada transformação primitiva. Além disso, a análise também inclui todos os métodos e construtores que exercitam este conjunto impactado direta ou indiretamente.

Geração de testes para as mudanças

Com base no conjunto de entidades impactadas, o objetivo é gerar uma coleção de testes para avaliar apenas as entidades impactadas pela transformação. A fim de aumentar a confiança na análise da transformação, a mesma coleção de testes deverá executar nos programas origem e destino. Desta maneira, é necessário identificar dentre os métodos

Lei 3 ⟨adicionar atributo⟩



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

públicos em comum (de classes públicas) do programa origem e destino, quais exercitam o conjunto de métodos impactados identificado no passo anterior.

A partir daí é gerada uma coleção de testes utilizando o Randoop [Pacheco et al. 2007], que é um gerador automático de testes de unidade que identificou *bugs* nas coleções de Java. O Randoop gera randomicamente um conjunto de testes dentro de um tempo limite informado pelo usuário considerando apenas os métodos que são recebidos como parâmetro. Após a criação da coleção, os testes serão executados no programas origem e destino. Se os resultados da coleção de testes forem iguais, aumenta-se a confiança de que a transformação preserva comportamento. Ou seja, a transformação é um refatoramento. Caso contrário, é mostrado para o usuário um conjunto de testes que mostram a mudança comportamental.

4.4 Planejamento e gerenciamento

4.4.1 Tempo

A estimativa do tempo destinado para cada atividade foi feita no início do estágio. Entretanto, algumas estimativas não foram precisas pelo surgimento de algumas dificuldades ao realizar as atividades. Ao longo do processo de desenvolvimento, as estimativas foram novamente realizadas, e a incerteza foi diminuindo. Na Tabela 4.1, são detalhados a estimativa inicial em horas para cada atividade e o tempo real gasto em cada uma.

4.4.2 Riscos

Na Tabela 4.2 são descritos os três principais riscos que foram identificados no início do estágio e que poderiam levar ao insucesso no processo de desenvolvimento.

4.4.3 Qualidade

O Safira necessita de um alto grau de rigor na sua qualidade (corretude e eficiência). O seu sucesso depende disso. O critério de qualidade do sistema foi avaliado com base em

Tempo		
Tarefa	**Tempo Estimado (h)**	**Tempo Real (h)**
Levantamento do estado da arte	50	60
Propor nova abordagem	40	70
Planejamento	20	20
Levantamos de requisitos	20	30
Estudo das tecnologias	40	30
Modelagem do sistema	20	20
Implementação	70	60
Validação	40	40
Total	**300**	**330**

Tabela 4.1: Estimativa de tempo das atividades do estágio.

Riscos			
Riscos	**Probabilidade**	**Efeito**	**Estratégia**
Não entendimento do domínio (análise de impacto, geração de testes e refatoramentos)	Alta	Sério	Leitura e discussão de artigos no início do projeto
Estimativas não confiáveis por não entendimento do domínio	Alta	Sério	Reuniões diárias de acompanhamento e leitura de artigos para entender o domínio
Requisitos não funcionais críticos	Alta	Sério	Entregar um protótipo e avaliar com o SafeRefactor no final de março

Tabela 4.2: Principais riscos do projeto.

resultados comparativos utilizando o SafeRefactor. O objetivo para atingir a qualidade desejada é manter a corretude, ou seja, detectar os mesmos *bugs*, e melhorar a eficiência do sistema com relação ao tamanho da coleção de testes gerados e ao tempo total da análise.

4.5 Processo de engenharia de requisitos

A seguir, serão descritas as principais etapas do processo de engenharia de requisitos.

4.5.1 Estudo de viabilidade

Primeiramente, foi feita uma pesquisa, realizando a leitura de vários artigos, a fim de aprender algumas técnicas utilizadas em abordagens relacionadas. Foram pesquisadas também as tecnologias que poderiam ser utilizadas. Foi feito um protótipo utilizando o *Reflection*, para realizar a análise de impacto. Para esta tarefa, é necessário ter acesso à todas as entidades do código que está sendo analisado, como classes, métodos, atributos, pacotes, dentre outras. Entretanto, foi constatada uma limitação desta tecnologia. Não é possível obter informações à nível de corpo de método, o que é extremamente necessário para a análise realizada pelo Safira. Além disso, foi observado que ela oferece um desempenho bem abaixo do desejado. Então, foi decidida a utilização da biblioteca AST do Eclipse, pois, ela atendeu melhor às necessidades do projeto.

4.5.2 Elicitação e análise de requisitos

Após o estudo de viabilidade, ficou bem claro que a ferramenta poderia ser desenvolvida dentro do tempo de estágio e com a utilização das tecnologias necessárias. Então, para elicitação de requisitos foram feitas mais pesquisas em artigos e realizadas algumas entrevistas.

Foram utilizados artigos com abordagens similares e temas relacionados para um melhor entendimento do domínio. Algumas entrevistas foram realizadas com desenvolvedores que utilizam ou já utilizaram refatoramentos, e com o orientador acadêmico do estágio, que já trabalha nesta área e entende bem a necessidade da ferramenta neste contexto.

Para um melhor entendimento dos requisitos funcionais do sistema, foi modelado um diagrama de casos de uso que ilustra mais detalhadamente os requisitos desejáveis. O diagrama está ilustrado na Figura 4.3.

4.5.3 Especificação

A seguir, os principais requisitos funcionais e não funcionais do Safira estão especificados.

Requisitos funcionais

Nas Tabelas 4.3 e 4.4, são especificados os dois principais requisitos funcionais do Safira.

RF-01	
Nome:	Avaliar transformação
Descrição:	O sistema deve permitir que o usuário escolha uma transformação a ser aplicada, dentre o conjunto de transformações do eclipse.
Atores:	Usuário do Sistema
Prioridade:	Essencial
Entradas e pré-condições:	A transformação a ser aplicada.
Saídas e pós-condições:	O sistema informa ao usuário se a transformação é um refatoramento.
Fluxos de eventos	
Fluxo principal:	1. O usuário seleciona a transformação que deseja aplicar e clica no botão "Safira"; 2. O sistema avalia a transformação. Esta atividade inclui a análise de impacto de mudanças e geração e execução dos testes. Por fim, o usuário será reportado se a transformação preserva o comportamento.

Tabela 4.3: Requisito funcional: avaliar transformação.

RF-02	
Nome:	Exibir testes falhos
Descrição:	A ferramenta mostra todos os testes que exibem uma mudança de comportamento se a transformação desejada não preservar o comportamento.
Atores:	Sistema
Prioridade:	Essencial
Entradas e pré-condições:	A transformação não preservar o comportamento.
Saídas e pós-condições:	Todos os testes que exibem a mudança de comportamento.
Fluxos de eventos	
Fluxo principal:	1. Após o sistema detectar que houve mudança comportamental, automaticamente serão exibidos os testes falhos.

Tabela 4.4: Requisito funcional: exibir testes falhos.

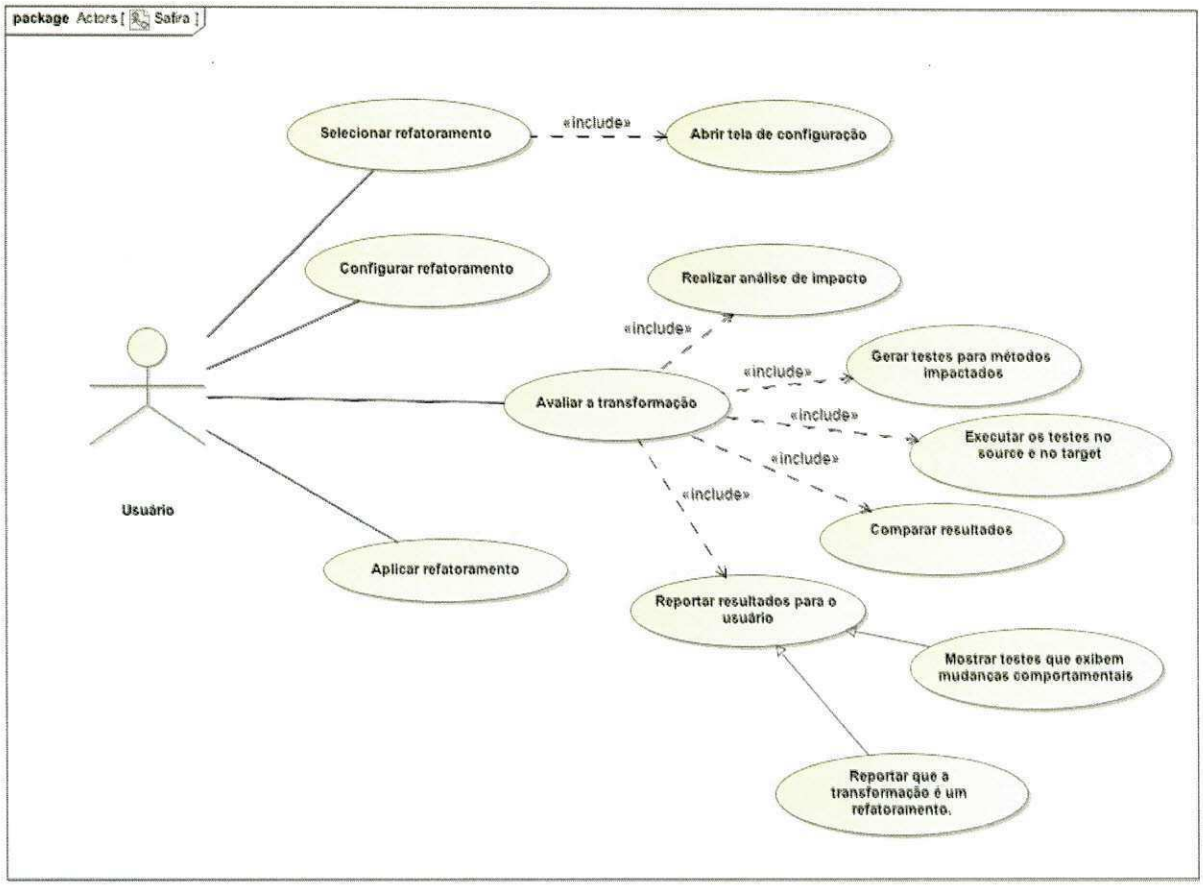


Figura 4.3: Diagrama de casos de uso.

Requisitos não funcionais

Na Tabela 4.5, são descritos os principais requisitos não funcionais.

4.5.4 Validação de requisitos

A validação dos requisitos foi feita juntamente com o orientador acadêmico. Foram realizadas reuniões semanais, possibilitando o contínuo acompanhamento de todo o projeto.

4.6 Análise e Projeto

A seguir, será descrita como foi realizada a etapa de análise e projeto.

4.6.1 Projeto arquitetural

Safira foi desenvolvida com base na arquitetura extensível do Eclipse, que permite desenvolver plugins para adicionar novas funcionalidades. A ferramenta possui dois módulos.

Identificador	Requisito	Descrição
RNF-01	Eficiência	O sistema deve avaliar toda a transformação dentro de no máximo 2 segundos por KLOC.
RNF-02	Corretude	O critério de corretude foi estabelecido comparando os resultados com os do SafeRefactor, uma ferramenta similar que considera todos os métodos em comum do programa.

Tabela 4.5: Requisitos não funcionais.

O *módulo Core*, responsável por implementar o processo da técnica utilizada e o *módulo GUI*, responsável por integrar a implementação da técnica à interface gráfica utilizada para aplicar os refatoramentos do Eclipse. Este último foi reutilizado da ferramenta SafeRefactor. Na Figura 4.4, é ilustrada a arquitetura de Safira.

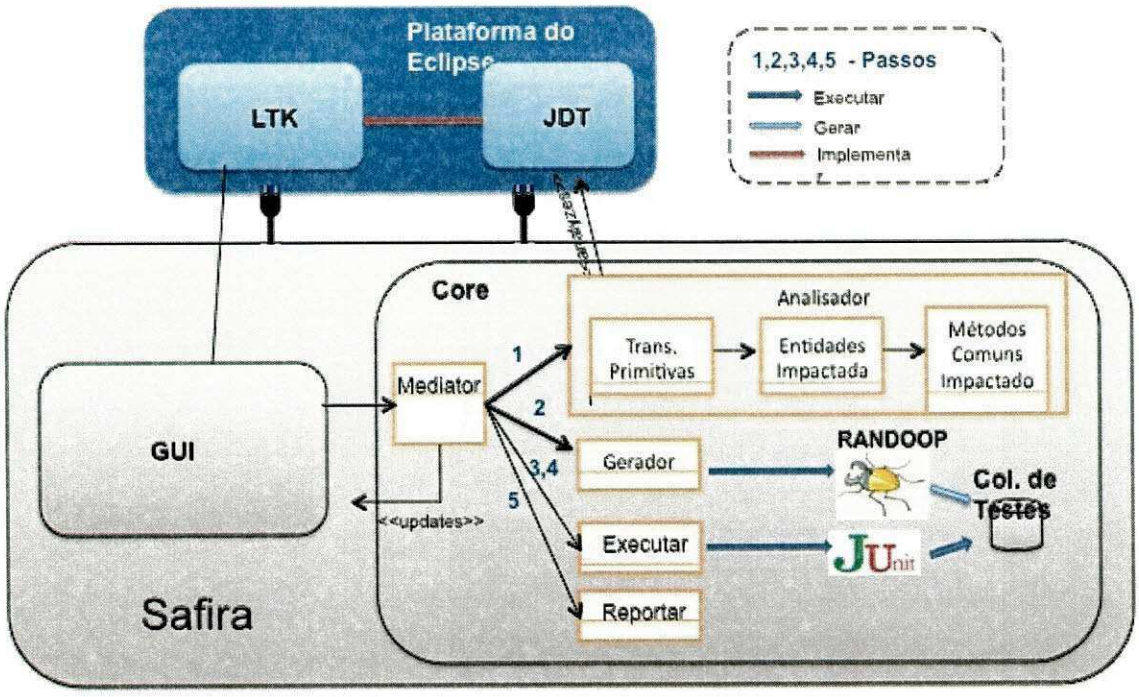


Figura 4.4: Arquitetura do Safira.

Módulo core

O Eclipse possui uma API de automatização de refatoramentos LTK que foi implementada no módulo JDT para criar a ferramenta de refatoração desta IDE. A partir desta API, serão acessadas as informações necessárias para realizar a análise estática no código.

Esta abordagem para avaliar a preservação de comportamento de uma transformação contém 5 etapas. A primeira é realizada no módulo Analisador, responsável por realizar a análise de impacto, detectando as transformação primitivas e as entidades impactadas por cada uma, e por identificar os métodos em comum que exercitam as entidades impactadas. Na segunda etapa o módulo Gerador é responsável por gerar testes para os métodos identificados na etapa anterior. No módulo Executar, são realizadas as Etapas 3 e 4, responsáveis por executar os testes no programa original e no programa modificado, respectivamente. Por fim, na Etapa 5, será reportado para o usuário o resultado da avaliação.

4.6.2 Diagrama de classes

Com base na especificação anterior, especificamos o diagrama de classes, que é apresentado na Figura 4.5. O diagrama ilustra as principais classes e relacionamentos entre elas.

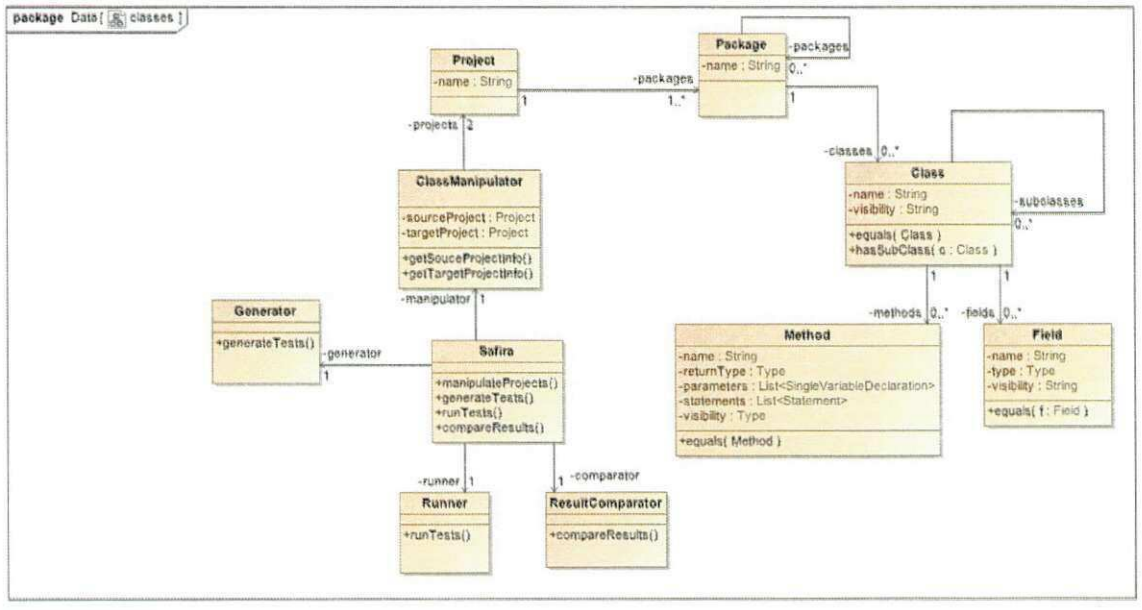


Figura 4.5: Diagrama de classes do Safira.

4.7 Implementação do Safira

Com base na descrição anterior, foi implementada Safira (Figura 4.6), uma ferramenta que avalia se uma transformação preserva ou não comportamento.

O Safira possui duas versões: linha de comando e interface gráfica (*plugin* do Eclipse). A versão linha de comando recebe como entrada dois programas e o tempo limite para geração de testes. Outra ferramenta é um *plugin* do Eclipse. No *plugin* são passados apenas o programa inicial e o refatoramento que o usuário deseja aplicar. A Figura 4.7

Figura 4.6: Logomarca do Safira.

mostra a tela que o usuário escolhe a transformação que deseja aplicar. Para avaliar esta transformação antes dela ser aplicada, deve-se clicar no botão "Safira".

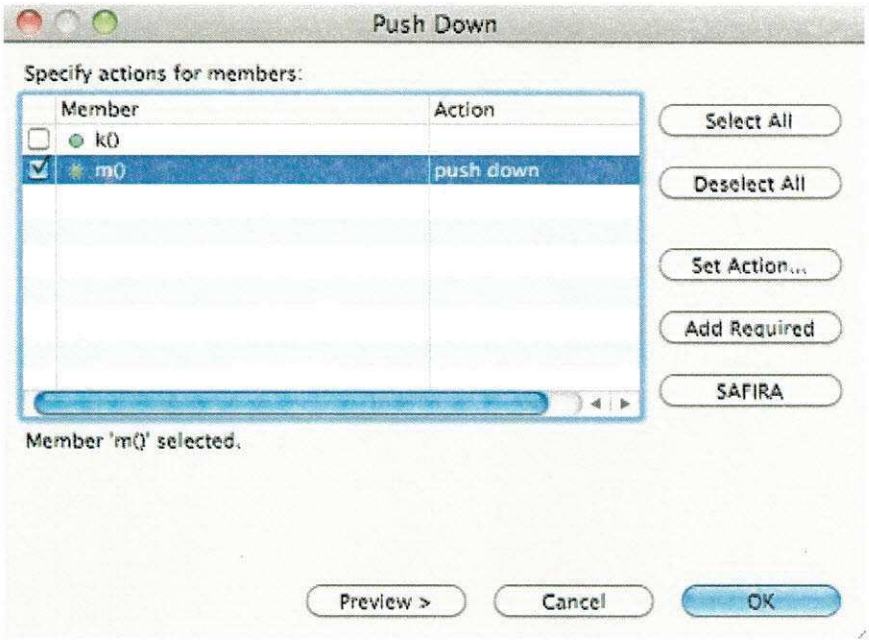


Figura 4.7: Tela de seleção do método e refatoramento.

Se a análise indicar que a transformação preserva o comportamento o usuário prossegue com o refatoramento, caso contrario será mostrada uma tela com os testes que mudaram o comportamento (Figura 4.8).

Foi utilizada a API de refatoramentos do Eclipse do projeto JDT para derivar o programa resultante desejado. A abordagem identifica os métodos em comum e faz a análise de impacto da mudança usando a API do Eclipse que modela a árvore sintática abstrata de Java. Para a geração automática de testes, utilizei o Randoop. Após a geração dos testes, será executado um script pelo ANT, que acionará a execução dos testes pelo JUnit nas duas versões do programa.

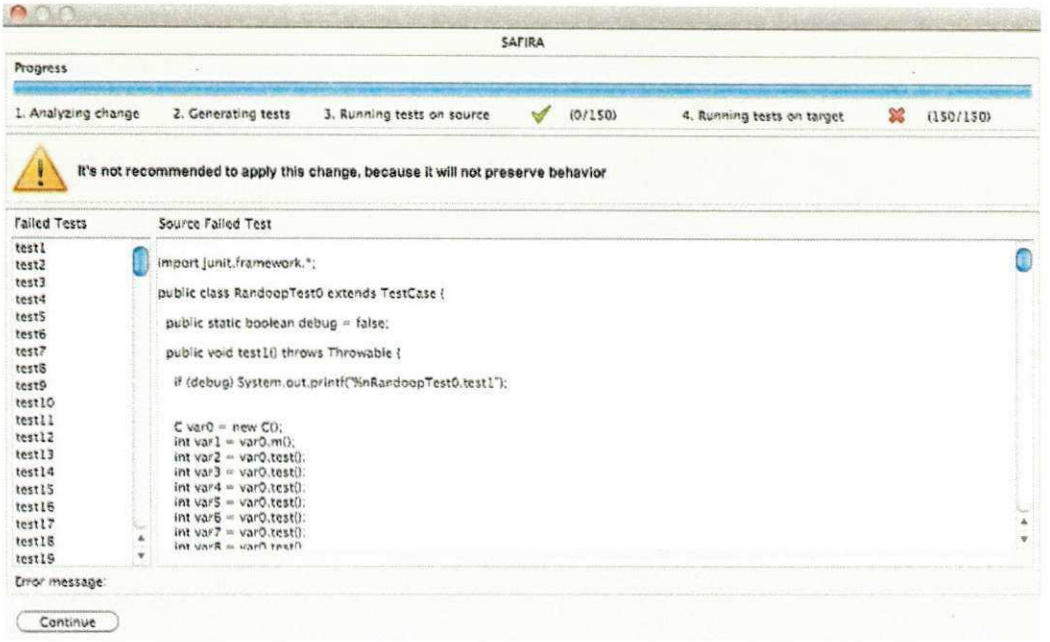


Figura 4.8: Tela de visualização dos testes que exibem mudanças comportamentais.

4.8 Validação

A seguir descrevo como validamos os requisitos.

4.8.1 Requisitos funcionais

A validação da ferramenta foi feita à medida que ela estava sendo desenvolvida, à começar do protótipo. Em cada reunião, foi mostrado o que havia sido desenvolvido, com discussões sobre os riscos, resultados e correções de alguns eventuais problemas.

4.8.2 Requisitos não funcionais

Após a ferramenta ter sido concluída, foram realizados testes comparando-a com outra ferramenta similar, o SafeRefactor. O objetivo da comparação foi avaliar se a corretude foi preservada e se houve uma redução no tempo total de análise e na coleção de testes gerados para avaliar uma transformação. A comparação foi feita de duas formas. A primeira contém refatoramentos aplicados a sistemas reais em Java de 3 a 23 KLOC. Estes foram aplicados por desenvolvedores manualmente ou usando ferramentas, e usaram testes para avaliar a preservação de comportamento. As colunas Refatoramento e KLOC na Tabela 4.6 descrevem o tipo da transformação aplicada e o tamanho do programa em linhas de código, respectivamente. Tanto o SafeRefactor quanto o Safira detectaram as mudanças comportamentais na Transformação 1. Diferentemente do que os desenvolvedores acharam, a Transformação 1 não preserva comportamento. O Safira detectou a mudança comportamental do JHotDraw, com uma coleção de testes 92,3% menor que a do SafeRefactor e em um tempo 64,5% menor.

Conjunto 1: Refatoramento de Sistemas Reais										
Transfor.	Programa	KLOC	Refatoramento	Tempo da Análise da Transfor. (s)		Número de Métodos		Testes Gerados		Tempo Total (s)	
				Safira	SR	Safira	SR	Safira	SR	Safira	SR
1	JHotDraw	23	Extrair Tratamento de Exceção	21.57	35.91	89	2854	38	495	33.34	94.7
2	JUnit	3	Inferir Tipo Genérico	1.73	8.1	35	347	31	296	12.47	53.1
3	VPoker	4	Inferir Tipo Genérico	5.67	8.35	38	39	5	30	31.56	45.5

Tabela 4.6: Refatoramentos em sistemas reais. SR = SafeRefactor.

Nos sistemas JUnit e VPoker não foram detectadas mudanças comportamentais. Observe que o tempo total de análise, números de testes gerados e métodos considerados para a geração de testes foi bem menor no Safira do que no SafeRefactor.

A segunda forma de avaliação contém 10 refatoramentos defeituosos, que não preservam o comportamento quando aplicados pelo Eclipse. Os refatoramentos defeituosos são aplicados a programas pequenos (menos de 15 LOC) contendo até 3 classes e com no máximo 2 métodos por classe. Ou seja, cada programa contém o mínimo de construções necessárias para expor a mudança comportamental. Este conjunto contém 7 tipos diferentes de refatoramentos. A descrição de cada problema está descrito na coluna Descrição do Bug na Tabela 4.7. A abordagem do Safira diminuiu em média 71,11% o tamanho da coleção de testes. Foi reduzido em média 44,64% do tamanho da coleção de métodos, sendo uma redução de 50% em metade das transformações. O tempo total foi reduzido em até 53,18% em relação ao SafeRefactor. Mesmo cada programa contendo poucos métodos e todos relevantes para expor a mudança comportamental, o Safira conseguiu reduzir o tempo de análise da transformação, tamanho da coleção de testes, número de métodos passados para o Randoop e o tempo total de análise. Ou seja, a análise de impacto tornou esta abordagem mais eficiente mantendo a sua corretude. Na Transformação 4 o Safira identificou um método em comum que não foi identificado pelo SafeRefactor. Este é um bug no SafeRefactor. O Randoop gerou a maioria dos testes para esse método. Por isso, nessa transformação, a quantidade de testes gerados e o tempo total de análise foram um pouco maior que os do SafeRefactor.

Com base na avaliação, pode-se concluir que os requisitos não funcionais apresentados na Tabela 4.5 foram satisfeitos.

Conjunto 2: Refatoramentos Defeituosos do Eclipse									
Transfor.	Refatoramento	Descrição do Bug	Tempo da Análise da Transfor. (s)		Número de Métodos		Testes Gerados		Tempo Total (s)	
			Safira	SR	Safira	SR	Safira	SR	Safira	SR
4	Rename Class	Mudança de hierarquia	0.043	0.022	3	4	17	2	6.46	5.14
5	Push Down Method	Habilita sobrecarga	0.024	0.013	3	7	14	114	6.45	13.8
6	Push Down Method	Habilita sobrecarga	0.026	0.013	3	6	16	95	6.46	9.3
7	Pull Up Method	Habilita sobrecarga	0.025	0.011	3	6	30	117	7.07	10.3
8	Pull Up Method	Mudança de visibilidade habilita sobrecarga	0.025	0.011	3	6	13	78	6.18	8.6
9	Pull Up Field	Mudança no valor do atributo	0.022	0.015	2	4	1	3	5.2	5.3
10	Move Method	Habilitar sobrescrita	0.029	0.016	2	4	1	3	5.76	5.8
11	Add Parameter	Habilita sobrescrita	0.023	0.011	3	5	20	80	6.46	8.8
12	Remove Parameter	Habilita sobrescrita	0.026	0.019	5	9	20	96	6.07	9.2
13	Increase Visibility	Mudança de visibilidade habilita sobrescrita	0.024	0.013	4	6	2	83	5.35	9.1

Tabela 4.7: Catálogo de refatoramentos defeituosos aplicados pelo Eclipse. SR = Safe-Refactor.

Capítulo 5

Considerações finais

O estágio teve um caráter tanto profissional, permitindo desenvolver a ferramenta, quanto de pesquisa com a necessidade de aprofundamento em algumas áreas não vistas durante a graduação para entender melhor o problema. O estágio permitiu conhecer um pouco três áreas da Engenharia de Software: Refatoramentos, Análise de Impacto e Testes. Também considero um ponto positivo o aprendizado de novas tecnologias utilizadas para o desenvolvimento da ferramenta para o estágio. Por estes motivos, a experiência no estágio foi muito válida e extremamente enriquecedora para formação tanto acadêmica quanto profissional.

Analisando vários projetos de desenvolvimento de software realizados no passado, autores [Cerpa e Verner 2009] citam os quatro fatores principais que causam o insucesso em projetos de software:

1. processo de desenvolvimento de software inadequado;
2. estimativas não confiáveis;
3. gerenciamento inadequado dos riscos;
4. coleta inadequada dos requisitos.

Houve bastante cuidado nestes quatro fatores.

Com relação ao processo, apesar da literatura propor vários processos, nenhum deles se adequou perfeitamente ao domínio do problema. Por ser um domínio não trivial, preferiu-se adotar algumas práticas (reuniões diárias, refatoramentos, testes, iterações, incrementos) de algumas metodologias, do que seguir um processo à risca. Para minimizar os riscos, especialmente no inicio do projeto tivemos reuniões diárias tanto para entender melhor o domínio como para validar os protótipos desenvolvidos. Isso permitiu entender melhor os requisitos do sistema. Além do entendimento, os requisitos não funcionais do Safira são críticos (corretude e desempenho). O fato de não satisfazer a um deles iria levar ao fracasso da ferramenta. Após algumas semanas de reuniões diárias, permitiu-me entender melhor o domínio e consequentemente minhas estimativas ficaram mais confiáveis. Ao final de março, foi realizada a primeira avaliação da ferramenta. Os resultados relacionados à satisfação dos requisitos não funcionais foram bem promissores. Com isso minimizou-se os principais riscos rapidamente.

De maneira geral, o estágio foi bastante construtivo e relevante. Apesar dos problemas enfrentados, houve um aprendizado em todas as etapas do projeto: realizar a pesquisa, fazer um levantamento do estado da arte, entender o problema, propor uma solução, seguir as etapas de um processo de software, incluindo o levantamento dos requisitos, modelagem do sistema, implementação, inclusive com novas tecnologias, e validar o sistema comparando com outra ferramenta e mostrando claramente o diferencial e vantagens obtidas. A avalicação do requisito não funcional foi uma atividade bastante enriquecedora, já que não tinha realizado durante a graduação. Por fim, é extremamente relevante não só propor uma ferramenta, mas comparar com outras para observar o seu real benefício. Essa foi outra atividade que contribuiu bastante para a formação da estagiária.

Bibliografia

[Beck e Andres 2004]BECK, K.; ANDRES, C. *Extreme Programming Explained: Embrace Change (2nd Edition).* [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321278658.

[Cerpa e Verner 2009]CERPA, N.; VERNER, J. M. Why did your project fail? *Commun. ACM*, ACM, New York, NY, USA, v. 52, p. 130–134, December 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1610252.1610286>.

[Law e Rothermel 2003]LAW, J.; ROTHERMEL, G. Whole program path-based dynamic impact analysis. In: *Proceedings of the 25th International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2003. (ICSE '03), p. 308–318. ISBN 0-7695-1877-X. Disponível em: <http://portal.acm.org/citation.cfm?id=776816.776854>.

[Pacheco et al. 2007]PACHECO, C. et al. Feedback-directed random test generation. In: *Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 75–84. ISBN 0-7695-2828-7. Disponível em: <http://dx.doi.org/10.1109/ICSE.2007.37>.

[Ren et al. 2004]REN, X. et al. Chianti: a tool for change impact analysis of java programs. In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA: ACM, 2004. (OOPSLA '04), p. 432–448. ISBN 1-58113-831-8. Disponível em: <http://doi.acm.org/10.1145/1028976.1029012>.

[Schaefer e Moor 2010]SCHAEFER, M.; MOOR, O. de. Specifying and implementing refactorings. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications.* New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 286–301. ISBN 978-1-4503-0203-6. Disponível em: <http://doi.acm.org/10.1145/1869459.1869485>.

[Schäfer, Ekman e Moor 2008]SCHäFER, M.; EKMAN, T.; MOOR, O. de. Challenge proposal: verification of refactorings. In: *Proceedings of the 3rd workshop on Programming languages meets program verification.* New York, NY, USA: ACM, 2008. (PLPV '09), p. 67–72. ISBN 978-1-60558-330-3. Disponível em: <http://doi.acm.org/10.1145/1481848.1481859>.

[Schäfer et al. 2009]SCHäFER, M. et al. Stepping stones over the refactoring rubicon. In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented*

Programming. Berlin, Heidelberg: Springer-Verlag, 2009. (Genoa), p. 369–393. ISBN 978-3-642-03012-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-03013-$0_1$7¿.

[Schwaber 2004]SCHWABER, K. *Agile Project Management With Scrum*. Redmond, WA, USA: Microsoft Press, 2004. ISBN 073561993X.

[Soares et al. 2010]SOARES, G. et al. Making program refactoring safer. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 27, p. 52–57, 2010. ISSN 0740-7459.

[Steimann e Thies 2009]STEIMANN, F.; THIES, A. From public to private to absent: Refactoring java programs under constrained accessibility. In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2009. (Genoa), p. 419–443. ISBN 978-3-642-03012-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-03013-$0_1$9¿.

Apêndice A

Plano de Estágio

UFCG – UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CEEI – CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
DSC – DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO

Plano de Estágio Integrado

Uma abordagem para avaliar refatoramentos utilizando análise de impacto e geração automática de testes

Melina Mongiovi Cunha Lima Sabino
Mat.: 20711028

Orientador: Prof. Dr. Rohit Gheyi

Fevereiro 2011

Ambiente do Estágio

O estágio será realizado no Laboratório *Software Productivity Group* (SPG), situado na Universidade Federal de Campina Grande, Campina Grande, Paraíba.

O trabalho será realizado por uma equipe de 2 (duas) pessoas, no qual eu atuarei como desenvolvedora e pesquisadora.

Supervisão

Supervisão Acadêmica

Nome: Rohit Gheyi

Endereço: Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Avenida Aprígio Veloso, 882 – CEP: 58.109-970
Bodocongó, Campina Grande, PB – Brasil.

Email: rohit@dsc.ufcg.edu.br

Supervisão Técnica

Nome: Gusthavo Cunha Lima Sabino

Endereço: Belmiro Pinto Brandão, 55
Mirante, Campina Grande, PB – Brasil.

Email: gusthavoclsabino@hotmail.com

Resumo do Problema

.

Refatoramento altera a estrutura interna de um programa preservando seu comportamento. Desenvolvedores podem realizar um refatoramento manualmente, que é susceptível a erros, pois implementar todas as pré-condições necessárias para cada tipo de refatoramento não é uma tarefa trivial e consome bastante tempo. Outra maneira é fazê-lo com ajuda de IDEs como Eclipse, NetBeans, JBuilder, IntelliJ, que pode suportar refatoração. Entretanto, ferramentas de refatoramento podem permitir transformações que gerem mudanças comportamentais.

Para aumentar a confiança desenvolvedores precisam testar a transformarção. Uma maneira é executar os testes de unidade e de regressão de todo o sistema após um refatoramento ter sido realizado. Entretanto, se o sistema for grande, uma boa suíte de testes também será grande, resultando em um grande consumo de tempo para executar todos os testes. Nesta situação, serão executados testes desnecessários que exercitam partes do programa que não foram impactadas pela transformação.

Proposta da Solução

O objetivo do estágio é desenvolver uma ferramenta que analisa se uma transformação preserva o comportamento do sistema, baseada em geração de testes guiada pelo impacto das mudanças.

A partir de uma transformação, nós analisaremos estaticamente todas as micro-mudanças originadas e identificaremos as entidades impactadas por estas mudanças. Em nossa abordagem, entidade impactada é todo método em comum nas duas versões do programa que foi impactado por uma mudança.

Utilizaremos o Randoop [1], uma ferramenta de geração automática de testes, testes serão gerados apenas para as entidades impactadas para avaliar se transformação introduziu mudanças comportamentais.

Metodologia

O desenvolvimento da ferramenta será feita de maneira iterativa e incremental, seguindo uma metodologia ágil com práticas de Scrum [2]. Além de assumir o papel de desenvolvedora, também serei responsável pelo planejamento, documentação e testes do sistema.

Atividades Planejadas

Devem ser desenvolvidas as seguintes atividades no estágio:

1. Desenvolver e documentar os seguintes módulos no sistema:

    a. Módulo de detecção de micro-mudanças no sistema, originadas pela transformação tais como:

        i. Adição de uma nova classe
        ii. Remoção de uma classe
        iii. Adição de um novo atributo
        iv. Remoção de um atributo
        v. Alteração do corpo de um método
        vi. Adição de herança
        vii. Remoção de herança

    b. Módulo de detecção das entidades impactadas pelo conjunto de mudanças.

    c. Módulo de geração de testes automáticos.

2. Desenvolver testes e ajustes na ferramenta.

Resultados Esperados

Espera-se que, ao fim do estágio, o sistema esteja finalizado conforme o cronograma apresentado, e esteja sendo utilizado para resolver o problema identificado. Com a utilização da ferramenta que será desenvolvida, espera-se que desenvolvedores passem menos tempo testando a transformação e tenham mais confiança de que o comportamento do programa não foi modificado após realizar uma atividade de refatoramento.

Cronograma de Atividades

O cronograma de atividades proposto é o seguinte:

Tarefa	***Fev***	***Mar***	***Abr***	***Mai***	***Jun***
Levantamento de requisitos	X				
Cronograma detalhado de trabalho	X				
Planejamento de *releases*	X				
Implementação		X	X	X	
Testes de unidade e ajustes na ferramenta				X	X
Criação da documentação	X	X	X		
Escrita de relatório técnico	X	X	X	X	X
Defesa do Estágio					X

Bibliografia

[1] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in ICSE, 2007, pp. 75–84.

[2] SCHWABER, K., Agile Project Management with Scrum ed. Microsoft Press, 2004. ISBN 978-0-735-61993-7.

Aprovação

Gusthavo Cunha Lima Sabino
Supervisor Técnico

Rohit Gheyi
Supervisor Acadêmico

Joseana Macêdo Fechine
Coordenadora da Disciplina Estágio Integrado