UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RENAN THIAGO DA SILVA ROSA

UMA EXPERIÊNCIA NA REFATORAÇÃO E NO TESTE DE SOFTWARE ORIENTADO A ASPECTOS

Belém - Pará
2018

RENAN THIAGO DA SILVA ROSA

UMA EXPERIÊNCIA NA REFATORAÇÃO E NO TESTE DE SOFTWARE ORIENTADO A ASPECTOS

Trabalho de Conclusão de Curso apresentado para obtenção do título de Bacharel em Ciência da Computação. Instituto de Ciências Exatas e Naturais. Faculdade de Computação. Universidade Federal do Pará.

Orientador: Prof. Dr. Rodrigo Quites Reis

Belém - Pará
2018

RENAN THIAGO DA SILVA ROSA

UMA EXPERIÊNCIA NA REFATORAÇÃO E NO TESTE DE SOFTWARE ORIENTADO A ASPECTO

Trabalho de Conclusão de Curso apresentado para obtenção do título de Bacharel em Ciência da Computação. Instituto de Ciências Exatas e Naturais. Faculdade de Computação. Universidade Federal do Pará.

Orientador: Prof. Dr. Rodrigo Quites Reis

Data da aprovação: Belém-PA. 09/07/2018

Banca Examinadora

_____

Prof. Dr. Rodrigo Quites Reis – Orientador

_____

Prof.ª. Dra. Carla Reis

_____

Prof. Dr. Cleidson

BELÉM
2018

“Nada se ganha abordando esses diversos aspectos simultaneamente. Isso não significa ignorar os outros aspectos, mas estar apenas fazendo justiça ao fato de que, sob o ponto de vista de um aspecto, o outro é irrelevante”.
Edsger W. Dijkstra

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Rodrigo Quites Reis, pela confiança e paciência com que me guiou nesta trajetória.

Agradeço ao Prof. Dr. Fabiano Cutigi Ferrari, pela disponibilização e ajuda com a ferramenta *Proteum/Aj*.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, pois acredito que sem o apoio deles seria impossível vencer esse desafio.

RESUMO

A Programação Orientada a Aspectos é um paradigma de desenvolvimento de software fortemente baseado no princípio de separação de responsabilidades. Este paradigma tem como meta melhorar a modularidade e reduzir a complexidade do software por meio de construtores que encapsulam as chamadas preocupações transversais que se espalham e ou se entrelaçam por todo o software. A refatoração orientada a aspectos intenta aumentar a modularidade e reduzir a complexidade do software já existente, reestruturando as preocupações transversais em um construtor chamado aspecto. No entanto, apesar dos benefícios, a adoção deste paradigma pode representar uma fonte potencial de dificuldades em relação à aplicação de técnicas convencionais de teste de software. Neste cenário, o Teste de Mutação, uma técnica de teste amplamente utilizada que tem por base em uma taxonomia criada a partir de falhas recorrentes cometidas por desenvolvedores, torna-se objeto de estudo deste trabalho. Como resultado deste experimento, notou-se que o teste de mutação aplicado ao software pós-refatoração é capaz tanto de avaliar o software sob teste, quanto avaliar a efetividade do conjunto de casos de testes.

***Palavras-chaves**:* Engenharia de Software, Programação orientada a Aspectos, Refatoração, Teste de Software e Teste de Mutação.

ABSTRACT

Aspect Oriented Programming is a software development paradigm that is strongly based on the principle of separation of concerns and aims to improve modularity and reduce software complexity by means of constructors that encapsulate so-called crosscutting concerns which are scattered over several modules and/or tangled with other concern-specific code. Aspect-oriented refactoring aims to increase modularity and reduce the complexity of existing software by restructuring the crosscutting concerns into a constructor called aspect. However, in spite of the benefits, the adoption of this paradigm may represent a potential source of difficulties in relation to the execution of conventional software testing techniques. In this scenario, the Mutation Testing, a widely used test technique that is based on a taxonomy created from recurrent faults committed by developers, becomes the object of study of this work. As a result of this experiment, it was noticed that the mutation test applied to post-refactoring software is capable of both evaluate the software under test and evaluate the effectiveness of the set of test cases.

Key-words*: Software Engineering, Aspect Oriented Programming, Refactoring, Software Testing and Mutation Testing.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE QUADROS

LISTA DE SIGLAS

ITD	*Inter-Type Declarations*
DAO	*Data Access Object*
AOP	*Aspect Oriented Programming*
XML	*Extensible Markup Language*
ES	*Engenharia de Software*
SQL	*Structured Query Language*
API	*Application Programming Interface*

1. INTRODUÇÃO

A finalidade primeva de um teste é encontrar erros que foram introduzidos involuntariamente pelos desenvolvedores e medir com que grau uma aplicação atende apropriadamente os requisitos especificados. Um bom teste é aquele que tem uma alta probabilidade de encontrar erros. Assim, desenvolvedores devem implementar e desenhar sistemas ou aplicações levando em conta a testabilidade, isto é, a capacidade de ser testado com o mínimo de esforço. Por conseguinte, a não execução da atividade de teste priva o desenvolvedor de obter um importante *feedback* sobre a qualidade presente no software.

Conforme a complexidade da aplicação aumenta, testar características não funcionais como segurança, tolerância a falhas e robustez torna-se muito complicado e dispendioso. Logo, se complexidade é o problema, dividir os requisitos em módulos é a solução. Dividir um problema intrincado em problemas menores é uma estratégia recorrente quando se lida com a alta complexidade de uma tarefa qualquer. Esta divisão quer dizer a modularização de uma responsabilidade (preocupação) bem definida, quando possível, em uma classe. Por conseguinte, a razão da existência de uma classe é exercer alguma responsabilidade, assumir um certo requisito. Neste cenário, garantir que não haja responsabilidades desconexas dentro de uma mesma classe leva a uma alta coesão no software e constitui uma boa prática de programação.

Entretanto, quando não é possível modularizar um requisito (ou responsabilidade) em uma classe específica, a única saída é fundir a implementação deste requisito não funcional junto e ao longo da implementação dos requisitos funcionais. Isso ocorre pois, como o espaço de implementação é unidimensional, focando-se apenas nos requisitos funcionais, técnicas convencionais de modularização são incapazes de lidar com requisitos não funcionais que se espalham por toda a aplicação [Pressman 2016].

Embora, seja possível classificar os requisitos em funcionais e não funcionais na fase de especificação, o paradigma de Programação Orientado a Objetos não fornece construtores que facilitem esta distinção. A fim de tratar estas questões, o paradigma Orientação a Aspectos (AO – em inglês *Aspect-Oriented*) provê mecanismos que habilitam mudanças não invasivas no código existente por meio da modularização destes requisitos em módulos chamados aspectos de acordo com sua importância para a aplicação. Esta capacidade é denominada pela comunidade internacional de Orientação a Aspectos como *separation of concerns*[1] [Laddad 2009]. Assim, é possível fazer injeção de código de maneira não invasiva em momentos-chave do fluxo de execução de um programa ou até mesmo em sua estrutura estática.

1.1. Apresentação do problema

Contudo, o paradigma Orientado a Aspectos também traz consigo alguns possíveis impactos igualmente indesejáveis à aplicação, como: acoplamento entre aspectos e classes ou entre aspectos e aspectos [Cherait and Bounour 2015], dificuldade de análise do fluxo de controle (útil para entender os relacionamentos do código em execução) [Ahmad et al. 2014], aumento na complexidade ciclomática [Pressman 2016] e outros erros (que os programadores comumente cometem) vinculados especificamente ao uso dos construtores introduzidos pela Programação Orientada a Aspectos [Ferrari et al. 2011].

Em face desta nova problemática, as técnicas convencionais de teste de software mostram-se insuficientes para aplicações orientadas a aspectos, justamente por negligenciar a camada de software modularizada em aspectos. Sendo assim, os resultados de um mapeamento sistemático da literatura desenvolvido pelo autor e apresentado no capítulo 3 expõe a quantidade de esforço investido pela comunidade científica na investigação de práticas de testes específicas para software orientado aspectos. Os resultados obtidos no

1 ***Separation of concerns*, embora seja um termo em inglês, é o usado em grande parte das referências brasileiras na área de Orientação a Aspectos, embora as traduções separação de conceitos e separação de preocupações também sejam encontradas.**

Mapeamento Sistemático salientam que ainda há pouca iniciativa para aplicação de testes especificamente para software orientado a aspectos e para mensurar de forma prática o esforço necessário para aplicá-lo.

Neste contexto, uma das técnicas retornadas pelo mapeamento, o Teste de Mutação, tem sido amplamente explorado como uma técnica derivada de critérios de seleção baseado em falhas nomeada como taxonomia de falhas [Ghani and Parizi 2013; Singhal et al. 2013]. Esta técnica se foca em criar versões alteradas do programa original, chamado mutante, inserindo sistematicamente operadores de erros, que são regras para modificar os módulos (classes ou aspectos) com o objetivo de simular falhas [Ferrari et al. 2015]. Ademais, a taxonomia de falhas provê um modelo para a abordagem que levará à detecção de falhas em uma aplicação bem como a avaliação e evolução do conjunto de dados de teste.

1.2. Justificativa para a pesquisa

Tendo em mente todas as adversidades do cenário de teste para software orientado a aspectos, é necessário um estudo sobre o tema a fim de superar tais obstáculos e prover a pesquisadores ou desenvolvedores um melhor entendimento sobre o custo/benefício de aplicação de Testes de Mutação para software orientado a aspectos.

1.3. Objetivos

1.3.1. Geral

Uma vez que é possível modularizar a implementação de requisitos não funcionais por meio de Orientação a Aspectos, este trabalho visa relatar a experiência de refatoração de software usando o paradigma de orientação a aspectos e, por fim, aplicar uma técnica de teste levando em conta as especificidades do paradigma de orientação a aspectos ao software pós-refatoração.

1.3.2. Específicos

Para concretização do objetivo geral deste trabalho, faz-se necessário:

- Mapeamento sistemático da literatura, de maneira a entender o cenário de testes específicos para o paradigma de orientação a aspectos bem com a maturidade e aplicabilidade das técnicas encontradas;
- Refatoração de um software usando o paradigma de orientação a aspectos, que considere o emprego de regras de boas práticas de refatoração para o paradigma, objetivando promover a separação entre as preocupações centrais e transversais presentes no software e consequentemente incrementar sua legibilidade e coesão; e
- Execução de uma técnica de teste no software pós-refatoração com a finalidade de testar não somente as funcionalidades já presentes na aplicação, mas também os construtores que encapsulam as preocupações transversais inseridas no software.

1.4. Organização do trabalho

O restante de trabalho está organizado como segue: O Capítulo 2 apresenta, em resumo, a fundamentação e as principais características do paradigma orientado a aspectos. O Capítulo 3 descreve e agrupa os esforços da comunidade científica em relacionada à literatura que descreve testes para o paradigma de orientação a aspectos. O Capítulo 4 apresenta a terminologia básica utilizada como referência neste trabalho durante os testes. O Capítulo 5 apresenta um modelo de falhas específico para software orientado a aspectos. O Capítulo 6 apresenta a refatoração de um software convencional usando o paradigma de orientação a aspectos. O capítulo 7 analisa os impactos do processo de refatoração usando métricas bem estabelecidas no contexto da engenharia de software. O Capítulo 8 executa o teste do software pós-refatoração usando uma técnica baseada no modelo de falhas descrito no Capítulo 5. O Capítulo 9 resume os resultados e discute as impressões sobre o capítulo anterior.

2. FUNDAMENTOS DA PROGRAMAÇÃO ORIENTADA A ASPECTOS

O paradigma Programação Orientada a Aspectos foi descrito pela primeira vez por [Kiczales et al. 1997]. A principal motivação para o desenvolvimento do paradigma foi tratar problemas de modularidade que dificilmente podem ser resolvidos por abordagens tradicionais de desenvolvimento como a programação procedural e orientada a objetos. Existem requisitos que não podem ser facilmente divididos em domínios ou concretizados em classes, mas se espalham por toda a aplicação, sendo chamados preocupações transversais (*crosscutting concerns).* Como exemplos de requisitos que se encaixam nesta categoria há: segurança, performance e tratamento de exceções. Logo, as preocupações transversais deixam de sobrecarregar o objetivo principal da aplicação, seja este qual for, por meio da adoção de uma nova unidade de modularização; *aspect*.

Na Figura 1, é possível observar os vários módulos do Tomcat [Apache, 2018], uma aplicação Web responsável por executar Java Servlets e renderizar páginas Web. Tomando como exemplo o requisito de *parsing* de documentos HTML, com a programação orientada a objetos é possível modularizar em uma classe distinta (marcada em vermelho) e obter, como consequência, um baixo acoplamento com isso.

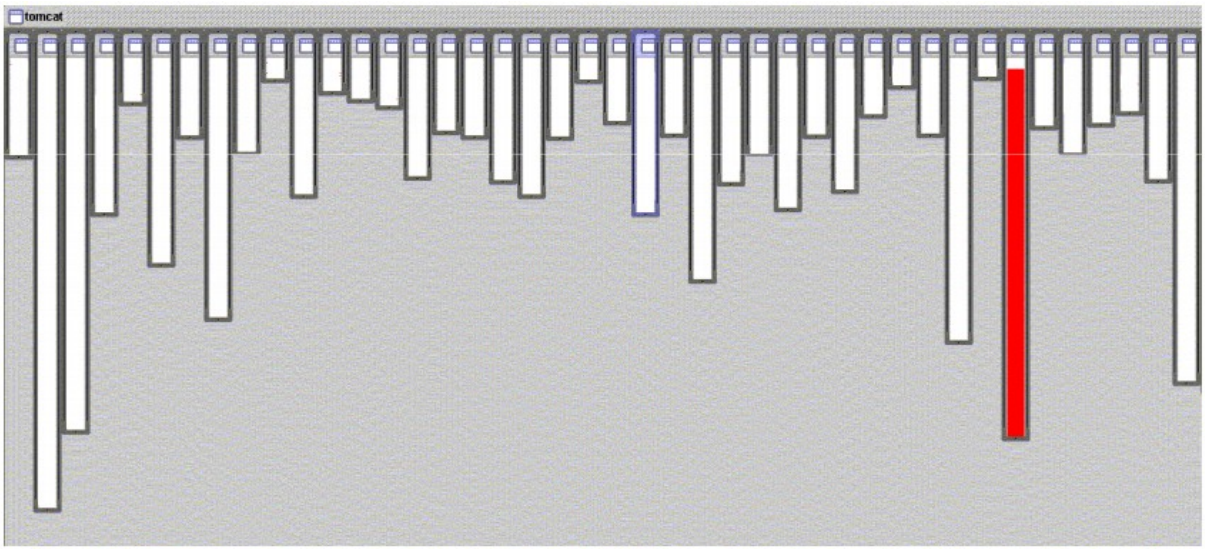

Figura 1: Exemplo de requisito não transversal: *parsing* de documentos HTML. [Hilsdale et al. 2001]

A Figura 2, à diferença da Figura 1, ilustra o entrelaçamento e espalhamento de código na mesma aplicação do requisito de *logging*. Este requisito é dito transversal por exigir que sua solução seja implementada simultaneamente em diferentes classes, justamente pela impossibilidade de separação entre o requisito de *logging* e as demais preocupações centrais. Na Figura 2, cada retângulo representa uma classe com trechos de código-fonte de tamanho variado responsável por lidar com uma parte da preocupação transversal. Esta abordagem de implementação viola o princípio de que cada classe deve ter responsabilidade sobre uma única funcionalidade do software [Martin 2002].

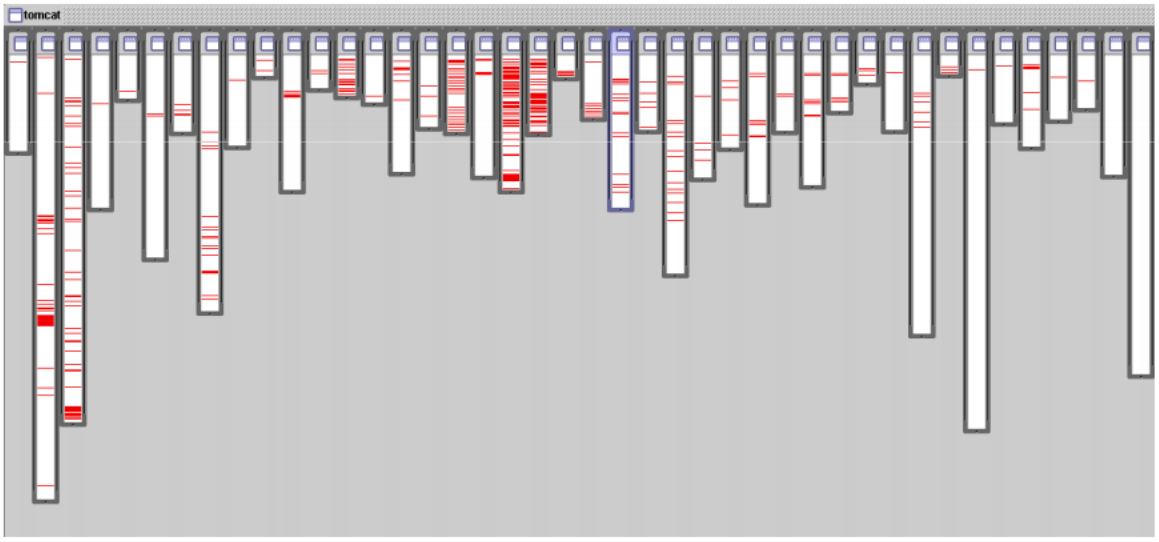

Figura 2: Exemplo de requisito transversal: *logging*. [Hilsdale et al. 2001].

Segundo [Laddad 2009], outra maneira de ver o mesmo problema é entender os requisitos, centrais e transversais, como um espaço multidimensional visto à esquerda na Figura 3. Desta forma abstrata, todos os requisitos são mutuamente independentes e livres de acoplamento, pois as alterações em um eixo não afetam outro. As mudanças de implementação no eixo da segurança, por exemplo, não devem reverberar efeitos colaterais em outros eixos. Entretanto, de fato, esta abordagem colapsa em uma implementação unidimensional convencional, visto no plano unidimensional à direita na Figura 3. Então, a implementação das

regras de negócios torna-se o foco principal e os outros requisitos são espalhados pelo restante da aplicação.

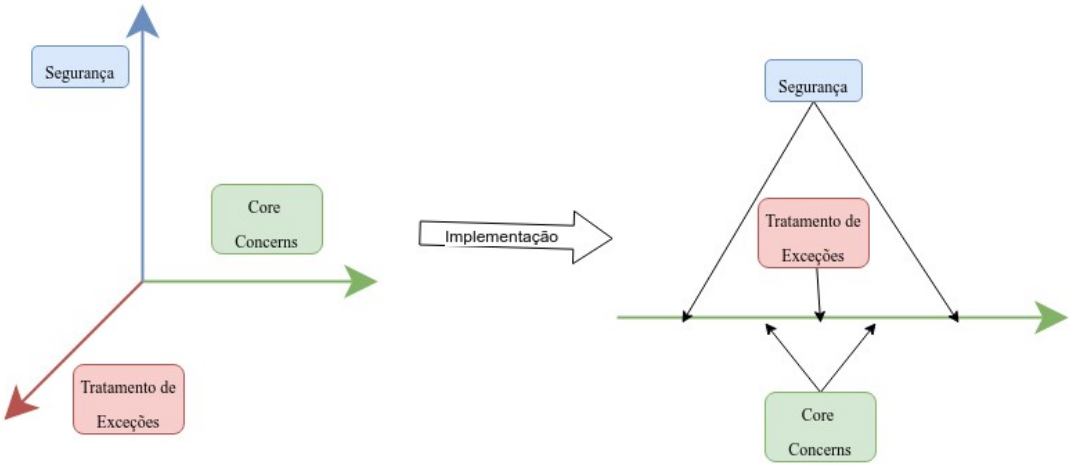

Figura 3. implementando requisitos multidimensionais com uma linguagem unidimensional.

2.1. Modularização com programação orientada a aspectos

Na Programação Orientada a Objetos, as preocupações centrais podem ser parcialmente acopladas através de interfaces, mas não há construtores que permitam fazer o mesmo com preocupações transversais [Silveira 2012].

Na linguagem Java, por exemplo, há estratégias específicas para lidar com acoplamento semântico entre objetos. Na Listagem 1 é apresentado um exemplo de um Objeto de Acesso a Dados (DAO - *Data Access Object*) da entidade Funcionário usado para listar o nome de todos que trabalham em determinado turno, que retorna um *ArrayList* com todos os nomes e os códigos dos funcionários presentes.

Listagem 1: Classe que lista o nome de todos que trabalham em um determinado turno como exemplo de acoplamento.

```
 1 public class FuncionarioDAO {
2     public ArrayList<String> buscaPorTurno(Turno turno) {...}
3
4     public void gravaEmLote(ArrayList<Funcionario> funcionarios) {}
5 }
6 FuncionarioDAO dao = new FuncionarioDAO();
7 ArrayList<String> nomes  = dao.buscaPorTurno(Turno.noite);
8 boolean presente = nomes.contains("George Orwell");
```

Fonte: [Silveira 2012].

Buscas como ilustrada na Listagem 1 são computacionalmente custosas, porém insignificantes em pequena escala. Um retorno do tipo *HashSet* é muito melhor para a escalabilidade do sistema, pois sua implementação da função `contains()` é mais eficiente por usar internamente uma tabela de espalhamento *(hash table)*, possibilitando assim consultas em tempo *O*(1). Entretanto, realizar esta mudança inutiliza o código que já faz uso da implementação acima, visto que há tanto acoplamento sintático (por causa da assinatura do método) e semântico (neste caso, o primeiro tipo de *Collections* permite repetições, ao passo que o segundo não).

Na linguagem Java, é possível tratar este problema utilizando um tipo mais genérico de *Collections.* Assim, é possível alterar o código sem inutilizar qualquer outra instrução de forma simples, pois qualquer mudança no método de busca não requer adequações em instruções que eventualmente utilizem o método `buscaPorTurno()`. Algo similar também pode ser observado para os argumentos recebidos pelo método `gravaEmLote()`. Este exemplo simples é passível de resolução por meios oferecidos pelo próprio paradigma da linguagem de programação. Não obstante, isso nem sempre é possível como já visto para requisitos não funcionais.

Agora, como exemplo, o leitor deve observar uma típica implementação de preocupações transversais em programação orientada a objetos. A classe de

segurança provê um método de validação. As classes dependentes desse método precisam embutir código para invocá-lo, como ilustrada na Figura 4. O módulo de segurança provê seus serviços para outros módulos por meio de uma API que abstrai a complexidade da implementação da camada de segurança. Como resultado, qualquer alteração na implementação do módulo de segurança que não implique alteração na interface do método não requer alteração ou refatoração no cliente. Entretanto, ainda assim, este arranjo ainda demanda que sejam feitas chamadas **explícitas** para a API de segurança.

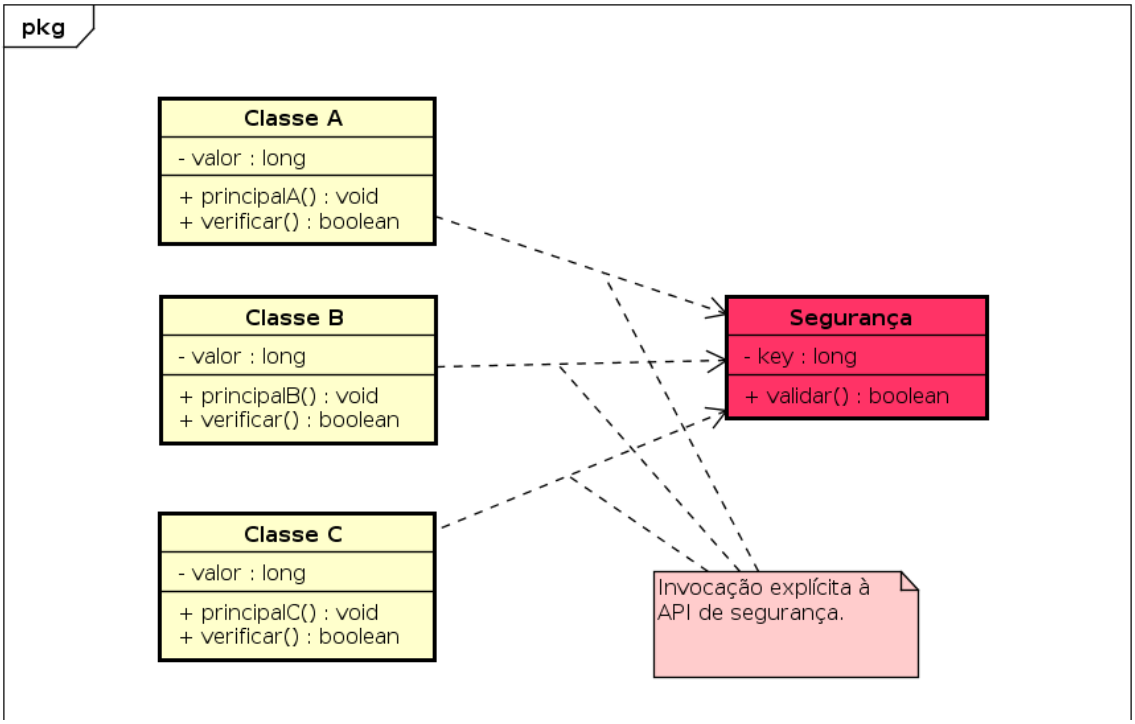

Figura 4: implementando segurança usando paradigma orientação a objetos.

Em programação orientada a aspectos, esta necessidade é suprimida por meio de invocações **implícitas** embutidas no módulo que encapsula as preocupações transversais necessárias à aplicação, resultando em maior simplicidade e legibilidade. A Figura 5 mostra a implementação orientada a aspectos da mesma funcionalidade de segurança vista na Figura 4.

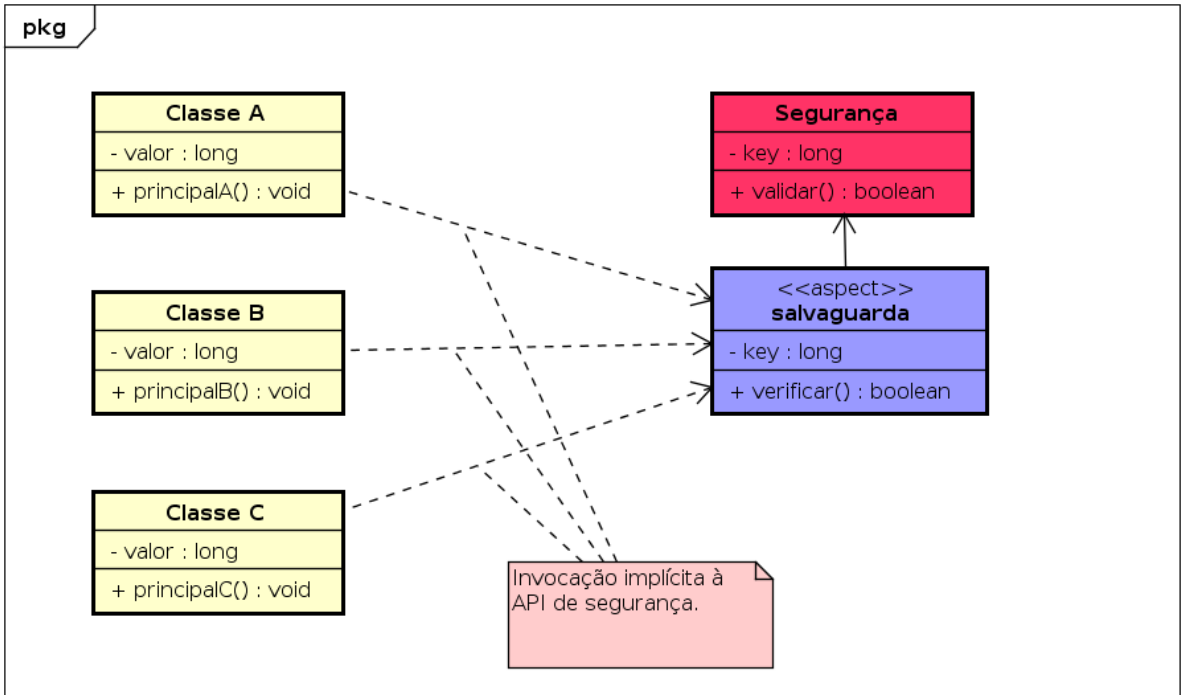

Figura 5: implementando segurança usando paradigma orientação a aspectos.

Na Orientação a objetos, um estilo arquitetural baseado em eventos também oferece o mesmo benefício apresentado na Figura 5, embora sem a mesma facilidade de implementação e transparência. Neste estilo arquitetônico, um evento pode ser um sinal, uma mensagem ou dados de outra função. As funções agem como geradores de evento ou consumidores de eventos e toda comunicação é indireta. Um exemplo conhecido de implementação da arquitetura baseada em eventos é padrão de projeto *Observer [Gamma 1995].* Este padrão define uma relação de um-para-muitos, assim quando um objeto muda de estado todas os seus dependentes são notificados e atualizados automaticamente.

2.2. AspectJ

Uma linguagem de propósito geral baseada em Java que implementa o paradigma de Orientação a Aspectos é chamada AspectJ *[Kiczales et al. 2001]*. Como qualquer outra linguagem, esta é composta de duas partes: a especificação da linguagem, a qual define a sintaxe e a semântica; e a implementação que inclui *weaver*[2] que pode tomar várias formas como um compilador e o *linker*. A linguagem AspectJ foi escolhida para este trabalho por ser o modelo de implementação dos construtores do paradigma orientado a aspecto mais investigada [Ferrari et al. 2015; Ghani and Parizi 2013]. Nas seções a seguir, é mostrado como a linguagem AspectJ mapeia cada elemento proveniente da especificação do paradigma orientado a aspecto em um construtor sintático [Hilsdale et al. 2001; Kiczales et al. 2001; Laddad 2009].

2.3. Construções transversais com AspectJ

As estruturas transversais são classificadas como comuns (*join point, pointcut* e *aspect*), estáticas (declaração) e dinâmicas (*advice*). Estas estruturas formam os blocos de construção que efetivamente habilitam o tratamento das preocupações transversais.

2.3.1. Pontos de execução identificáveis

Um sistema expõem pontos no decorrer de sua execução. Estes pontos podem ser a execução de um método, a criação de objetos ou o lançamento de exceções. Tais pontos identificáveis no sistema são chamados *join points*, e são caracterizados como pontos em um programa em execução nos quais se pode inserir comportamentos adicionais bem como substituir outros. Na Figura 6, o diagrama de sequência mostra alguns possíveis pontos de interesse representados na forma de anotações. Estes pontos são expostos no sistema durante a criação de uma nova categoria.

---

2 **Um processador *weaver* compõe o sistema final combinando as classes e os aspectos através de um processo chamado *weaving (tecelagem)*.**

O leitor deve perceber que vários *join points* são expostos quando o método `incluir`() do objeto `IFCategoria` é invocado. Estes pontos são discriminados na Figura 6 em forma de notação que contém a classe, o método e o seu tipo de ação. Entres as ações, a chamada e execução de métodos são as mais comuns, mas também há leitura e escrita de variável, instanciação de objetos entre outros. (Nem todos os pontos de interesse são mostrados.)

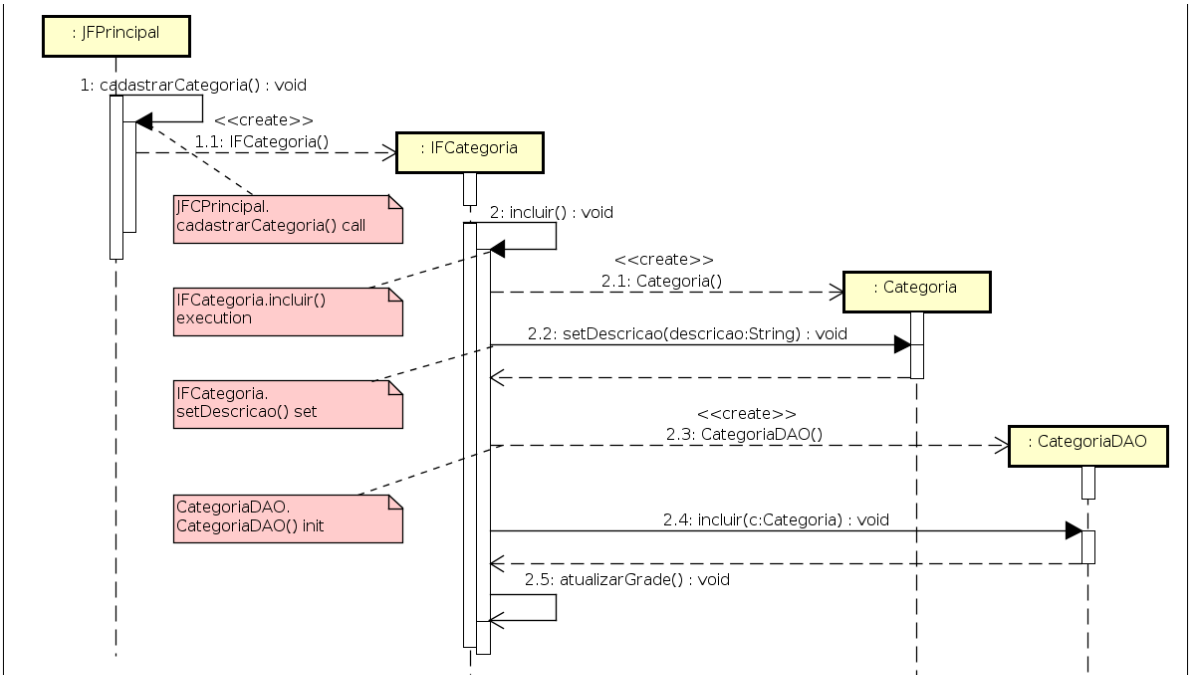

Figura 6: Pontos de interesse expostos em um programa em execução.

2.3.2. Uma estrutura para encapsular pontos de interesse

Implementar um conjunto específico de preocupações transversais requer que um conjunto desejado de *join points* seja encapsulado (ou capturado). A estrutura *pointcut* seleciona um ponto qualquer e seu contexto (informação em tempo de execução), se necessário, que satisfaz uma condição estabelecida pelo programador. Desta forma, um *pointcut* especifica um critério de seleção que atua sobre, principalmente, métodos, campos e anotações usando a assinatura que descreve e discrimina estes pontos, como mostra a Figura 7.

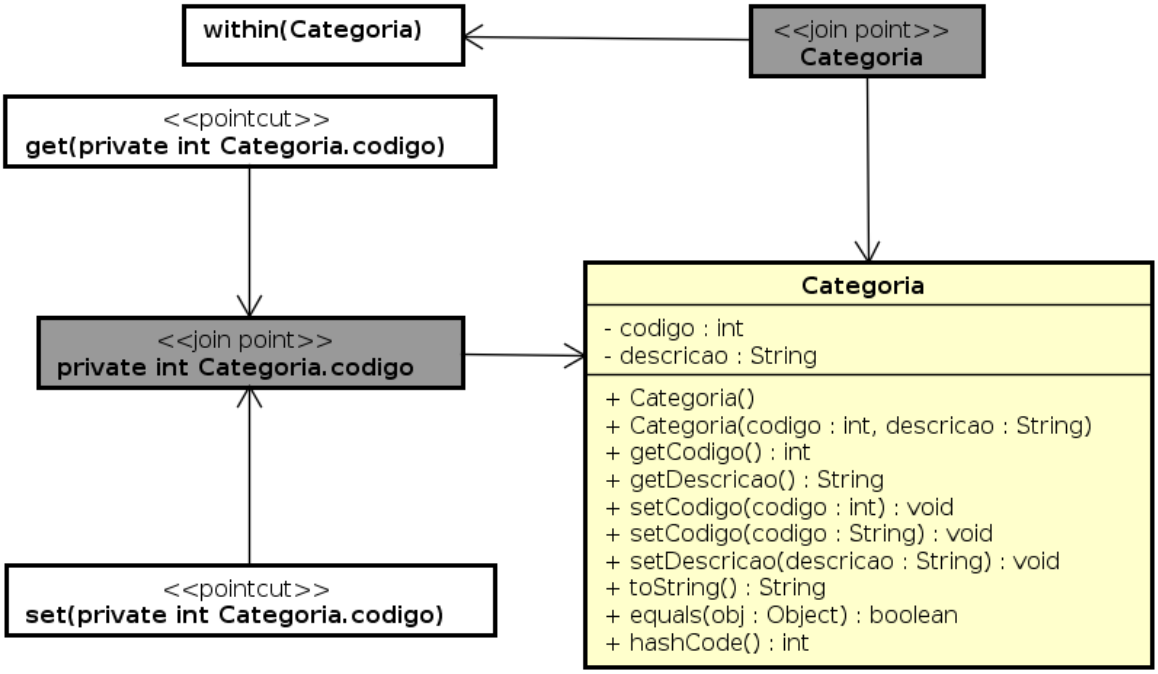

Figura 7: Os *pointcuts get* e *set* selecionam um ponto de interesse representado pelo retângulo cinza .

2.3.3. Uma estrutura para alterar o comportamento dinâmico

A estrutura que fornece o comportamento dinâmico toma forma em um *advice,* uma estrutura semelhante a um método que define o comportamento transversal a ser executado em um ponto de interesse encapsulado. Define-se um comportamento adicional (aumentado) ou alternativo que é inserido antes, depois ou durante a execução do ponto de interesse. Na Tabela 1, são categorizados os tipos de *advice* de acordo com a ordem em que podem ser invocados.

Tabela 1: Classificação de *advice* em disponíveis em AspectJ.

Tipo do a*dvice*	Descrição
before	Executado a *priori* do ponto de interesse.
after (*finally*)	Executado a *posteriori* do ponto de interesse, independente do resultado.
after returning	Executado a *posteriori* da execução com sucesso de um ponto de interesse.
after throwing	Executado a *posteriori* da execução com falha de um ponto de interesse.
around	Executado ao redor do ponto de interesse.

Fonte: [Laddad 2009].

Na Figura 8 é apresentada a sintaxe utilizada em três tipos de *advice* para acrescentar um comportamento dinâmico à aplicação no ponto de interesse `incluir()`. A sintaxe de um *advice* por ser decomposta em três partes: a declaração do *advice* que indica em que momento o mesmo deve ser invocado (antes, durante ou após o ponto de interesse); a especificação do *pointcut* que aponta qual ponto de interesse deve receber um novo comportamento e, finalmente, o corpo do *advice* que implementa o novo comportamento desejado.

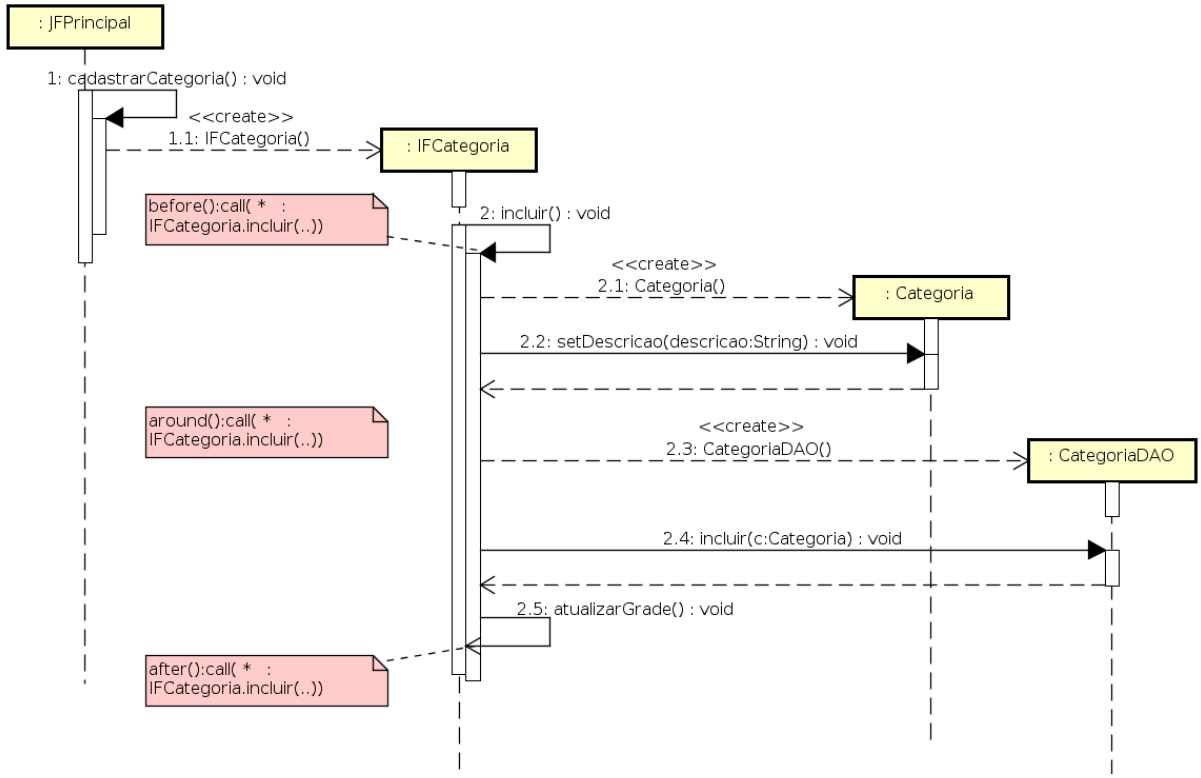

Figura 8: Fluxos alternativos de execução que podem ser inseridos em diferentes pontos de interesse para altera o comportamento original do programa.

2.3.4. Estrutura transversal estática

As mudanças estruturais são realizadas na forma de declarações, chamadas *inter-type declarations* (ITD) ou simplesmente introduções. Esta estrutura altera classes, interfaces ou aspectos do sistema. Este recurso é usado para acrescentar um campo, um método ou a implementação de uma interface em uma classe. O exemplo da Listagem 2 mostra a equivalência entre dois trechos de código.

Listagem 2: Exemplo de modificação estática de uma classe.

Programa original

```
1 public static interface Access{   getLastAccessedTime(); }
2 public class Communicator implements  Access{...}
```

Programa equivalente

```
1 public aspect TrackinkAspect {
2     declare parents :  Communicator  implements Access
3 }
4 public class Communicator{}
```

Fonte: [Laddad 2009].

2.3.5. Um módulo que implementa todas as preocupações transversais

Como a finalidade de um paradigma é ter toda a lógica transversal embutida em um módulo, a estrutura que realiza essa lógica é chamada de *aspect*. Este pode ou não estar relacionado com outros aspectos. A Figura 9 mostra como todos esses conceitos se relacionam em um sistema genérico no qual, independente do modelo de implementação orientado a aspectos, *join points* sempre terão um papel importante.

Ainda na Figura 9, é mostrado uma visão geral dos construtores abordados anteriormente. Os *join points* expostos ao longo da execução do sistema são selecionados por um *pointcut* que é vinculado a um *advice.* O *advice*, por sua vez, altera, aumenta ou acrescenta um comportamento desejado à aplicação em tempo de execução. Outro construtor ilustrado na Figura 9 está relacionado à estrutura estática da aplicação. Este construtor se especializa em dois tipos: introduções (ITD) e declarações de erros e avisos. O construtor do tipo ITD permite alterar, aumentar ou acrescentar novos fluxos ao programa em tempo de tecelagem. Já o *Weave time declarations* permite ao programador declarar em tempo de tecelagem *(weaving process)* erros ou avisos que são disparados se alguma condição estabelecida pelo programador não for satisfeita.

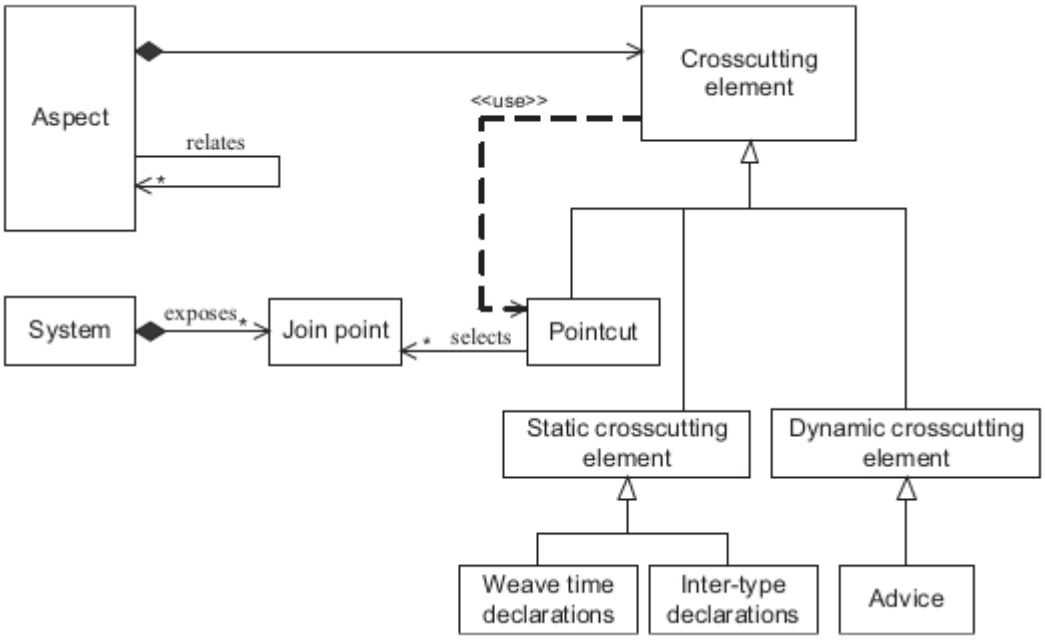

Figura 9: Implementação genérica de um sistema orientado a aspectos. [Laddad 2009]

Por padrão, exatamente uma instância de um tipo de aspecto é criada automaticamente para encapsular um comportamento transversal. Esta instância está disponível em qualquer momento dentro do intervalo de execução de um programa por meio do método estático `aspectof(..)`. Se um aspecto **A** for definido como [<`perthis`>(`Pointcut`)], então um aspecto do tipo A é criado para cada objeto que é o objeto alvo ou em execução (ou seja, *this*) em qualquer um dos pontos de interesse selecionados pelo Pointcut. Da mesma forma, se um aspecto A for definido como [<`pertarget`>(`Pointcut`)], um aspecto do tipo A será criado para cada objeto que seja o objeto de destino dos pontos de interesse selecionados pelo *Pointcut*. Se um aspecto A for definido como [<`percflow`>(`Pointcut`)> ou [<`percflowbelow`>(`Pointcut`)], então um aspecto do tipo A é criado para cada fluxo de controle dos pontos de interesse selecionados por um *pointcut*. Esta instância do aspecto está associada separadamente ao fluxo de controle do ponto de interesse ou abaixo do fluxo de controle do ponto de interesse, respectivamente.

3. MAPEAMENTO SISTEMÁTICO

Este capítulo visa buscar um panorama compreensível a respeito das técnicas de teste para software orientado a aspectos. Outros autores realizaram estudos semelhantes com a intenção de construir um amplo conjunto de referências sobre as técnicas de teste para o paradigma orientado a aspectos, como é mostrado na seção 3.1 deste capítulo. Porém, este trabalho não visa construir um novo conjunto bibliográfico de pesquisas em ES, mas apenas entender o contexto de teste para o paradigma de orientação a aspectos em um período mais recente (2013 a 2017). Este período foi escolhido arbitrariamente na época da realização da pesquisa. A seção 3.1 traz quais outros trabalhos relacionados a teste para software orientado a aspectos existem até o momento desta pesquisa. A seção 3.2 apresenta os objetivos e metodologia usada durante o processo de mapeamento sistemático da literatura. A seção 3.3 destaca as questões a serem respondidas pelo mapeamento sistemático da literatura. A seção 3.4 descreve a estratégia de busca usada para o mapeamento sistemático da literatura. A seção 3.5 define os critérios necessários para incluir ou excluir os resultados obtidos na próxima seção. A seção 3.6 apresenta os resultados do mapeamento após a aplicação dos critérios de inclusão e exclusão definidos na seção anterior. A seção 3.7 categoriza os trabalhos obtidos pelo processo de mapeamento que tratam de Teste de Mutação. A seção 3.8 discute os resultados do mapeamento sistemático da literatura.

3.1. Trabalhos relacionados às técnicas de teste para software orientado a aspecto.

O trabalho de [Singhal et al. 2013] publicou resultados de um amplo mapeamento sistemático da literatura sobre as várias técnicas disponíveis para teste de software orientado a aspectos no período de 1999 a 2013. E, como resultado, o mapeamento inicialmente retornou 8631 publicações após a pesquisa. Destes, 120 foram selecionados para a leitura e revisão de seus resumos e conclusões. Finalmente, 33 foram escolhidos como adequados para o foco do mapeamento e relevantes para a análise. Entre 1999 e 2000 não houve publicações; então, após

2001 houve um aumento perceptível no número de publicações. O estudo ressalta que no início, os pesquisadores estavam mais interessados em projeto e desenvolvimento orientado a aspectos. Posteriormente, à medida que a importância dos testes orientados a aspectos se tornou mais clara, as publicações relacionadas com testes de aplicações orientadas a aspectos também começaram a aumentar em quantidade.

O trabalho de [Ghani and Parizi 2013] também se propõe a construir um panorama compreensível de referências relacionado à realização de testes de software orientado a aspectos, com intento de ajudar iniciantes no assunto a reunir informações. A pesquisa, à semelhança do anterior, considerou um conjunto variado de fontes como: periódicos, conferências e relatórios técnicos publicados entre os anos de 2002 à 2011. O artigo reúne referências para 81 publicações relacionadas com diferentes tipos de teste de aplicações orientada a aspectos, embora sem especificar os detalhes do processo de pesquisa.

3.2. Processo de mapeamento sistemático

A meta principal do mapeamento sistemático é prover um resumo de uma área de pesquisa, identificar a qualidade, tipo de pesquisas e os resultados disponíveis. Regularmente, é desejável mapear a frequência de publicações para delinear possíveis tendências [Marew et al. 2007]. Pesquisadores como [Ghani and Parizi 2013] apontam que, à época de sua pesquisa, ainda havia uma lacuna na literatura científica a respeito de pesquisas vinculadas à produção de estudos referenciais sobre teste de software orientado a aspectos objetivando um cenário mais geral.

Uma metodologia de pesquisa transparente é primordial para que outros pesquisadores também possam replicá-la e verificar os resultados, sem ambiguidade ou ruídos, por si mesmos. Um mapeamento sistemático retorna um amplo intervalo de contextos que tornam as conclusões mais gerais e fáceis de categorizar [Marew et al. 2007]. As atividades essenciais do processo de mapeamento são apresentados na Figura 10. Este modelo de processo usado para este trabalho é baseado, com adaptações às especificidades deste trabalho, em

[Marew et al. 2007]. Cada atividade ilustrada na Figura 10 possui uma saída. O mapeamento sistemático constitui o resultado final do processo. Estas atividades são detalhadas com maior rigor nas seções subsequentes.

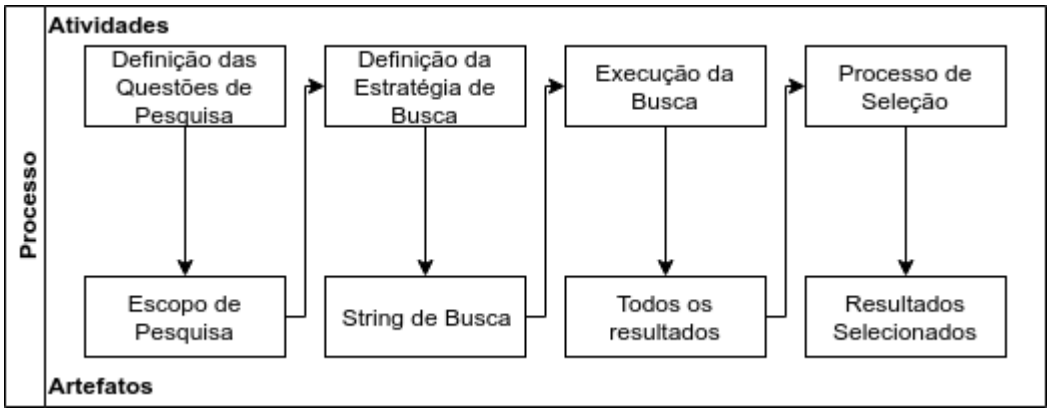

Figura 10: Abstração do processo de mapeamento sistemático.

3.3. Questões de pesquisa

O mapeamento sistemático realizado neste trabalho tem como objetivo principal identificar pesquisas e técnicas de teste em programas orientados a aspectos. Na tabela 2, foram estabelecidas as definições de pesquisa primárias e, para cada uma, critérios de inclusão e exclusão definidos mais à frente.

Tabela 2: Questões que nortearam a pesquisa.

Item	Descrição
Q 01	Quais técnicas e ou critérios de teste têm sido aplicados em software orientado a aspectos?
Q 02	Quais destas técnicas de teste são específicas para software orientado a aspectos?
Q 03	Quais estudos têm sido realizados com o objetivo de validar abordagens de teste para software orientado a aspectos?

Fonte: Autor.

3.4. Definição da estratégia de busca

Definir uma *string* de busca adequada é essencial para a Pesquisa Sistemática ser bem-sucedida, uma vez que a *string* deve retornar o máximo de estudos relevantes. Os procedimentos gerais podem ser resumidos da seguinte forma: a seleção preliminar consiste verificar que estudos primários recuperados dos repositórios devem ser selecionados para leitura completa após uma análise das partes específicas (título e resumo, por exemplo) destes mesmos estudos. Na etapa de seleção final, cada estudo primário deve ser lido na íntegra, selecionado ou descartado, de acordo com os critérios de inclusão e exclusão. Os dados de interesse são então extraídos e armazenados em formulários personalizados. Durante todo o processo, tarefas de documentação devem ser realizadas para permitir auditabilidade e replicabilidade. A expressão de busca apresentada no Quadro 1 foi ajustada para a máquina de busca do portal de periódicos do Capes para realização da primeira fase da pesquisa.

Quadro 1: Expressão de busca usada no mapeamento

TITLE-ABS-KEY(((*aspect-oriented software*) *OR* (*aspect-oriented applications*) *OR* (*aspect-oriented programs*)) *AND* (*testing*))

Fonte: Autor.

A estrutura deve ser guiada por questões de pesquisa e palavras-chave que podem ser retiradas de cada aspecto da estrutura. Por exemplo, o resultado de um estudo pode levar à palavras-chave como: estudos de caso ou experimentos, os quais a abordagem de pesquisa determina a sua acurácia e a sua envergadura.

3.5. Critérios de inclusão e exclusão

Os critérios aqui definidos são usados para justificar a inclusão ou exclusão de cada resultado retornado após a execução da *string* de busca. Estes critérios levam em conta a relevância do trabalho para as questões a serem respondidas pela pesquisa. Estes critérios são categorizados e descritos na Tabela 3.

Tabela 3: Definição dos critérios usados para inclusão ou exclusão.

Critérios de Inclusão	**Descrição**	**Critérios de Exclusão**	**Descrição**
CI 01	O estudo menciona explicitamente no contexto da engenharia de software a questão pesquisada, e, a partir do resumo, é possível deduzir se a publicação contribui para o resultado final.	CE 01	Estudos que não reportam descobertas empíricas relacionadas com as questões de busca ou literaturas que estão apenas disponíveis em forma de apresentação ou resumos.
CI 02	Estudos técnicos e outras literaturas que descrevem estudos empíricos a respeito de teste de software orientado a aspecto.	CE 02	O estudo extrapola o domínio da ES ou não contribui de forma relevante para o resultado final.

Fonte: Autor.

Tabela 3: Definição dos critérios usados para inclusão ou exclusão.

(continuação)

Critérios de Inclusão	Descrição	Critérios de Exclusão	Descrição
CI 03	Mostrar evidência de que os resultados apresentados foram originados a partir de estudos, pesquisas ou práticas.	CE 03	Reportar resultado idêntico a outro estudo já selecionado pelo mapeamento.

Fonte: Autor.

O procedimento para seleção dos estudos foi dividido em três fases conforme descrição a seguir:

a) Aplicação da expressão de busca na fonte escolhida, os resultados obtidos foram catalogados e armazenados, extraindo-se as seguintes informações dos artigos: Título da publicação, Autor(es), Ano da publicação e Resumo/*Abstract*.

b) Os resultados retornados após a execução da fase **a**, passam por uma nova seleção nos quais seus Resumos/*Abstracts* são avaliados conforme os critérios de inclusão e exclusão definidos. Caso o estudo não atenda aos objetivos da revisão, o mesmo é excluído da lista dos estudos de interesse após a execução das seguintes ações: 1) identificação de quais critérios de exclusão são utilizados como justificativa para excluir o estudo; 2) armazenamento das publicações excluídas.

c) Nos estudos selecionados durante a fase **b**, realiza-se uma análise do conteúdo através da leitura na íntegra de cada artigo e aplicação dos critérios de inclusão e exclusão definidos. Os estudos excluídos nessa fase passam pelos mesmos procedimentos de exclusão da fase **b**. Os artigos selecionados para a lista final passam pelos seguintes procedimentos: 1) identificação da(s)

questão/questões que o estudo responde; 2) identificação dos critérios de inclusão utilizados para selecionar o estudo.

3.6. Resultados

Na primeira fase, foram retornados e armazenados 426 resultados, sendo que 8 foram selecionados durante segunda fase, logo após a leitura de seus resumos. A Tabela 4 apresenta, em resumo, na ordem em que foram retornados pela máquina de busca, informações sobre os estudos incluídos e excluídos, em companhia de sua justificativa após a terceira e última fase de seleção.

Tabela 4: Resultado geral do mapeamento sistemático.

Critérios usados	Título do artigo	Ano	Autores
CI 03	Dependence flow graph for analysis of aspect-oriented programs	2014	Syarbaini A. Abdul, G. Fazlida, M. S.
CI 02 03	Testing of aspect-oriented programs: difficulties and lessons learned based on theoretical and practical experience	2015	Ferrari, F. Cafeo, B. Levin, T. Lacerda, J. Lemos, O. Maldonado, J. Masiero, P.
CI 02 03	Evaluating different strategies for integration testing of aspect-oriented programs	2014	Assunção, W. Colanzi, T. Vergilio, S. Ramirez Pozo, A.

Fonte: Autor.

Tabela 4: Resultado geral do mapeamento sistemático.

(continuação)

Critérios usados	Título do artigo	Ano	Autores	Tipo
CI 02 03	An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs	2015	Wedyan, F. Ghosh, Su. Vijayasarathy, L.R.	Teste Estrutural.
CE 02	An Assessment of Software Testability using Fuzzy Logic Technique for Aspect-Oriented Software	2015	Singh, P. Sangwan, O.	Estudo de Caso.
CI 02 03	Using mutation to design tests for aspect-oriented models	2017	Lindström, B. Offutt, J. Sundmark, D. Sten, A. Pettersson, P.	Teste de Mutação.
CE 02	Aspect-oriented program testing: an annotated bibliography	2013	Ghani, A. Parizi, R.	Pesquisa.
CI 02 03	Towards the practical mutation testing of AspectJ programs	2013	Ferrari, F. Rashid, A. Maldonado, J.	Teste de Mutação.

Fonte: Autor.

3.7. Teste de Mutação para programação orientada a aspecto

Pode-se considerar que o mapeamento sistemático não foi conclusivo com relação às técnicas de testes obtidas como resultado. Como consequência disso, e por motivos de aprendizagem, optou-se por explorar a técnica de Testes de Mutação para teste de software orientado a aspectos. A Tabela 5 apresenta como artefato final os resultados que tratam de Testes de Mutação. O próximo capítulo foca unicamente em teste baseado em falhas e na técnica de modelagem de falhas por meio de operadores de mutação - Testes de Mutação.

Tabela 5: Resultado final do mapeamento sistemático

Critérios usados	Título do artigo	Ano	Autor	Tipo
CI 02 03	Testing of aspect-oriented programs: difficulties and lessons learned based on theoretical and practical experience	2015	Ferrari, F. Cafeo, B. Levin, T. Lacerda, J. Lemos, O. Maldonado, J. Masiero, P.	Teste de Mutação, Teste Estrutural e reutilização de conjuntos de teste em diferentes paradigmas de programação.
CI 02 03	Using mutation to design tests for aspect-oriented models	2017	Lindström, B. Offutt, J. Sundmark, D. Sten, A. Pettersson, P.	Teste de Mutação.
CI 02 03	Towards the practical mutation testing of AspectJ programs	2013	Ferrari, F. Rashid, A. Maldonado, J.	Teste de Mutação.

Fonte: Autor.

Como já mencionado em capítulos anteriores, as potencialidades e dificuldades do paradigma provêm um amplo cenário para novas falhas. Identificar e classificar tais falhas representam um importante passo para definição de modelos a serem usados em abordagens de testes. A primeira tentativa para se definir uma taxonomia sobre a natureza de falhas em aplicações orientadas a aspectos foi feita por [Alexander et al. 2004] que, primeiramente, identificou candidatos a possíveis fontes de falhas. Baseado nestas falhas, os autores propuseram uma taxonomia que inclui seis tipos de falhas que foram expandidas no decorrer do tempo por outros autores, mas que em resumo são:

- **Abrangência incorreta na descrição de um *pointcut*** - a abrangência determinada pelo descritor de um *pointcut* determina quais pontos de interesse são selecionados. Se a abrangência for muito pequena, alguns pontos de interesse necessários não são selecionados. Se a abrangência for muito grande, são selecionados pontos de interesse adicionais que devem ser ignorados. Em ambos os casos pode haver um comportamento incorreto.
- **Precedência entre aspectos incorreta -** a ordem de tecelagem é determinada pela especificação de precedência de um aspecto. Um aspecto com maior precedência tem seu *advice before* executado antes do *advice before* em um aspecto de precedência inferior. Se a precedência não for especificada, a ordem em que um *advice* é aplicado permanece indefinida.
- **Estabelecimento incorreto de pós-condições -** espera-se que as pós-condições de um método sejam satisfeitas, independentemente de haver aspectos entrelaçados na lógica de negócios. Logo, as pós-condições não podem causar a quebra de contratos comportamentais em qualquer que seja o contexto da aplicação.
- **Falhas na preservação do estado de invariantes -** invariantes são variáveis que mantém sempre o mesmo valor ao longo de uma iteração. Garantir que o processo de tecelagem não cause violações em invariantes de estado é outro desafio para os desenvolvedores de software orientado a aspectos.
- **Foco incorreto no fluxo de controle -** o descritor de um *pointcut* especifica quais pontos de interesse devem ser capturados. Geralmente há casos em que a informação necessária para tomar corretamente uma decisão de captura só está disponível em tempo de execução. Às vezes, os pontos de interesse devem ser selecionados apenas em um contexto de execução específico. Este contexto pode estar dentro da estrutura de controle de um objeto específico ou dentro do fluxo de controle que ocorre em um nível subjacente de um ponto na execução.

As abordagens de Teste de Mutação para software orientado a aspectos sofrem de uma limitação similar às pesquisas sobre taxonomia de falhas em programação orientada a aspecto. Poucos estudos têm, igualmente, reportado as dificuldades práticas para aplicar Teste de Mutação para software orientado a aspectos [Ferrari et al. 2015].

3.8. Conclusões sobre o mapeamento

Dificuldades provenientes do aumento da complexidade de software dificultam a rastreabilidade de erros, a manutenibilidade de código e a “testabilidade” da aplicação. Entretanto, apesar da programação orientada a aspectos conferir um meio para a superação de alguns destes impedimentos, novos tópicos mostram-se em aberto, em especial - testes. Portanto, deixar que um programador descubra o que não fazer de forma empírica é, fora de qualquer dúvida, no mínimo contraproducente e mais provavelmente desastroso. Ademais, com o que foi apresentado e discutido, espera-se que a exploração do cenário de testes para a programação orientada a aspectos reportada neste trabalho possa ajudar pesquisadores e ou desenvolvedores a coletar informações sobre técnicas praticáveis e efetivas de teste.

4. FUNDAMENTOS DE TESTE DE SOFTWARE

O intuito final uma abordagem de teste de software é revelar defeitos inseridos inadvertidamente durante a implementação. O teste de software também pode servir como uma medida quantitativa para o grau com que um sistema atinge um determinado atributo [Pressman 2016]. Logo, é prudente estabelecer com antecedência uma sistemática para os testes e executá-los progressivamente, assim precavendo que o teste de software seja feito irrefletidamente, perdendo-se tempo e alocando esforço demasiado, além de evitar que erros simples de serem descobertos e corrigidos passem despercebidos. A seção 4.1 detalha a nomenclatura usada neste trabalho. A seção 4.2 aborda o conjunto de condições sob as quais determina-se se um software sob teste satisfaz os requisitos necessários. A seção caracteriza a técnica de teste para software empregada neste trabalho.

4.1. Terminologia básica

Neste trabalho, foi escolhida a nomenclatura definida pela [ISTQB, 2018] (*International Software Testing Qualifications Board*), uma instituição que, resumidamente, define e mantém um corpo de conhecimento sobre boas práticas de teste de software. Seguem alguns dos termos usados neste trabalho.

- **Erro -** Uma ato humano feito enquanto programando que produz um resultado incorreto (popularmente, *bug*).
- **Defeito -** Resultado de um erro, ou em outras palavras; a diferença entre a atual implementação de um produto e a suposição de uma implementação consistente em um estado correto. Um exemplo de defeito pode ser um passo incorreto ou, pior ainda, uma omissão.
- **Falha -** Ocorre quando o código que corresponde ao Defeito (*Fault*) é executado. Logo após isso, o Defeito cruza a fronteira do sistema e torna-se visível ao usuário.

- **Teste -** Corresponde ao ato de exercitar o programa com casos de teste. Um teste possui duas metas distintas: encontrar falhas ou demonstrar a correta execução de um fluxo qualquer. Alternativamente, um teste pode ser usado para provar que um determinado erro está ausente.
- **Caso de Teste -** Possui identidade e é associado ao comportamento de um programa. Engloba um conjunto de entradas pré-definidas e saídas esperadas. Pode ou não exigir pré-condições.

O modelo de ciclo de teste na Figura 11 aponta que no processo de desenvolvimento há três ocasiões em que um defeito poder ser inserido, que são as atividades anteriores ao teste (especificação, projeto e codificação), resultando em um ou mais defeitos que podem ser propagados pelo restante das atividades do processo de desenvolvimento. Durante o teste, o defeito é classificado, isolado e reparado. Apesar da atividade de teste remover um defeito e consequentemente reparar a aplicação, nada impede que novos defeitos sejam inseridos nas atividades posteriores ao teste.

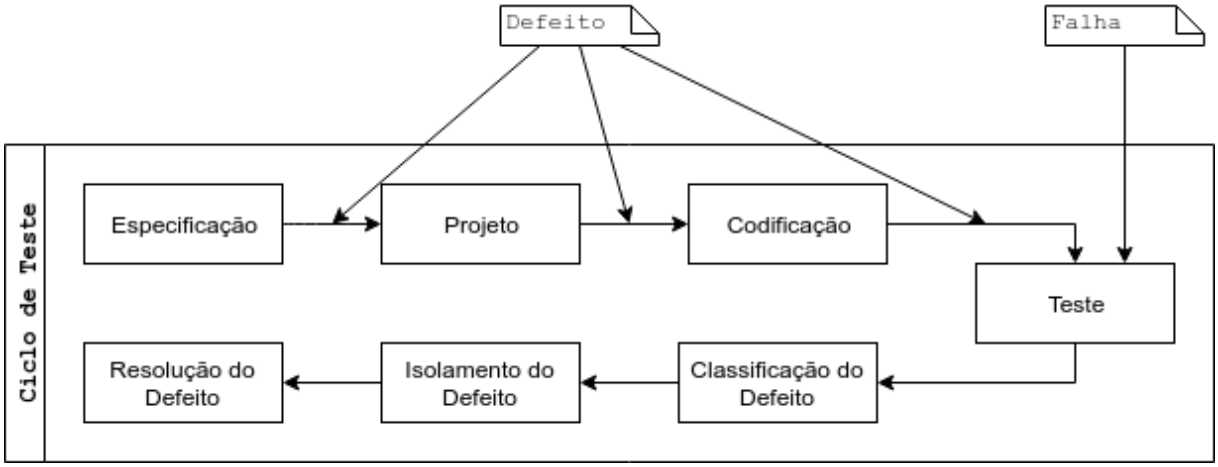

Figura 11: Ciclo de teste de um programa [Jorgensen 2013]

4.2. Caso de Teste

Esta seção descreve um cenário para ilustrar o conceito de caso de teste com base na redação fornecida por [Jorgensen 2013]. Dado um programa e sua especificação, deve-se considerar ***S*** o conjunto de especificações de comportamento e ***P*** o conjunto de comportamentos implementados, como ilustrado na Figura 12. Entre todos os possíveis comportamentos, os esperados estão no círculo intitulado *S* e os comportamentos observados estão no círculo intitulado *P*. Se qualquer comportamento especificado não tiver sido implementado, há um defeito por omissão. Por conseguinte, um programa que se desvia do comportamento desejado, corresponde a um defeito por comissão ou missão. Finalmente, a intersecção corresponde à porção correta.

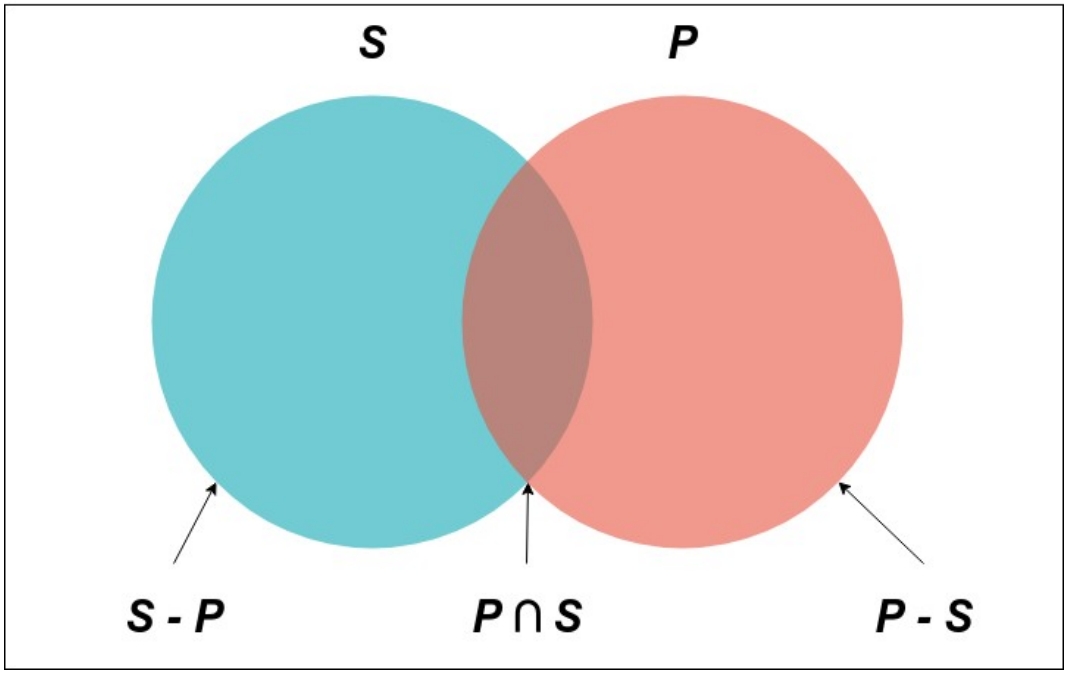

Figura 12: Diferença entre o comportamento especificado (S) e implementado (P).

Agora, sabendo que um caso de teste estressa um comportamento de um programa, deve-se considerar a relação entre os conjuntos **S**, **P** e **T** ilustrado na Figura 13. Há regiões especificadas que não são testadas (2 e 5), regiões especificadas que são testadas (1 e 4) e casos de teste que estressam o

comportamento de regiões não especificadas (3 e 7). Cada um destes domínios é importante, pois se existe um comportamento para o qual não se encontra um caso de teste, o teste está necessariamente incompleto. Se há um caso de teste correspondente a um comportamento não especificado, algumas conclusões podem ser obtidas: o caso de teste é injustificado, a especificação é deficiente ou o testador deseja determinar que um comportamento não especificado não ocorre.

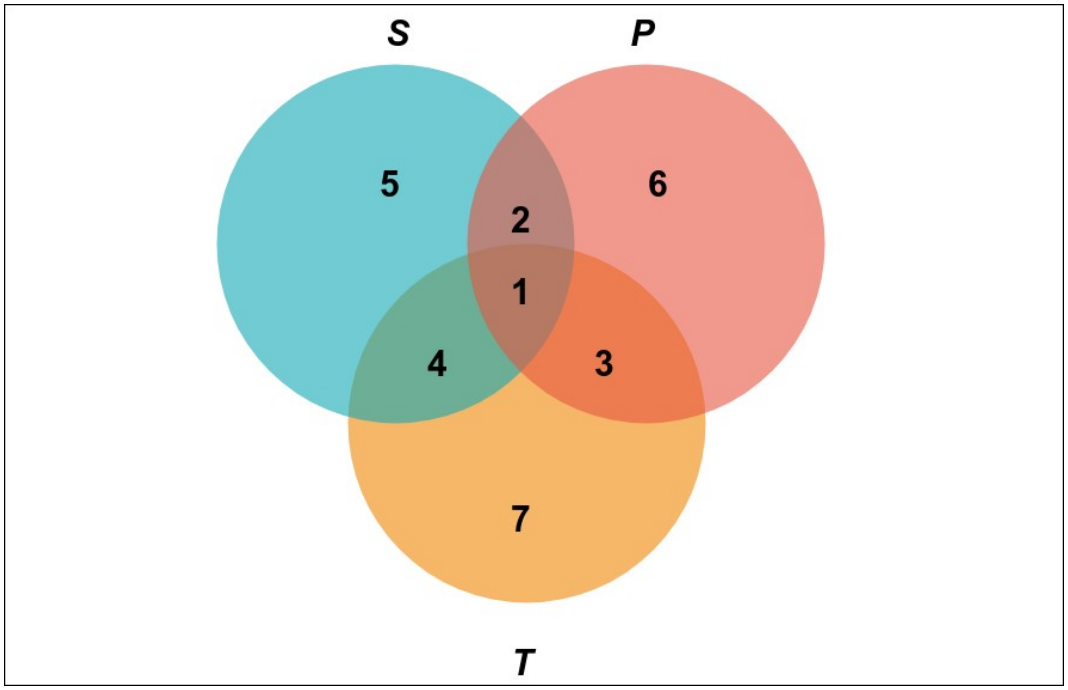

Figura 13: Diferença entre o comportamento especificado (S), implementado (P) e testado (T) .

4.3. Teste baseado em falhas

O Teste baseado em falhas é norteado por informações concebidas sobre erros prosaicos que programadores podem cometer durante o desenvolvimento e em como tratar esta questão com intuito de demonstrar a ausência de tais falhas. Os tipos de erros que programadores podem cometer variam. Interpretar de forma inadequada onde uma preocupação transversal se aplica é um erro comum que pode se manifestar em formas de diferentes tipos de falhas. Originalmente proposta por [Demillo et al. 1978], esta técnica requer que o testador crie um dado de teste

que cause um finito e bem especificado conjunto de defeitos, que resultem em uma falha.

A técnica de teste baseado em falhas mais investigada e aplicada é o Teste de Mutação [Ghani and Parizi 2013; Singhal et al. 2013], que consiste sobretudo quando - após a execução do programa (como é mostrado em detalhes a seguir) são geradas cópias ligeiramente alteradas do programa a ser testado, cada cópia contendo uma única falha advinda da nomenclatura de falhas já citada. Cada cópia é chamada de mutante e espera-se que tenha um comportamento diferente do original.

Portanto, dado um programa *P*, os operadores de mutação encapsulam um conjunto de regras de alteração aplicadas a *P* com a intenção de criar um conjunto de mutantes *M*. Então, para cada *m* pertencente a *M*, deve-se executar um teste *t* pertencente a *T* originalmente desenhado para *P*. Considere que P(*t*) e M(*t*) sejam saídas *P* e *M* em *t*. Então, *M* é um programa equivalente de *P* se, e somente se, P(*t*) = M(*t*) para todos *t* pertencente a *T*. Se m(t) ≠ P(t), o Mutante é considerado *morto*. Caso contrário, deve-se melhorar *T* para que este revele a diferença entre *m* e *P [Demillo et al. 1978].*

O processo interativo de execução do teste mutante pode ser categorizado em quatro passos descritos por [Demillo et al. 1978] e a seguir reproduzidos.

Passo 1: um programa *P* e um conjunto de dados de teste cuja adequação deve ser comprovada são recebidos como entrada; o programa é executado sobre este mesmo conjunto de dados de teste; se o programa fornecer respostas incorretas, certamente o programa contém erros que se manifestam em forma de uma falha (Figura 11). Por outro lado, se o programa fornecer respostas corretas, pode ser que o programa ainda esteja em erro, mas o conjunto de testes não sensibiliza o código de forma suficiente para provocar a manifestação desse erro em forma de falha.

Passo 2: é gerado um conjunto de programas mutantes *M* diferindo apenas na ocorrência de erros triviais.

Passo 3: os mutantes gerados são executados, e, para cada $m \in M$, há apenas duas possibilidades para a imagem de $t \in T$ sob m(t):

a) no qual os dados retornam diferentes resultados a partir de P, m(t) ≠ P(t) ou;
b) no qual os dados retornam os mesmos resultados a partir de P, ou seja, m(t) = P(t).

No caso **a**, é dito que o mutante *m* está morto, pois foi distinguido pelos dados do teste e não é mais necessário permanecer no processo de teste, pois sua falha foi detectada [Offutt and Pan 1998] . No caso **b**, é dito que o mutante *m* está vivo.

Passo 4: análise dos mutantes; um mutante pode estar vivo por duas razões:

a) o conjunto de testes deve ser melhorado, pois não possui sensibilidade suficiente para distinguir (induzir à falha) o erro que deu origem ao mutante *m*.
b) o mutante *m* e *P* são equivalentes e qualquer conjunto de dados de teste não os distinguirá, pois possuem o mesmo comportamento funcional.

A Listagem 3 mostra uma simples função à esquerda que compara dois valores e retorna o menor entre ambos. À direita, cinco operadores de mutação são precedidos e discriminados pela letra grega delta (Δ). O leitor deve perceber que na linha 10, o operador de mutação substitui a variável `i` por `MinVal`. Entretanto a variável `MinVal` recebe o valor da variável i na linha 6, o que torna o operador de mutação na linha 10 equivalente e, por conseguinte, indistinguível do fluxo original do programa. Após os cinco mutantes terem sido executados, cada um pode cair nas duas categorias definidas no passo 3.

Listagem 3: Comparação entre uma função normal e a mesma função modificada.

Programa Original	Programa com operadores Mutantes
1 FUNCTION Min(i,j : Integer)	1 FUNCTION Min(i,j : Integer)
2 RETURN Integer IS	2 RETURN Integer IS
3 MinVal : Integer	3 MinVal : Integer
4	4
5 BEGIN	5 BEGIN
6 MinVal = i;	6 MinVal = i;
7 IF(j < i) THEN	7 Δ MinVal = j;
8 MinVal = j;	8 IF(j < i) THEN
9 END IF;	9 Δ IF(j > i) THEN
10 RETURN (MinVal);	10 Δ IF(j < MinVal) THEN
11 END Min;	11 MinVal = j;
	12 Δ TRAP;
	13 Δ MinVal = i;
	14 END IF;
	15 RETURN (MinVal);
	16 END Min;

Fonte: [Offutt and Pan 1998].

Uma vez aplicados os operadores de mutação, segue-se a análise da interação dos mutantes vivos com o código-fonte da aplicação. Esta análise visa revelar o porquê dos mutantes permanecerem vivos. Após essa análise, espera-se identificar quais casos de teste precisam ser atualizados. A atualização dos casos de teste intenta torná-los sensíveis aos operadores de mutação que geram mutantes vivos. Outro possível resultado da análise dos mutantes vivos é a conclusão de que o programa sob teste contém erros e que o mesmo necessita ser atualizado. Este processo é repetido até que se atinja um resultado satisfatório. Todos estes passos são descritos em detalhes no capítulo 8.

5. MODELO DE FALHAS PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS

No teste baseado em falhas, os requisitos de testes são provenientes de uma nomenclatura composta de erros recorrentes cometidos por programadores. Logo, em um modelo de falhas para programação orientada a aspectos, estes erros devem refletir as características únicas do paradigma. Na seção 5.1 é apresentado um panorama sobre as falhas catalogadas especificamente para o paradigma de orientação a aspectos, bem como as condições necessárias para que a geração de operadores de mutação seja considerada efetiva. A seção 5.2 possui o grupo com maior número de operadores de mutação, esta seção descreve os operadores de mutação para *pointcuts*. As seções 5.3 e 5.4 descrevem operadores de mutação para *advice* e Introduções, respectivamente.

5.1. Taxonomia de falhas para software orientado a aspectos

Esta seção investiga a taxonomia de erros específicos para programas orientados a aspectos. Entretanto, posteriormente, se deve definir características gerais a serem satisfeitas com a intenção produzir programas mutantes sob certos critérios. Tais condições almejam garantir que um defeito encapsulado por um operador de mutação possa ser notado, em forma de uma falha, após a execução de um programa [Offutt and Pan 1998]. Nesta seção, é apresentado um excerto da taxonomia de falhas, refinada e catalogada, em trabalhos de pesquisa feitos pelo professor Fabiano Ferrari, onde [Ferrari et al. 2015] é a referência usada. As tabelas 6 - 9, sumarizam e descrevem os tipos de defeitos de acordo com a estrutura ao qual pertencem.

Tabela 6. Defeitos relacionados a descritores de *pointcuts*.

Identificador	Descrição
D1.1	Seleção de um superconjunto de um conjunto de pontos de interesse pretendido.
D1.2	Seleção de um subconjunto de um conjunto de pontos de interesse pretendido.
D1.3	Seleção incorreta de um conjunto de pontos de interesse, o qual inclui pontos pretendidos e não pretendidos.
D1.4	Seleção incorreta de um conjunto de pontos de interesse, o qual inclui apenas pontos não pretendidos.
D1.5	Uso incorreto de *pointcuts* predefinidos pela linguagem orientada a aspectos.
D1.7	Correspondência (**matching**) incorreta de pontos de interesse baseada em um padrão de lançamento exceções.
D1.8	Correspondência incorreta de pontos de interesse baseada em uma circunstância dinâmica.

Fonte: [Ferrari et al. 2015].

Tabela 7: Defeitos relacionados a advice

Identificador	Descrição
D2.1	Especificação incorreta do tipo de *advice*.
D2.2	Fluxo de controle ou de dados incorreto devido à interação não intencionada entre aspectos.
D2.3	Lógica de *advice* incorreta, resultado da violação de invariantes ou falha ao estabelecer pós-condições.
D2.4	Laço infinito resultante da interação entre mais de um *advice*.
D2.5	Acesso incorreto à informação do *join point*.
D2.6	*Advice* incorretamente vinculado ao *pointcut*.

Fonte: [Ferrari et al. 2015].

Tabela 8. Defeitos relacionados a declarações de introdução.

Identificador	Descrição
D2.1	Introdução incorreta de um método, resultando em uma substituição imprevista ou não resultando em uma substituição esperada.
D3.2	Introdução de um método em uma classe incorreta.
D3.3	Alteração inconsistente na hierarquia de classes por meio de declaração de herança, resultando em comportamento não intencional.
D3.4	Introdução incorreta de um método, resultado em uma substituição inesperada.
D3.5	Omissão na declaração de interface ou introdução de uma interface corrompendo a identidade do objeto.
D3.6	Alterações incorretas no fluxo de controle de exceções resultantes da interação entre classe e aspecto ou de cláusulas que alteram a gravidade da exceção.
D3.7	Equívoco ou omissão na precedência de expressões composta por aspectos.
D3.8	Equívoco nas regras de instanciação ou emprego de aspectos, resultando em instanciação não intencionada de aspectos.
D3.9	Equívoco na definição de políticas de imposição de avisos e erros.

Fonte: [Ferrari et al. 2015].

Tabela 9. Defeitos relacionados ao programa base.

Identificador	Descrição
D4.1	O programa base não oferece os *join points* necessários, nos quais um ou mais aspectos foram projetados para serem aplicados.
D4.2	A evolução do programa causa quebra no *pointcut.*
D4.3	Outros problemas relacionados ao programa base, tais como refatoração incorreta ou código transversal duplicado.

Fonte: [Ferrari et al. 2015].

5.1.1. Critérios de efetividade

O Teste baseado em restrições (*Constraint-based testing* - CTB) usa restrições para a geração dos dados de teste e pode ser entendido como uma expressão algébrica que limita o intervalo de entrada do programa a fim de satisfazer alguma regra, como, por exemplo ($x > 0$). Neste cenário, CTB representa a condição em que o mutante deverá morrer (ou para um defeito produzir uma falha). Se um teste matar um mutante, o conjunto de restrições é satisfeito pelo caso de teste e este é dito efetivo por fazer o programa mutante manifestar uma falha. Do contrário, se o conjunto de restrições não puder ser satisfeito, o teste é considerado ineficaz, pois caso de teste algum pode matar o mutante e, por consequência, o mutante é considerado equivalente ao programa original [Offutt and Pan 1998].

Considere agora o seguinte caso, dado um programa *P,* um operador de mutação engloba uma série de modificações a fim de gerar um conjunto de mutantes *M.* Então para cada $m \in M$, um teste $t \in T$ aplicado a *m* precisa ter estas três características:

Acessibilidade: Um programa mutante é representado por uma mudança sintática simples enquanto o resto das instruções permanecem sintaticamente iguais ao do programa original, o requisito mínimo para um teste matar um mutante é que este execute o operador mutante.

Necessidade: Para um teste matar um mutante, é preciso fazer o mutante adquirir um estado inconsistente quando este alcança o operador de mutação, ou seja, manifestar uma falha.

Suficiência: O estado final de *M* compulsoriamente deve diferir do estado final de P.

Considerando as já mencionadas particularidades estruturais da programação orientada a aspectos, a seguir são definidos os critérios essenciais de efetividade para os principais grupos dentro da linguagem AspectJ, segundo [Ferrari et al. 2013; Lindström et al. 2017].

Defeitos relacionados a *pointcuts*: a diferença entre um ponto de interesse desejado e um ponto de interesse atualmente capturado resulta na aplicação de um comportamento alternativo em resposta à captura deste ponto de interesse indesejado. Isto também equivale a dizer que houve uma falha em aplicar um comportamento em resposta a um ponto de interesse intencionado. A Figura 14 ilustra quatro tipos de defeito [Lemos et al. 2006]. A área vermelha representa o conjunto de pontos de interesse de fato selecionado pelo *pointcut*, enquanto a área azul representa o que deveria ter sido selecionado caso o *pointcut* estivesse correto. Ainda na Figura 14, um *pointcut* pode ser descrito como segue.

a) Seleção de um conjunto de pontos de interesse qualquer que possui uma interseção com o conjunto de pontos de interesse desejado.
b) Seleção de um conjunto de pontos de interesse qualquer, porém sem interseção alguma como o conjunto de pontos de interesse desejado.
c) Seleção de um conjunto que contém o conjunto de pontos de interesse desejado.
d) Seleção de um conjunto que é composto unicamente de partes do conjunto de pontos de interesse desejado.

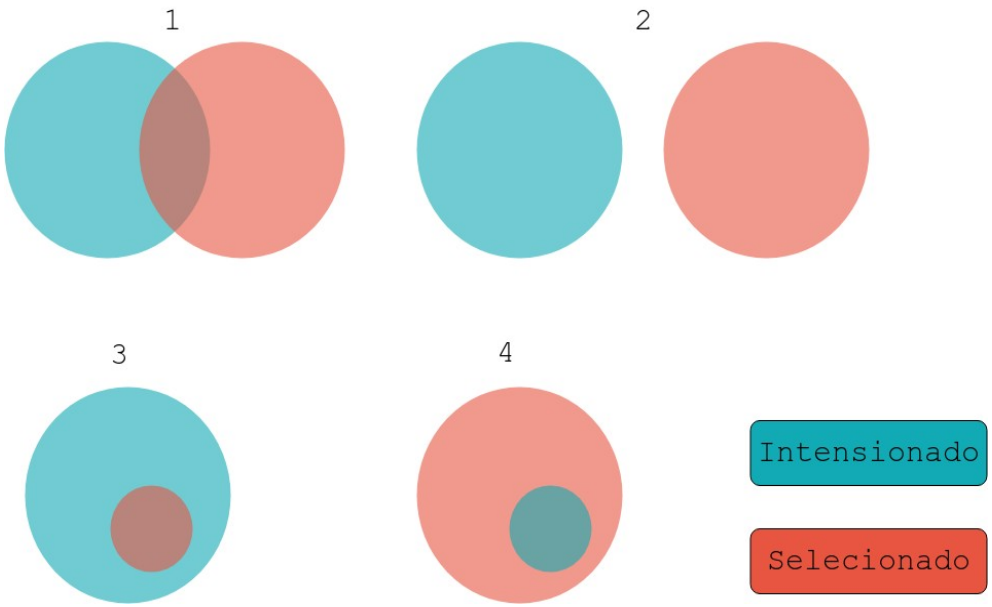

Figura 14: Falhas relacionadas aos descritores de *pointcuts*.

Defeitos relacionadas a *advice*: neste caso, os defeitos podem ocorrer tanto no corpo do método quanto em sua assinatura. Para estes defeitos, há três condições necessárias: o defeito é dito Acessível se há, ao menos, um caminho realizável no fluxo de execução que leve até o *advice;* a ativação do *advice* satisfaz a condição de Necessidade*,* desde que no fluxo de execução o operador mutante seja executado; e a Suficiência é satisfeita quando um estado inconsistente por efeito da execução de um *advice* se propaga até o fim do fluxo de execução.

Defeitos relacionadas a declarações*:* para inserções incorretas como sobrescrever ou implementar um método erroneamente, as três condições permanecem válidas: os defeitos podem ser acessados produzindo um estado inconsistente que se propaga até o fim do fluxo de execução. Como exemplo, uma aresta pode estar vinculada ao nó errado.

5.1.2. Operadores de mutação para AspectJ

Baseado nas categorias de defeito específicas que ocorrem em programas orientados a aspectos apresentadas nas seções anteriores, as próximas seções exemplificam como um operador de mutação encapsula tais defeitos e os mutantes que resultam desta mudança sintática. Esta mudança introduzida pelo operador de mutação objetiva simular erros comuns a fim de encontrar falhas mais complexas de forma mais simples e rápida. Os operadores são categorizados em três grupos

distintos que representam defeitos relacionados às três classes estruturais implementadas pela linguagem AspectJ.

5.2. Operadores de mutação para *pointcuts*

Esta subseção apresenta a categorização e exemplificação de operadores de mutação baseada na redação feita por [Ferrari et al. 2011]. Operadores de mutação para *pointcuts* focam-se em seus descritores, pois, como já visto sucintamente na seção anterior, e mais especificamente descrito por [Baekken 2006], durante a seleção de um ponto de interesse, há dois resultados possíveis ao se avaliar a expressão referente a um ponto exposto qualquer; ou o ponto exposto é selecionado, ou não é.

5.2.1. Fortalecimento de *pointcuts* (PCS)

Este operador reduz a abrangência de seleção de pontos de interesse em comparação com a abrangência originalmente especificada pelo descritor do *pointcut* antes de aplicado o operador de mutação, como mostrado na Figura 14, conjunto 3. Logo, os pontos de interesse de fato selecionados são um subconjunto do conjunto capturado pelo *pointcut* original. Dado um *pointcut*, a Tabela 10 descreve quais operadores podem ser aplicados dentro desta categoria para reduzir o conjunto de pontos de interesse capturado pelo *pointcut* após a aplicação do operador de mutação.

Tabela 10. Descrição de operadores relacionados ao fortalecimento de *pointcut*.

Operador	Descrição
PSSR (*Pointcut Strengthening by Subtype Replacement*)	Substituição de uma classe Java ou classe definida pelo usuário por sua subclasse imediata, se existente.
PSWR (*Pointcut Strengthening by Wildcard Remova*l)	Remoção de coringas do pacote, método, construtor ou campo.
PSDR (*Pointcut Strengthening by Declare Annotations Removal*)	Remoção de anotações do tipo declare dos aspectos.

Fonte: [Ferrari et al. 2011].

5.2.2. Enfraquecimento de *pointcuts* (PCW)

Este operador expande a abrangência de seleção de pontos de interesse em comparação com a abrangência originalmente especificada pelo descritor de *pointcut* antes de aplicado o operador de mutação, como mostrado na Figura 14, conjunto 4. Logo, os pontos de interesse capturados pelo *pointcut* original são um subconjunto do conjunto capturado pelo *pointcut* mutante. Dado um *pointcut*, a Tabela 11 descreve quais operadores podem ser aplicados dentro desta categoria para expandir o conjunto de pontos de interesse capturado pelo *pointcut* após a aplicação do operador de mutação.

Tabela 11: Descrição de operadores relacionados ao enfraquecimento de pointcuts

Operador	**Descrição**
PWSR (*PointcutWeakening by Supertype Replacement*)	Substituição de uma classe mais específica por sua superclasse ou sua implementação mais geral em um *pointcut*.
PWIW (*Pointcut Weakening by Insertions of Wildcards*)	Inserção de coringas em pacotes, métodos, construtores ou campos. Este operador também substitui modificadores por coringas.
PWAR (*Pointcut Weakening by Annotations Removal*)	Remoção de anotações presentes em um *pointcut*.

Fonte: [Ferrari et al. 2011].

5.2.3. Substituição de *pointcuts* (PCR)

Este operador altera a abrangência de seleção originalmente especificada no descritor do *pointcut,* resultando em sobreposição e disjunção de pontos de interesse, como ilustrado na Figura 14, conjuntos 1 e 2. Este operador também pode criar os cenários de seleção equivocada de subconjuntos ou superconjuntos caso um dos *pointcuts* selecione um subconjunto de pontos de interesse capturado por outro *pointcut*. Dado um *pointcut*, a Tabela 12 descreve quais operadores podem ser aplicados dentro desta categoria para sobrepor ou isolar o conjunto de pontos de interesse capturado pelo *pointcut* após a aplicação do operador de mutação.

Tabela 12. Descrição de operadores relacionados à substituição de *pointcuts*.

Operador	Descrição
PCGS (*Pointcut Changing by get-set Replacement*)	Substituição de um *pointcut get* por *set*, e vice-versa.
PCCR (*Composition Participant Pointcut Replacement*)	Substituição de um *pointcut* específico, onde este faz parte da descrição de um outro *pointcut* ou que é usado como parâmetro de correspondência no fluxo de controle.
PCLO (*Composition Logical Operators Replacement*)	Substituição do operador lógico (&& ou ||) presente na composição de regras.
PCCC (*Control Flow Scope Changing Description*)	Substituição de um *pointcut* cflow por cflowbelow, e vice-versa.

Fonte: [Ferrari et al. 2011].

Com o operador PCCC, espera-se uma mudança na execução do conjunto de *advice*. A Listagem 4 apresenta um exemplo de como este operador pode ser empregado.

Listagem 4: Exemplo de aplicação do operador mutante PCCC.

Programa Original

```
1 public aspect PCCCGeneric {
2     pointcut PointcutA ( ) :call(*SomeClass.SomeMethod(..));
3     pointcut PointcutB ( ) :call(*SomeClass.AnotherMethod(..));
5     . . .
6     pointcut PointcutComposition ( ): PointcutA && cflow(PointcutB());
7 }
```

Programa com Operador Mutante

```
1 public aspect PCCCGeneric {
2     pointcut PointcutA ( ) :call(*SomeClass.SomeMethod(..));
3     pointcut PointcutB ( ) :call(*SomeClass.AnotherMethod(..));
5     . . .
6 Δ pointcut PointcutComposition (): PointcutA && cflowbelow(PointcutB());
7 }
```

Fonte: [Ferrari et al. 2011].

5.2.4. Enfraquecimento ou fortalecimento de *pointcuts* (PCR)

Este operador reduz ou expande a abrangência de seleção de pontos de interesse em comparação com a abrangência originalmente especificada pelo descritor do *pointcut* antes de aplicado o operador de mutação, produzindo o resultado vistos na Figura 14, conjuntos 3 e 4. Dado um *pointcut*, a Tabela 13 descreve quais operadores podem ser aplicados para reduzir ou expandir o conjunto de pontos de interesse capturado pelo *pointcut* após aplicação do operador de mutação.

Tabela 13. Descrição de operadores relacionados ao possível enfraquecimento ou fortalecimento de *pointcuts*.

Operador	**Descrição**
POPL (*PointcutWeakening or Strengthening by Parameter List Changing*)	Substituição, remoção ou inserção de coringas da lista de parâmetros em um *pointcut*.
POAC (*Pointcut Weakening or Strengthening by After Clause Changing*)	Inserção, alteração ou remoção da cláusula de *returning* ou *throwing* da definição de um *advice* executado após um ponto de interesse.
POEC (*Pointcut Weakening or Strengthening by Exception Throwing Clause Changing*)	Altera o tipo de exceção lançada.

Fonte: [Ferrari et al. 2011].

5.3. Operadores de mutação para *advice*

Os operadores mutantes para *advice* abordam defeitos onde os mesmos são ignorados ou vinculados ao *pointcut* errado. Estes operadores implementam mudanças sintáticas com o propósito de, segundo Ferrari [Ferrari et al. 2011]: alterar o momento em que um *advice* é executado; ativar ou desativar a execução de um *join point;* alterar a origem da informação estática presente no interior do *advice;* negar a execução de um *advice* por meio de sua remoção e substituir o *pointcut* vinculado ao *advice*.

5.3.1. Substituição do tipo do *advice* (ABAR)

Este operador altera o momento em que um *advice* é executado, variando entre antes e depois do ponto desejado. Como resultado, pode haver possíveis falhas e estados inconsistentes devido à mudanças no fluxo de execução. A Listagem 5 mostra um exemplo de como este operador pode ser empregado.

Listagem 5: Exemplo de aplicação do operador mutante ABAR.

Programa Original

```
1 public aspect ABARGeneric {
2     pointcut abarPointcut ( ) :call(*SomeClass.SomeMethod(..));
3     before( ) : abarPointcut( ) {
5     . . .
6     }
7 }
```

Programa com Operador Mutante

```
1 public aspect ABARGeneric {
2     pointcut abarPointcut ( ) :call(*SomeClass.someMethod(..) );
3 Δ   after( ) : abarPointcut( ) {
4     . . .
5     }
6 Δ   after() returning : abarPointcut(){
7     . . .
8     }
9 Δ   after() throwing: abarPointcut() {
10    . . .
11    }
12 }
```

Fonte: [Ferrari et al. 2011].

5.3.2. Remoção de declaração do a*dvice* (APSR)

Este operador remove a declaração `proceed()`, uma após a outra. A menos que essa declaração seja invocada, o ponto de interesse será ignorado. Portanto, a supressão desta declaração resulta em execução parcial, provavelmente orientando a um estado inconsistente. A Listagem 6 mostra um exemplo de como este operador pode ser empregado.

Listagem 6: Exemplo de aplicação do operador mutante APSR.

Programa Original

```
1 public aspect APSRGeneric {
2     pointcut apsrPointcut ( ) :call (* SomeClass. SomeMethod(..));
3     around( ) : apsrPointcut( ) {
4     . . .
5         proceed();
6     . . .
7     }
8 }
```

Programa com Operador Mutante

```
1 public aspect APSRGeneric {
2     pointcut apsrPointcut ( ) :call (* SomeClass.SomeMethod(..));
3     around( ) : apsrPointcut( ) {
4     . . .
5 Δ   // proceed();
6     . . .
7     }
8 }
```

Fonte: [Ferrari et al. 2011].

5.3.3. Execução de declaração do a*dvice* (APER)

Este operador capacita a execução da declaração `proceed()` por meio da remoção das condições de guarda que a ladeiam. Caso a declaração `proceed()` apareça dentro de um bloco aninhado de condicionais, a operação de remoção é aplicada incrementalmente. Como resultado, espera-se a execução de um comportamento não intencionado, que pode levar a um estado inconsistente. A Listagem 7 mostra um exemplo de como este operador pode ser empregado.

Listagem 7: Exemplo de aplicação do operador mutante APER.

Programa Original

```
1 public aspect APERGeneric {
2     pointcut aperPointcut ( ) : call(*SomeClass.SomeMethod(..));
4     around( ) : aperPointcut( ) {
5     if (aerPredicado)
6             proceed()
7     }
8}
```

Programa com Operador Mutante

```
1 public aspect APERGeneric {
2     pointcut aperPointcut ( ) : call(*SomeClass.SomeMethod(..));
4     around( ) : aperPointcut( ) {
5 Δ   if (true)
6             proceed()
7     }
8}
```

Fonte: [Ferrari et al. 2011].

5.3.4. Mudança de informação estática (AJSC)

Segundo [Laddad 2009], em um *advice* há três variáveis disponíveis. Estas variáveis contém informações sobre o ponto de interesse que recebe a ação do *advice*, como o nome do método, o objeto `this` e os argumentos do método. O objeto `thisJoinPoint`, caso usado em um *advice*, é alocado toda vez que um advice é executado para capturar o atual contexto dinâmico. Em contraposição, `thisJoinPointStaticPart` é alocado apenas uma vez por ponto de interesse durante a execução de um programa. A última variável, `thisEncloingJoinPointStaticPart,` guarda as informações estáticas sobre o ponto de interesse de inclusão, que também é referenciado como o contexto de inclusão. O contexto de inclusão de um ponto de interesse varia de ponto de interesse para ponto de interesse.

O operador AJSC substitui as variáveis disponíveis em cada *advice* que carrega informação sobre o ponto de interesse atualmente capturado. Neste caso, o operador substitui `thisJoinPointStaticPart` por `thisEnclosingJoinPointStaticPart` e vice-versa. Como resultado, espera-se o acesso às informações incorretas em cada *advice*. A Listagem 8 detalha um exemplo de como este operador pode ser empregado.

Listagem 8: Exemplo de aplicação do operador mutante AJSC.

Programa Original

```
1 public aspect AJSCGeneric {
2     pointcut ajscPointcut( ):call(*SomeClass.SomeMethod(..));
3     Object around( ) : ajscPointcut( ) {
4     long  start = System.nanoTime();
5     Object ret = proceed();
6     long end = System.nanoTime();
7     long total = end-start;
8    System.out.print(thisJoinPointStaticPart.getSignature() + "took"
9         + total + " nanoseconds");
10    return ret;
11    }
12 }
```

Programa com Operador Mutante

```
1 public aspect AJSCGeneric {
2     pointcut ajscPointcut( ):call(*SomeClass.SomeMethod(..));
3     Object around( ) : ajscPointcut( ) {
4     long  start = System.nanoTime();
5     Object ret = proceed();
6     long end = System.nanoTime();
7     long total = end-start;
8Δ System.out.print(thisEnclosingJoinPointStaticPart.getSignature()+"took"
9         + total + " nanoseconds");
10    return ret;
11    }
12 }
```

Fonte: [Ferrari et al. 2011].

5.3.5. Prejuízo pela remoção de um *advice* (ABHA)

Este operador remove um *advice* implementado impedindo a execução do comportamento transversal dinâmico. Assim, espera-se que um dado comportamento omitido não seja executado. A Listagem 9 detalha um exemplo de como este operador pode ser empregado.

Listagem 9: Exemplo de aplicação do operador mutante ABHR.

Programa Original

```
1 public aspect ABHAGeneric {
2     pointcut abhaPointcut ( ) :call(*SomeClass.SomeMethod(..));
4
5     before( ) : abhaPointcut( ) {
6     …
7     }
8 }
```

Programa com Operador Mutante

```
1 public aspect ABHAGeneric {
2     pointcut abhaPointcut ( ) :call(*SomeClass.SomeMethod(..));
4
5 Δ   //before( ) : abhaPointcut( ) {…}
6 }
```

Fonte: [Ferrari et al. 2011].

5.3.6. Substituição de um *pointcut* vinculado a um *advice* (ABPR)

Estes operadores modelam defeitos relacionados a substituição não intencionado de um *pointcut* vinculado a um *advice* por outro *pointcut* definido no mesmo aspecto, resultando em um comportamento não intencionado. A Listagem 10 detalha um exemplo de como este operador pode ser empregado.

Listagem 10: Exemplo de aplicação do operador mutante ABPR

Programa Original

```
1 public aspect ABPRGeneric {
2   before( ) : PointcutA(){...};
3
4   after( ) : PointcutB(){...};
5     ...
6}
```

Programa com Operador Mutante

```
1 public aspect ABPRGeneric {
2 Δ before( ) : PointcutB(){...};
3
4 Δ after( ) : PointcutA(){...};
5   ...
6}
```

Fonte: [Ferrari et al. 2011].

5.4. Operadores de mutação para declarações

Estes operadores modelam defeitos relacionados a fluxo não intencionado devido à inserção de novas estruturas, resultando em um estado final errôneo. Estes operadores implementam mudanças sintáticas com o propósito de, segundo Ferrari [Ferrari et al. 2011]: modificar a precedência entre aspectos; alterar a severidade de exceções; remover declarações de erros e avisos e modificar regras relacionadas à instanciação de aspectos.

5.4.1. Mudança na declaração de precedência (DAPC)

Este operador altera a precedência entre dois ou mais aspectos, resultando em ordem de execução de um *advice* sobre pontos de interesse comuns não intencionada. Em razão do grande número de possibilidades resultantes da combinação entre os *n* aspectos presentes no momento de aplicação deste operador de mutação, O(n!), este número é restringido a no máximo 4 ou 5 variações deste mesmo operador. A Listagem 11 mostra um exemplo de como este operador pode ser empregado.

Listagem 11: Exemplo de aplicação do operador mutante DAPC.

Programa Original

```
1 public aspect DAPCGeneric {
2     declare precedence: Aspect A, Aspect B, Aspect C;
3
4           ...
5}
```

Programa com Operador Mutante

```
1 public aspect DAPCGeneric {
2 Δ declare precedence: Aspect C, Aspect A, Aspect B;
3
4 Δ declare precedence: Aspect B, Aspect A, Aspect C;
5
6 Δ declare precedence: Aspect C,Aspect B, Aspect A;
7    ...
8}
```

Fonte: [Ferrari et al. 2011].

5.4.2. Omissão na declaração de precedência (DAPO)

Este operador omite a declaração de precedência entre aspectos, resultando em ordem arbitrária de execução de *advice* sobre pontos de interesse comuns. A Listagem 12 mostra um exemplo de como este operador pode ser empregado.

Listagem 12: Exemplo de aplicação do operador mutante DAPO.

Programa Original

```
1 public aspect DAPOGeneric {
2     declare precedence: Aspect A,Aspect B;
3}
```

Programa com operador Mutante

```
1 public aspect DAPOGeneric {
2 Δ    //declare precedence: Aspect A,Aspect B;
3}
```

Fonte: [Ferrari et al. 2011].

5.4.3. Remoção na declaração de precedência (DSSR)

Exceções declaradas como leves não mais requerem imediata implementação de tratamento. Assim, a declaração *soft* permite que uma exceção do tipo *Checked* seja tratada como *Unchecked*. Este operador remove a declaração do tipo 'leve' (*soft*), possivelmente resultando em um fluxo de controle não intencionado. A Listagem 13 detalha um exemplo de como este operador pode ser empregado.

Listagem 13: Exemplo de aplicação do operador mutante DSSR.

Programa Original

```
1 public aspect DSSRGeneric {
2   pointcut dapcPointcut ( ) :call (* SomeClass. SomeMethod(..));
4
5     declare soft: anExcepion : dapcPointcut ( )
6}
```

Programa com operador Mutante

```
1 public aspect DSSRGeneric {
2   pointcut dapcPointcut ( ) :call (* SomeClass. SomeMethod(..));
4
5 Δ  //declare soft: anExcepion : dapcPointcut ( )
6}
```

Fonte: [Ferrari et al. 2011].

5.4.4. Mudança na declaração de precedência (DEWC)

Este operador altera a declaração de erro e de aviso, resultado em um possível fluxo de execução não intencionado. Este operador funciona removendo a declaração de erro e de aviso, um por vez, e os substitui por declarações de avisos e erros, ou vice-versa. A Listagem 14 apresenta um exemplo.

Listagem 14: Exemplo de aplicação do operador mutante DEWC.

Programa Original

```
1 public aspect DEWCGeneric {
2    pointcut dewcPointcut ( ) :call(*SomeClass.SomeMethod(..));
4
5     declare error: dewcPointcut ( ): "some message"
7}
```

Programa com operador Mutante

```
1 public aspect DEWCGeneric {
2    pointcut dewcPointcut ( ) :call(*SomeClass.SomeMethod(..));
4
5   Δ declare warning: dewcPointcut ( ): "some message"
6}
```

Fonte: [Ferrari et al. 2011].

5.4.5. Mudança na cláusula de instanciação (DAIC)

Este operador alterar as cláusulas de instanciação de um aspecto, resultando em um possível estado de inconsistência de um objeto ou de um aspecto. Este operador funciona removendo ou substituindo uma cláusula por outra, esta última variando entre *perthis*, *pertarget*, *percflow* e *percflowbelow.* A Listagem 15 apresenta um exemplo deste operador.

Listagem 15: Exemplo de aplicação do operador mutante DAIC

Programa Original

```
1 public aspect DAICGeneric pertarget ( aPointcut ( ) ) {...}
```

Programa com Operador Mutante

```
1 Δ public aspect DAICGeneric perthis ( aPointcut ( ) )
2 Δ public aspect DAICGeneric percflow ( aPointcut ( ) )
3 Δ public aspect DAICGeneric percflowbelow ( aPointcut ( ) )
```

Fonte: [Ferrari et al. 2011].

Nem todos os operadores mutantes gerados são úteis. Alguns dos operadores descritos anteriormente podem não ser compiláveis (chamados de anômalos) ou são sintaticamente incorretos. Como por exemplo, operadores que alteram cláusulas que coletam contexto que é usado no corpo de um *advice* específico, por via de regra, não são compiláveis.

5.5. Ferramenta de Automação de Teste de mutação

A disponibilidade de ferramentas de automação é uma questão importante a ser ponderada por desenvolvedores de software antes incorporar Teste de Mutação ao processo de software a fim de se garantir que a tarefa de teste seja adequadamente executada e a relação de custo/benefício seja satisfatória. Esta subseção aborda esta questão introduzindo a ferramenta de automação para Testes de Mutação *Proteum/AJ.* Esta ferramenta para programas Java escrita em AspectJ [Ferrari et al. 2010] implementa os passos elaborados por [Demillo et al. 1978] e descritos na seção .

5.5.1. Proteum/AJ

A ferramenta Proteum/AJ é usada neste trabalho por já ter sido usada para avaliação de aplicações orientadas a aspectos sob Teste de Mutação e pela facilidade de acesso à ferramenta e de contato com o desenvolvedor da mesma. O processo de teste é descrito em resumo como segue e também é ilustrado na Figura 15.

- **Pré-processamento da aplicação original -** a ferramenta recebe um arquivo compactado e um conjunto de casos de teste *JUnit*. Como artefato deste pré-processamento, a ferramenta devolve um arquivo descompactado e uma lista com os aspectos alvos.
- **Execução da aplicação original -** a aplicação descompactada é enviada para o módulo de teste (*Test Runner*) junto com os arquivos de caso de teste. O módulo de teste executa a aplicação sobre o conjunto de teste disponível, o que produz como artefato de saída um XML contendo o resultado dos testes.
- **Geração dos mutantes -** O Motor de Mutação (*Mutation Engine*) recebe a lista com os aspectos alvos e o conjunto de operadores de mutação como entrada e produz os mutantes como saída. Estes mutantes são então enviados para o Compilador de Mutação (*Mutant Compiler*).
- **Execução dos mutantes -** cada mutante é enviado para Compilador de Mutação que então invoca o compilado AspectJ [Ajc, 2018]. O Compilador Mutante detecta mutantes não compiláveis e os classifica como anômalos

(*stillborn*). Para mutantes compiláveis, o módulo de teste os executa sobre o conjunto atual de testes usando a tarefa iajc do Ant. Os resultados são armazenados e enviados para o módulo Avaliador de Testes. O avaliador contrasta os resultados com os resultados originais e identifica quais mutantes devem ser mortos.

- **Análise dos mutantes -** por fim, o Módulo de Análise de Mutantes cria relatórios de texto que mostram a diferença entre o código original e o código alterado, ressaltando os trechos de código para cada mutante.

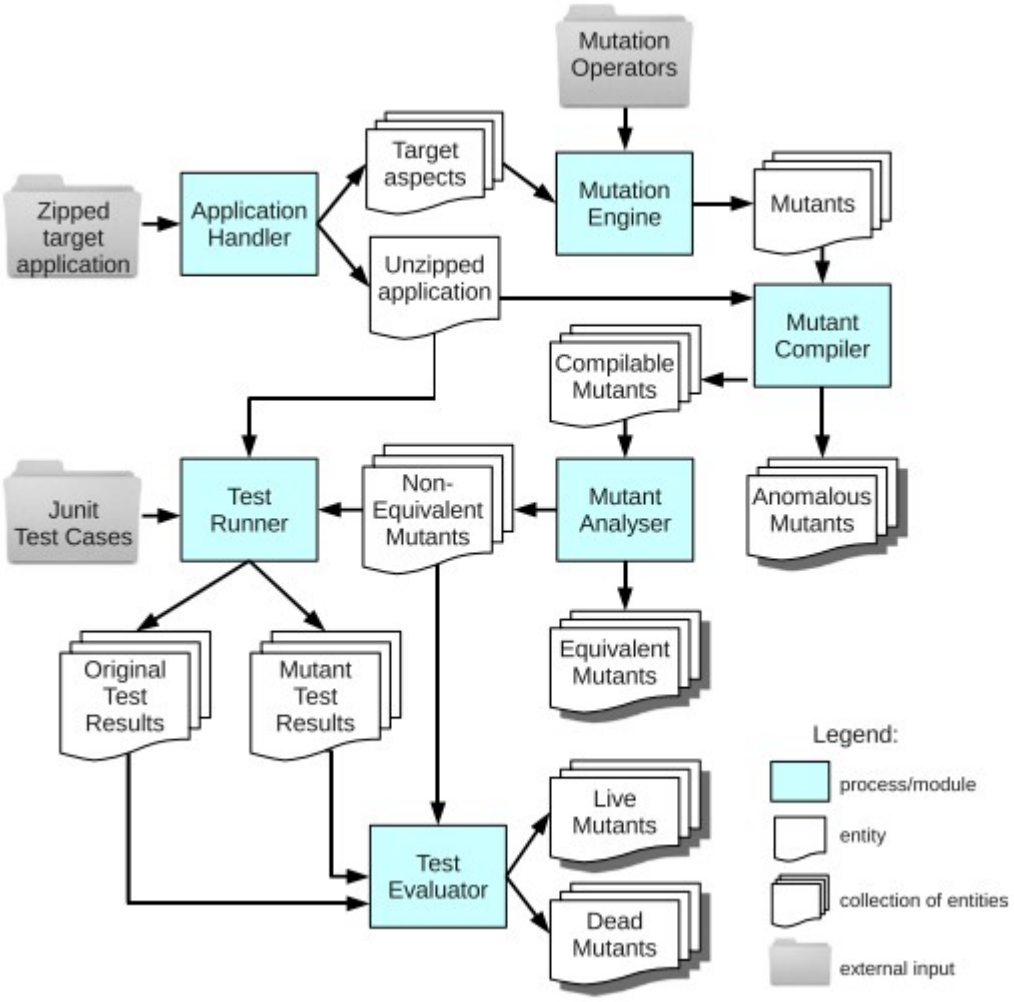

Figura 15: Fluxo de execução. [Ferrari et al. 2013]

6. REFATORAÇÃO[3]

Refatoração, como definido por [Fowler 1999], refere-se a reestruturação de um software como a finalidade de melhorar sua estrutura interna, embora sem alterar seu comportamento externo. A atividade de refatoração busca melhorar as características não funcionais de um software, como: modularidade, testabilidade inteligibilidade e outras. Este capítulo aborda a refatoração de um software originalmente orientado a objetos usando o paradigma de orientação a aspectos, ou seja, o encapsulamento das preocupações transversais do software usado como objeto de estudo em um aspecto. O processo de refatoração foi orientado ao incremento de legibilidade do código que trata das regras de negócio (ou preocupações centrais) e separação de responsabilidades. Na seção 6.1 é brevemente descrita a aplicação que passou pelo processo de refatoração. A seção 6.2 expõe a estrutura a ser usada como padrão para detalhar o processo de refatoração. A seção 6.3 traz a motivação para o processo de refatoração escolhido e os tipos aplicados ao código-fonte bem como o resultado de cada tipo de refatoração.

6.1. Sistema de Banco de Dados

Em princípio, foi avaliado a possibilidade de fazer uso, como objeto de experiência, um projeto no Estágio Profissional do autor. Porém, por motivos de tempo, disponibilidade, complexidade e permissão de acesso ao código fonte, esta escolha se mostrou estéril. Ciente destas dificuldades, a aplicação usada em alternativa é um programa originalmente construído na linguagem Java implementada por [Santos 2014], escolhida especialmente pela didática e clareza com que a aplicação é descrita pelo autor. A aplicação apresentada realiza o cadastro de algumas entidades básicas que posteriormente são mantidas persistentes por meio de um banco de dados MySQL, como mostra a Figura 16.

---

3 **Apesar do intento principal da refatoração ser a reestruturação interna, porém sem alteração do comportamento externo, há mudanças relacionadas à extração do tratamento exceções que fogem desta norma. Principalmente em relação ao comportamento externo da aplicação, uma vez que mensagens mais específicas são exibidas ao usuário.**

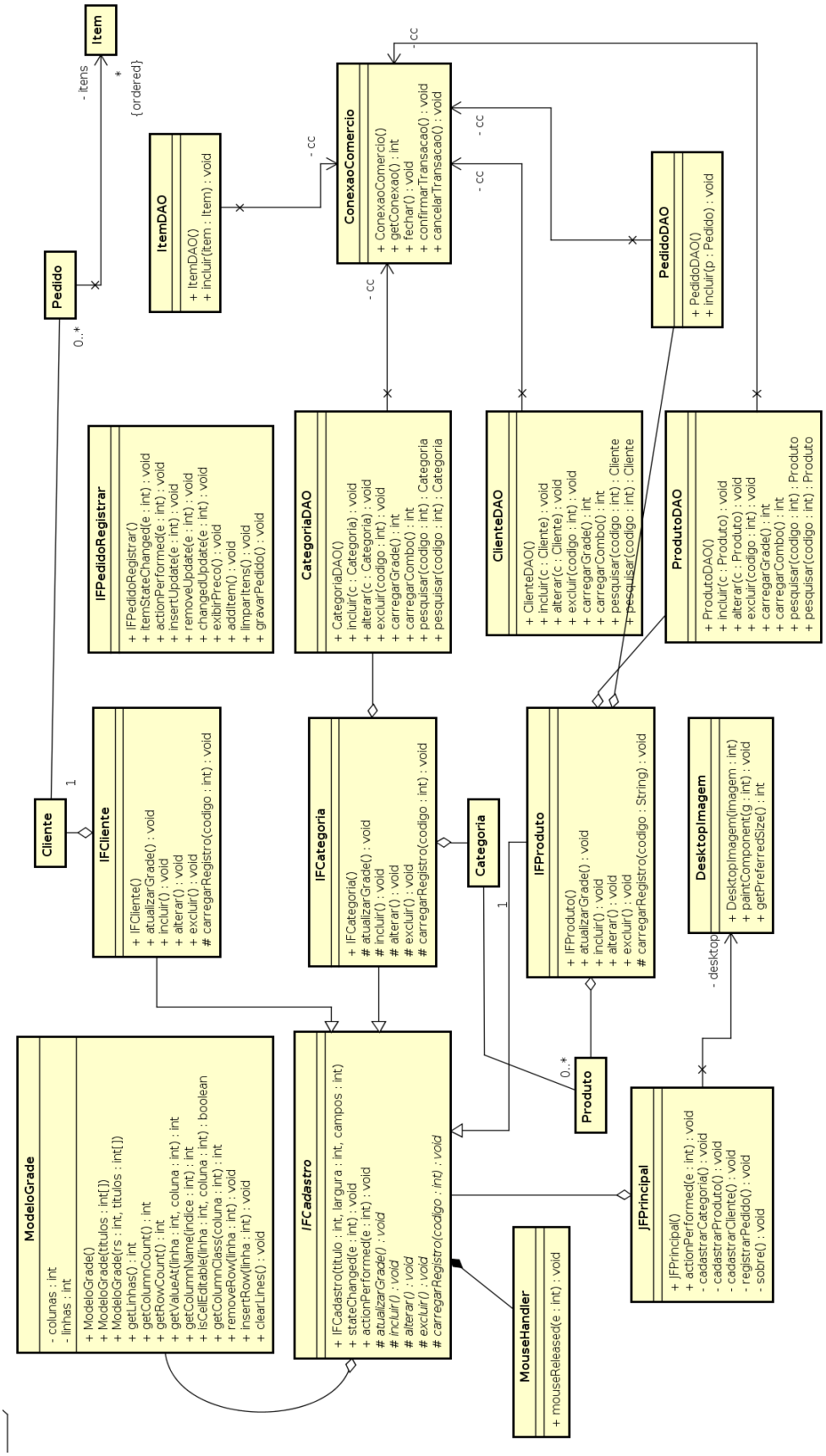

Figura 16: Diagrama de classes da aplicação em seu estado original. [Santos 2014]

6.2. Modelando a refatoração

Esta subseção usa sugestões de refatoração provenientes de um catálogo de refatoração elaborado por [Monteiro 2004]. A refatoração neste contexto quer dizer a extração das preocupações transversais de uma classe e posterior agrupamento destas preocupações em um aspecto ou a utilização dos recursos do paradigma orientado a aspecto para modularização do código base, conforme o benefício proporcionado pela refatoração. Para a descrição adequada do fluxo de trabalho, compreendendo a apresentação de trecho código original resumido e sua refatoração, emprega-se o padrão apresentado no Quadro 2.

Quadro 2: Fluxo do processo de refatoração.

• Motivação • Nome da refatoração que melhor se encaixa ao domínio do problema e descrição da ação recomenda; • Apresentação do trecho de código que tipifica um problema geral (espalhamento, entrelaçamento ou acoplamento); e • Apresentação da implementação proposta para solucionar o problema ou que melhora o estilo implementado.

Fonte: [Simon 2005].

Para a descrição dos artefatos produzidos durante o processo de refatoração é empregado o padrão apresentado no Quadro 3, que identifica o artefato produzido pela atividade performada.

Quadro 3: Descrição padrão usada para detalhar a atividade realizada durante a refatoração.

Nome: <<*nome/título da atividade*>>
Data: <<*data em que a atividade foi realizada*>>
Descrição: <<*objetivo da atividade e descrição do que foi realizado*>>
Artefato de entrada: <<*amostra de um arquivo utilizado na realização da atividade e/ou no qual foi aplicado a refatoração*>>
Artefatos de saída: <<*amostra de um arquivo modificado pela atividade ou que foi criado a partir da realização da atividade*>>
Exemplo de Refatoração: <<*exemplo de código antes e depois da refatoração*>>

Fonte: [Simon 2005].

Os artefatos de entrada podem ser precedidos por um asterisco, o que indica qualquer arquivo que tenha um determinado padrão em sua assinatura. Esta nomenclatura foi empregada para tornar mais simples a definição dos artefatos de entrada, pois muitos arquivos podem compartilhar uma mesma característica referente às suas assinaturas. Os exemplos de refatoração apresentados consistem em trechos de código fonte do artefato de saída.

6.3. Refatorar para extração de características

Exposto no capítulo 2 (Figura 2) como um dos motivos para o desenvolvimento do paradigma orientado a aspectos, o espalhamento de código referente a uma característica transversal e ou entrelaçamento de código com uma característica transversal não relacionada à função primeva de uma classe transparece uma situação prosaica que ocorre em diversas instâncias de código. A refatoração apresentada nesta seção lida com essa questão, isto é, mover os vários elementos implementados em uma, ou comumente mais de uma, classe para um aspecto. No paradigma de orientação a objetos, a extração de métodos e variáveis figura como uma prática corriqueira e está entre os tipos de refatoração mais simples e óbvias. O programador, se assim desejar, pode ainda agrupar os métodos e variáveis das classes envolvidas (quando possível) em uma classe interna.

Entretanto, as limitações do paradigma de orientação a objetos impedem que o processo de refatoração vá adiante. Com a programação orientada a aspectos, este processo pode ir além ao oferecer construtores que permitem ao programador extrair esta classe que, em princípio é interna, e posteriormente inseri-la em um aspecto. Uma outra maneira recorrente para se organizar o código é tornar os serviços oferecidos pelas classes mais claros é fazendo uso de interfaces. Estas estruturas podem ser completamente modularizadas também inserindo-as em um aspecto, bem como as conexões com as classes que as implementam por meio de introduções e declarações do tipo “implemente” dentro de um aspecto. Este trabalho não tenta cobrir todos os possíveis cenários que são passíveis de refatoração, mas apenas intenta exemplificar com um nível de detalhamento que permita a replicação desta prática.

6.3.1. Extrair Tratamento de Exceções

Aparentando ser um dos casos mais severos de espalhamento e entrelaçamento, o tratamento de exceções, apesar de necessário para lidar com erros em tempo de execução, nubla a legibilidade da aplicação, além de deteriorar sua simplicidade e objetividade o que redunda em maior dificuldade de manutenção.

Todo o código é escrito para executar sob certas condições. Se essas condições não forem válidas, pode ser que o código não execute as ações corretamente. As exceções constituem o mecanismo pelo qual a aplicação pode ser informada sobre os desencontros entre o código e o ambiente, e o programador pode ser informado de maneira eficiente sobre o comportamento da aplicação.

Na documentação Java [Oracle, 2018], são destacadas vantagens propiciadas pelo uso de tratamento de exceções em programas, como:

- Todas as exceções lançadas são objetos, o agrupamento e diferenciação de tipos de erros, logo, é um resultado natural da hierarquia de classes.
- Habilidade de propagar o erro, reportando-o acima na pilha de execução.
- Promoção de um meio para separação dos detalhes relacionados ao comportamento da aplicação em caso de um evento excepcional.

Sobre esta última vantagem, o leitor deve considerar o pseudocódigo relacionado à leitura de um arquivo apresentado na Listagem 16. À primeira vista, trata-se da implementação de uma tarefa simples se forem desconsiderados os potenciais problemas como o arquivo em questão não ser passivo de leitura, tamanho do arquivo necessário para alocação de memória não puder ser recuperado, erro durante alocação de memória e ou erro ao fechar o arquivo.

Listagem 16: Exemplo de aplicação que tem como objetivo ler um arquivo.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Fonte: [Oracle, 2018]

Para lidar com estes possíveis eventos, o código apresentado à esquerda na Listagem 16 precisa de tantas linhas de código referentes a detecção de erros que estas linhas auxiliares acabam por escondem o fluxo principal da função e a tornam quase ininteligível. Ainda na Listagem 16, à direita, é apresentada a mesma função, porém agora usando tratamento de exceção ao invés da abordagem tradicional de detecção de erros.

Listagem 17: Comparação entre duas aplicações que têm como objetivo ler um arquivo.

Tradicional

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into
memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose &&
errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode && -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Orientado a Objetos

```
readFile() {
    try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch
(sizeDeterminationFailed) {
        doSomething;
    } catch
(memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Fonte: [Oracle, 2018]

Apesar da função à direita na Listagem 17 possuir menos linhas de código para detectar, reportar e tratar erros relacionados à função de leitura original, há ainda uma maneira mais radical e transparente de prover à aplicação um comportamento desejável em casos excepcionais. A Listagem 18 apresenta um exemplo de tratamento de exceções usando aspectos.

Listagem 18: Tratamento de exceções usando aspectos.

```
method1 {
     call method2;
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

```
public aspect Exception{

public pointcut
AnyException(Exception ex):
      execution(* *.method1(..));

after() throwing(Exception ex):
       AnyException(ex){

     log("error",ex.getMessage()) ;
    }
}
```

Fonte: Autor.

O leitor deve observar que no trecho da tabela referente à refatoração de tratamento de exceções é tomado como artefato de entrada a classe `IFProduto.java` que trata as exceções do nível subjacente. Para agrupar as exceções no nível mais adequado, aproveita-se a possibilidade de propagação de erros. Portanto, tão logo um erro seja propagado, este pode ser tratado em um *advice* apropriado. Onde este *advice* está vinculado a um *pointcut* apontando para o nível na hierarquia da aplicação em que se espera capturar a exceção. Agora, segundo o padrão retratado na seção 6.2, a Tabela 14 mostra o resultado da refatoração Extrair Tratamento de Exceções.

Tabela 14: Descrição da refatoração Extrair Tratamento de Exceções.

Nome: Extrair Tratamento de Exceções
Data: 21/04/2018
Artefato de entrada: `IFProduto.java`
`Artefatos de saída:` `Exceptions.aj/IFProduto_v2.java`
Descrição: • Crie um aspecto vazio no pacote apropriado • Se o tipo de exceção for do tipo *Unchecked*, desfaça os blocos *try/catch* e lance as exceções para nível desejado, enquanto for possível. • Se o tipo de exceção for *Checked*, desfaça os blocos try/catch, use uma declaração do tipo *softening* apontando para a função que não permite o lançamento de exceções e use um *advice* do tipo *around*. • No aspecto, selecione os pontos de interesse no nível em que as exceções não podem mais ser lanças ou em outro ponto desejado. Encapsule a função que lança o tipo de exceção desejada por meio de um *pointcut*. • Crie um *advice* do tipo *after() throwing* para cada *pointcut* e passe a exceção capturada pelo *pointcut* para o corpo do *advice*. Realize as operações desejadas dentro do corpo do *advice*.

Tabela 14: Descrição da refatoração Extrair Tratamento de Exceções.

(continuação)

Exemplo de Refatoração:

- Código antes da refatoração:

```
public class IFProduto extends IFCadastro {
  private JFormattedTextField tfPreco;
  private JComboBox<Categoria> coCategoria;

  public IFProduto() {
    super("Cadastro de produtos",300,4);

    tfPreco = new JFormattedTextField(new Double(0));
    coCategoria = new JComboBox<Categoria>();

    pnRotulos.add(new JLabel(" Preço: "));
    pnRotulos.add(new JLabel(" Categoria: "));
    pnCampos.add(tfPreco);
    pnCampos.add(coCategoria);

    try {
      coCategoria.setModel(new DefaultComboBoxModel<Categoria>(
        new CategoriaDAO().carregarCombo()));
    }
    catch (Exception ex) {
      showMessageDialog(this,ex.getMessage(),"Erro",ERROR_MESSAGE);
    }
  }

  public void atualizarGrade() {
    try {
      ResultSet rs = new ProdutoDAO().carregarGrade();
      tbDados.setModel(new ModeloGrade(rs,new String[]{"Código",
        "Descrição"}));
      tbDados.getColumnModel().getColumn(0).setMaxWidth(50);
    }
    catch (Exception e) {
      showMessageDialog(this, e.getMessage(),"Erro",ERROR_MESSAGE);
    }
  }
...
}
```

Tabela 14: Descrição da refatoração Extrair Tratamento de Exceções.

(continuação)

Exemplo de Refatoração:

- Código após a refatoração:

```
public aspect Exceptions {
      public pointcut DAOExceptions(): execution(private *
JFPrincipal.*(..)throws DAOException);

      after() throwing (DAOException dex): DAOExceptions(){
            JOptionPane.showMessageDialog(null, dex.getMessage(), "Erro",
JOptionPane.ERROR_MESSAGE);
      }

      public pointcut CCEExceptions(): execution(private *
IFCadastro+.*(..)throws ClassCastException );

      after() throwing (ClassCastException ex): CNFExceptions(){
            JOptionPane.showMessageDialog(null, ex.getMessage(), "Tipo do
elemento especificado incompatível",
                        JOptionPane.ERROR_MESSAGE);
      }

      public pointcut CNFExceptions(): execution(private *
IFCadastro+.*(..)throws ClassNotFoundException );

      after() throwing (ClassNotFoundException ex): CNFExceptions(){
            JOptionPane.showMessageDialog(null, ex.getMessage(), "Driver
não encontrado!", JOptionPane.ERROR_MESSAGE);

      }
...
}

public class IFProduto_V2 extends IFCadastro {
...
      coCategoria.setModel(new DefaultComboBoxModel<Categoria>(
        new CategoriaDAO().carregarCombo()));
...
  }
```

A Figura 17 ilustra a atividade de refatoração escolhida e representada pelo *controller* rotulado como ETE (Extrair tratamento de Exceções). A atividade de extração do tratamento de exceções aplicada à classe `IFProduto.java` produz como resultado uma nova classe, livre da responsabilidade de tratar as exceções, e um aspecto que contém os *pointcuts* que vinculam a nova versão da classe refatorada ao aspecto `Exceptions.aj` bem como um *advice* encarregado de tratar

as exceções que por ventura sejam lançadas por `IFProduto_V2.java`. Este vínculo é representado na imagem por uma seta tracejada com o estereótipo *crosscutting* entre chaves angulares.

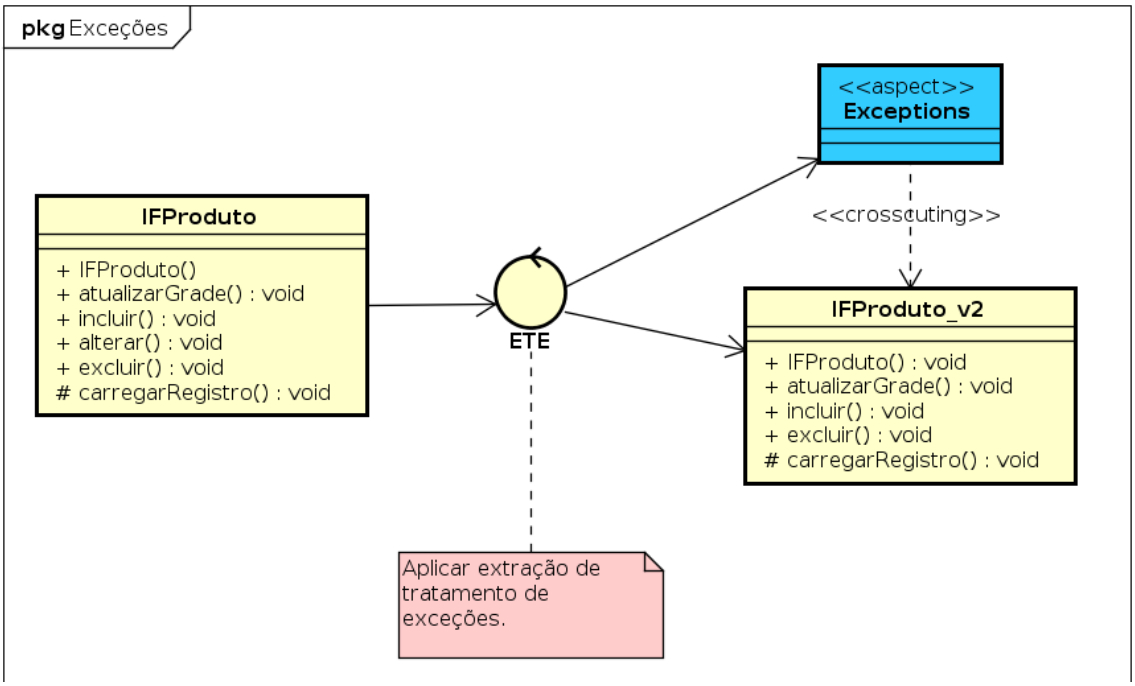

Figura 17: Extraindo a responsabilidade de tratar exceções de uma classe.

Este mesmo processo de refatoração é aplicado sequencialmente a toda a aplicação, retirando de todas as classes todo e qualquer código relacionado (espalhado e entrelaçado) com o tratamento de exceções do código principal da aplicação.

6.3.2. Extrair gerenciamento de transações

Em uma implementação tradicional, completa e extensa cada método que possui transações lidam com problemas de: vazamento de memória, transações não finalizadas corretamente, excesso de conexões abertas e até mesmo SQL *injection*. Ainda assim, a classe responsável pela lógica de negócio ficará “poluída” por tantas precauções que se espalham e que não podem ser refatoradas diretamente em outro método. Neste cenário, um aspecto pode servir para modularizar as partes redundantes para a lógica de negócio. A descrição na Tabela 15 é baseada em um

exemplo de refatoração de chamado “extração de *advice*”, sugerido por [Hanenberg et al. 2003].

Tabela 15. Descrição da refatoração Extrair Gerenciamento de Transações.

Nome: Extrair Gerenciamento de Transações
Data: 01/04/2018
Artefato de entrada: `*DAO.java`
Artefatos de saída: `Transactions.aj/*DAO_v2.java`
Descrição: • Criação de um *pointcut* que capture o método relevante à transação e seu contexto. • Utilização de um *around advice* para encapsular a transação. Este *advice* substitui o ponto de interesse escolhido, executa o código original e retorna o seu resultado por meio da instrução `proceed()`. • Finalmente, após os passos anteriores, é possível retirar os fragmentos de código que são alheios à lógica de negócio do método principal e os copiar para o corpo do *advice*.

Tabela 15: Descrição da refatoração Extrair Gerenciamento de Transações.

(continuação)

Exemplo de Refatoração:

- Código antes da refatoração:

(Código base sem gerenciamento de transações.)

- Código após a refatoração

```
public pointcut transactionOperation(Connectable dao)
      : (call( public void Connectable+.*(..) throws SQLException )
      ||execution( public Vector Connectable+.*(..) throws SQLException )
            ||execution( public Categoria Connectable+.*(..) throws
SQLException )
                  ||execution( public Produto Connectable+.*(..) throws
SQLException ))
                        && target(dao);

      Object around(Connectable dao) throws DAOException:
transactionOperation(dao) {
            String method = thisJoinPoint.getSignature().getName();
            try {
                  Object ret = proceed(dao);

                  if (shouldShow(method)) {
                        showMessage(method);
                  }
                  return ret;
            } catch (SQLException e) {
                  try {
                        if (dao.getConnection() != null)
                              dao.getConnection().cancelarTransacao();
                        throw new DAOException("Erro na operação " +
method, e);
                  } catch (SQLException e1) {
                        throw new DAOException("Erro no operação " +
method + " e rollback.", e1);
                  }
            }finally {
                  try {
                        if (dao.getConnection() != null)
                              dao.getConnection().fechar();
                        if (shouldShow(method)) {
                              JOptionPane.showMessageDialog(null,
"Conexão fechada com sucesso!", "Mensagem",
      JOptionPane.INFORMATION_MESSAGE);
                        }
                  } catch (SQLException e2) {
                        return new DAOException("Erro ao fechar a
comexão.", e2);               }}}
```

A Figura 18 ilustra a atividade de refatoração descrita na Tabela 15. Todas as classes que interagem diretamente com o banco de dado e que antes estavam suscetíveis aos problemas discriminados anteriormente passam a ser vinculadas a um aspecto chamado `Transaction.aj` que assume a responsabilidade de garantir que caso uma transação falhe, suas consequências são tratadas dentro de um aspecto.

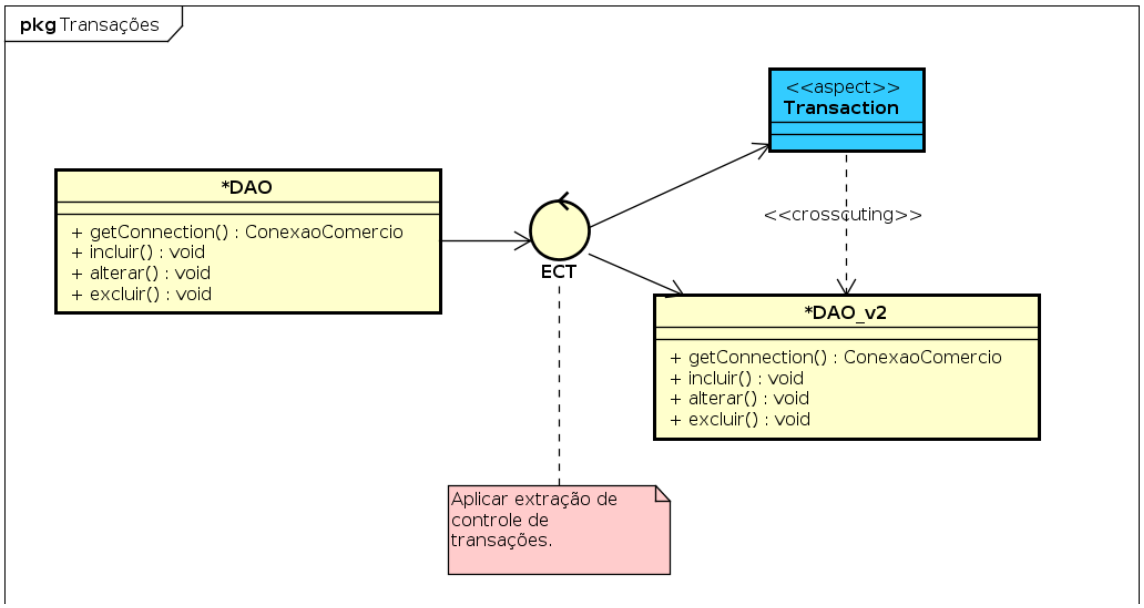

Figura 18: Extraindo o gerenciamento de transações de uma classe.

6.3.3. Extrair classe interna

Transformar uma classe interna em uma classe autônoma é um passo intermediário antes de inseri-la em um aspecto, como é feito posteriormente. A Classe interna é um tipo especial de classe que é definida dentro de outra classe, intitulada classe externa. Este recurso é usado quando programadores se veem à frente de limitações de acoplamento incontornáveis. Ademais, as classes internas trazem benefícios como: possibilidade de estruturação de código, são encapsuladas da mesma maneira que os atributos e métodos da classe externa, são invisíveis a todas as outras classes do pacote ao qual a classe externa pertence, podem

acessar diretamente atributos pertencentes à classe externa, possuem referência *this,* entre outras.

Porém, com opções oferecidas pelo paradigma orientado a aspectos, esta pode não ser a escolha mais vantajosa do ponto de vista da simplicidade. Assim como a refatoração de métodos extensos facilita a leitura e entendimento da responsabilidade de uma classe, também aumentam suas chances de reutilização.

Outro fator que merece ser mencionado diz respeito a estudos de performance de programas em tempo de execução que ajudaram a delinear premissas para algoritmos modernos de coleta de lixo em máquinas virtuais. Segundo a mais importante hipótese [Silveira 2012] advinda destes estudos, a hipótese das gerações [Lieberman and Hewitt 1983], geralmente 95% dos objetos criados ao longo da execução de um programa têm vida extremamente efêmera. Portanto, a melhor técnica que um desenvolvedor pode utilizar é adaptar a demanda de memória de seu programa à hipótese das gerações, ou em outras palavras, priorizar muitos objetos pequenos e encapsulados de acordo com a necessidade, que logo são coletados pelo algoritmo de coleta de lixo da máquina virtual, do que grandes objetos que demoram a sair da memória. A atividade de extração de uma classe interna é descrita na Tabela 16.

Tabela 16. Descrição da refatoração Extrair Classe Interna.

Nome: Extrair Classe Interna
Data: 12/04/2018
Artefato de entrada: `IFCadastro_v2.java`
Artefatos de saída: `IFCadastro_v3.java/MouseHandler_v2.java`
Descrição: • Procure por qualquer código dentro da classe interna relacionado ao comportamento que deve ser mantido dentro da classe hospedeira. • Crie na classe interna um campo privado do tipo da classe externa. • Crie um construtor público para a classe interna e inclua um parâmetro para receber a classe hospedeira como argumento. Atualize qualquer código relacionado à criação de instâncias da classe interna na classe envolvente. Se a classe interna não for privada, verifique também possíveis usos fora da classe de inclusão. • Compile e teste. • Procure por todas as referências diretas aos campos pertencentes ao objeto delimitador. • Procure por chamadas para métodos privados feitos dentro da classe interna. Relaxe as regras de acesso desses métodos. • Compile e teste. • Crie uma classe autônoma com o mesmo nome da classe interna. Copie o texto de origem da classe interna para a classe autônoma e adicione “público” antes do nome da classe. Verifique as importações que devem acompanhar a classe, bem como as importações que a classe hospedeira não precisa mais. Exclua a classe interna. • Compile e teste.

Tabela 16: Descrição da refatoração Extrair Classe Interna.

(continuação)

Exemplo de Refatoração:

- Código antes da refatoração:

```
public abstract class IFCadastro_v2 ...{
...
class MouseHandler extends MouseAdapter {

            public void mouseReleased(MouseEvent e) {
                  if (e.getButton() != MouseEvent.BUTTON1)
                        return;
                  JTable tb = (JTable) e.getSource();
                  if (tb.getSelectionModel().isSelectionEmpty())
                        return;
                  int lin =
tb.getSelectionModel().getMinSelectionIndex();
                  String codigo = tb.getModel().getValueAt(lin,
0).toString();

                  try {
                        carregarRegistro(codigo);
                        tpAbas.setSelectedComponent(pnManutencao);
                        tfDesc.requestFocus();
                  } catch (Exception ex) {
                        showMessageDialog(IFCadastro.this,
ex.getMessage(), "Erro", ERROR_MESSAGE);
                  }
            }
      }
}
```

- Código após a refatoração:

```
public static class MouseHandler_v2 extends MouseAdapter {
      private IFCadastro _enclosing;

      public MouseHandler(IFCadastro cadastro) {
                  this._enclosing = cadastro;
      }

      public void mouseReleased(MouseEvent e) {
            ...
            }
  }
```

A Figura 19 ilustra a atividade descrita na Tabela 16. A classe interna que antes era vinculada a sua classe externa por meio de uma relação de composição é extraída e passa a ser uma classe autônoma sem prejuízo para a classe externa, que agora é mais simples e inteligível.

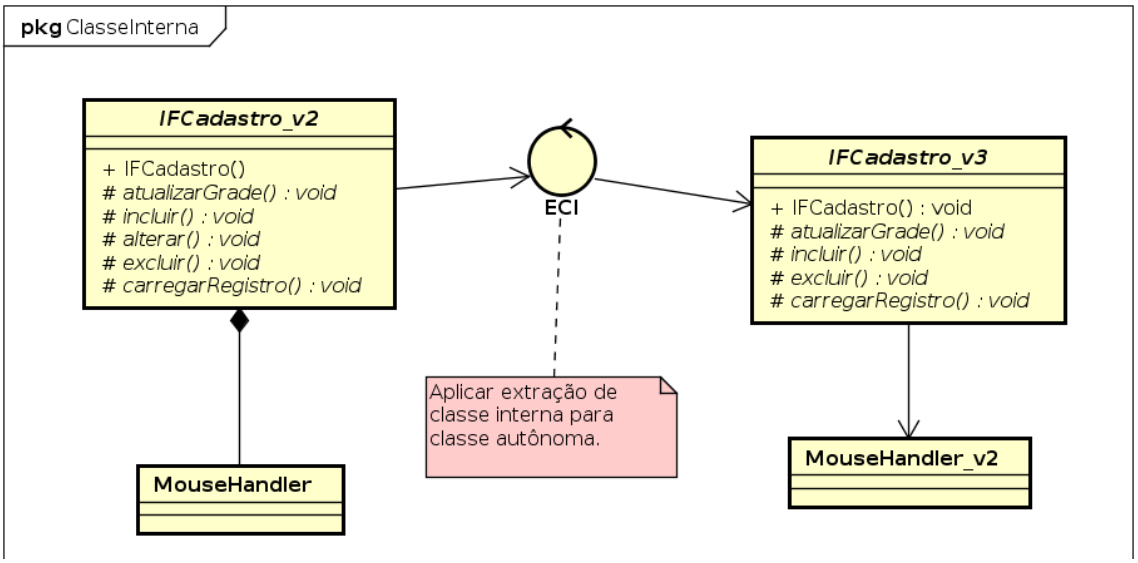

Figura 19: Extraindo uma classe interna para classe autônoma.

6.3.4. Substituir declaração de implementação por parentesco

Em Java, as interfaces têm como propósito principal suprir a ausência de herança múltipla oferecendo um mecanismo mais eficiente. São também uma maneira de representar vários papéis interpretados por uma classe. Sendo assim uma interface é uma espécie de contrato contendo serviços que podem ser assinados pela classe. A linguagem AspectJ permite encapsular muitos desses papéis dentro de aspectos, os quais recorrem a interfaces para representá-los. A Tabela 17 apresenta em detalhes a atividade de substituição de implementação por uma declaração de parentesco.

Tabela 17. Descrição da refatoração Substituir Implementação por Parentes.

<table>
<tr><td>Nome: Substituir Implementação por Parentesco</td></tr>
<tr><td>Data: 19/04/2018</td></tr>
<tr><td>Artefato de entrada: <code>Connectable.java</code></td></tr>
<tr><td>Artefatos de saída: <code>Implementations.aj</code></td></tr>
<tr><td>Descrição:
Crie a “declaração parental” apropriada dentro do aspecto.
Na classe que realiza a interface, remova a cláusula <code>implements</code> referente à interface.
Compile e teste.</td></tr>
<tr><td>Exemplo de Refatoração:
Código antes da refatoração:
<pre>public class CategoriaDAO_v2 implements Connectable{...}</pre>
Código após a refatoração:
<pre>public aspect Implementations {
...
      declare parents: CategoriaDAO_v3 implements Connectable;
}
public class CategoriaDAO_v3{...}</pre></td></tr>
</table>

6.3.5. Inserir Interface *inline* no aspecto

O cenário ideal para este procedimento ocorre quando um recurso está sendo extraído da base de código existente. Contudo, conforme a característica é extraída, em um certo ponto, somente o aspecto guarda a referência para a interface. Neste cenário, não há motivo para manter a interface independente, sendo melhor inseri-la no aspecto que guarda a sua referência, como ilustra a Tabela 18.

Tabela 18. Descrição da refatoração Inserir Interface no Aspecto.

<table>
<tr><td>Nome: Inserir Interface no Aspecto</td></tr>
<tr><td>Data: 19/04/2018</td></tr>
<tr><td>Artefato de entrada: <code>Connectable.java</code></td></tr>
<tr><td>Artefatos de saída: <code>Implementations.aj</code></td></tr>
<tr><td>Descrição:
Crie dentro do aspecto uma cópia da interface autônoma.
Remova a interface autônoma.
Compile e teste.</td></tr>
<tr><td>Exemplo de Refatoração:
Código antes da refatoração:
<pre>
public interface Connectable {
	public ConexaoComercio getConnection();
}
</pre>
Código após a refatoração:
<pre>
public aspect Implementations {
	public static interface Connectable {
		public ConexaoComercio getConnection();
	}
	declare parents: CategoriaDAO_v3 implements Connectable;
...
}
</pre></td></tr>
</table>

6.3.6. Inserir classe *inline* no aspecto

Esta situação ocorre quando uma classe auxiliar de tamanho diminuto (ou um pouco menos subjetivamente, uma classe com menos de uma tela cheia de linhas de código) é modularizada em um aspecto. Classes internas são geralmente pequenas e essa refatoração é recomendada apenas para classes pequenas. Para exemplificar, é tomado como amostra o artefato de saída da atividade anterior obtido após a atividade de extração de classe interna. A compilação e teste da classe concluem a refatoração. A Tabela 19 detalha a atividade de inserção de uma classe em um aspecto.

Tabela 19. Descrição da refatoração Inserir Classe no Aspecto.

Nome: Inserir Classe no Aspecto

Data: 16/04/2018

Artefato de entrada: `MouseHandler_v2.java`

Artefatos de saída: `Auxiliary.aj/MouseHandler_v3.java`

Descrição:

- Crie uma cópia do código-fonte da classe autônoma, dentro do aspecto. Acrescente o modificador estático antes do modificador público seguido pelo nome da classe.
- Adicione na seção de importação do aspecto qualquer importação que possa ser necessária à classe.
- Remova a classe autônoma.
- Compile e teste.

Exemplo de Refatoração:

- Código antes da refatoração:

Classe `MouseHandler_v2.java` apresentada na subseção 6.3.3

- Código após a refatoração:

```
privileged public aspect Auxiliary {

public static class MouseHandler_v3 extends MouseAdapter {
      private IFCadastro _enclosing;

      public MouseHandler(IFCadastro cadastro) {
                  this._enclosing = cadastro;
      }

      public void mouseReleased(MouseEvent e) {
            ...
            }
  }

}
```

6.4. Resultado da refatoração

A imagem Figura 20 representa o resultado do processo de refatoração. O diagrama possui quatro aspectos, representados em azul, obtidos como resultado das atividades de refatoração. Ademais, foi criada uma interface para agrupar o papel de conexão como o banco de dados. Com a utilização desta interface, as classes que a implementam se tornam genéricas o suficiente, o que evita a repetição de código no aspecto encarregado de encapsular a preocupação transversal desejada. Então, o aspecto `Transactions.aj` passa a atuar sobre esta única interface, em vez de várias classes. Por fim, o aspecto `Exceptions.aj` passa a capturar e a tratar exceções das classes pertencentes à camada de apresentação. O aspecto `Exceptions.aj` possui uma classe interna usada para encapsular exceções especificamente referentes ao banco de dados da aplicação.

Após este processo de refatoração, 25 casos de teste foram executados com a finalidade de mostrar que o comportamento da aplicação permanece o mesmo se comparado com o comportamento observado antes do processo de refatoração. Uma vez que comportamento de fato permanece inalterado, o capítulo 7 examina quais são os possíveis impactos (positivos ou negativos) do processo de refatoração usando o paradigma orientação a aspectos.

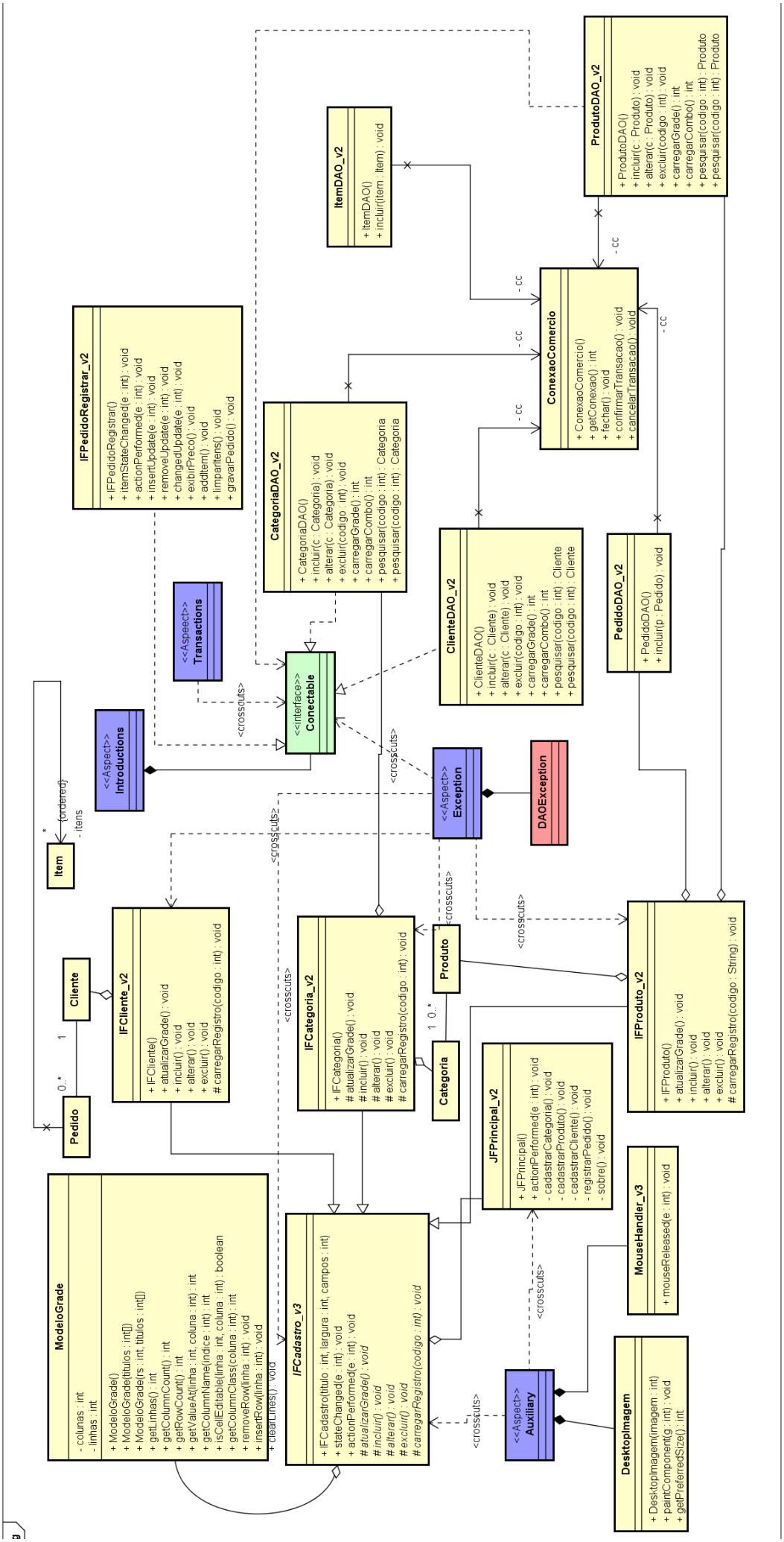

Figura 20: Diagrama de classes da aplicação em seu estado pós-refatoração.

7. IMPACTOS DA REFATORAÇÃO

Este capítulo intenta reportar os efeitos observados no software pós-refatoração. A maior melhoria aplicada na aplicação apresentada na figura Figura 16, sem dúvida, é a extração das preocupações espalhadas e entrelaçadas no código-fonte. Ademais, houve a implementação não invasiva de gerenciamento de transações, contudo sem alterar o comportamento externo da aplicação. A seção 7.1 apresenta as métricas usadas como referencial para comparar o software antes e depois do processo de refatoração. A seção 7.2 apresenta a comparação e avaliação do resultado final do processo de refatoração.

7.1. Métricas

Nesta seção é apresentado as métricas selecionadas para a comparação objetiva entre as duas versões do software usado como objeto de estudo. As métricas usadas aqui foram escolhidas por serem o conjunto de métricas para software orientado a objetos mais amplamente referenciadas dentro da literatura científica [Ribeiro et al. 2015]. Foram selecionadas sete métricas no total. Dentre estas, seis são descritas na redação de [Chidamber and Kemerer 1994]. Também é usada uma métrica para o número de linhas de código (LOC) para a avaliação do software, selecionada por ser a mais popular.

Uma métrica importante que precisa ser destacada (por ser usada indiretamente neste trabalho) é relativa à quantidade fluxos (ou caminhos) lógicos que podem ser percorridos ao longo da execução de um software. Estes caminhos lógicos em um programa podem ser modelados na forma de um grafo dirigido chamado de Grafo de Fluxo. Neste modelo, um vértice representa uma instrução ou um bloco de instruções e uma aresta representa um fluxo de controle. Uma aresta precisa terminar em um vértice, mesmo se esse vértice não represente instrução alguma. Um vértice que contenha uma condição é chamado de vértice predicado e é caracterizado por duas ou mais arestas que emanam do mesmo.

Neste cenário, segundo [Pressman 2016], a complexidade ciclomática de um software fornece uma medida quantitativa da complexidade lógica de um programa baseada no seu grafo de fluxo. Esta medida assume o número de caminhos que introduzem pelo menos um novo vértice pertencente ao conjunto base de um programa. Esta métrica também fornece um limite superior para o número de testes que devem ser realizados para com a finalidade de que todas as instruções (vértices) foram executadas pelo menos uma vez. Por fim, seguem as métricas usadas neste trabalho.

- **Acoplamento entre objetos (CBO) -** esta métrica conta em uma determinada classe o número de outras classes com que esta primeira está acoplada, e vice-versa. Duas classes são ditas acopladas quando métodos declarados uma classe são usados em outra, ou seja, quando uma classe depende de outra para funcionar, como na listagem Listagem 1.
- **Falta de coesão em métodos (LCOM) -** esta métrica mede a assimetria entre métodos em uma classe. Esta métrica é definida como LCOM = $|P| - |Q|$, se $|P| > |Q|$, onde P é o número de métodos pares que não compartilham um atributo comum e Q assume o número de métodos pares que compartilham um atributo comum. Caso a diferença seja negativa, LCOM assume um valor nulo, neste caso 0.
- **Profundidade da árvore de herança (DIT) -** esta métrica retorna o tamanho da maior profundidade da árvore de hierarquia. Esta medida é feita a partir da classe que se deseja tomar essa medida e somando o número de níveis até a raiz da hierarquia. Conforme a profundidade da árvore de herança aumenta, as classes nos níveis inferiores passam a herdar mais métodos. Consequentemente, isso dificulta a previsão do comportamento destas classes.
- **Número de nós filhos (NOC) -** esta métrica retorna o número de subclasses imediatamente subordinadas a uma classe na hierarquia de herança. À medida que o número de filhos cresce, a reutilização aumenta, porém a quantidade de testes também aumenta.

- **Resposta por classe (RFC) -** esta métrica define o conjunto de métodos que podem ser executados em resposta a uma mensagem que é recebida por um objeto. Conforme este conjunto aumenta, o esforço necessário para a execução dos testes também aumenta.
- **Métodos ponderados por classe (WMC)-** esta métrica retorna o somatório das complexidades dos métodos em uma classe, onde cada método é ponderado pela complexidade ciclomática. O número de métodos e sua complexidade são indicadores razoáveis da quantidade de esforço necessária para implementar e testar uma classe. Ademais, quanto maior o número de métodos, mais complexa é a árvore de herança. Por fim, à medida que o número de métodos cresce em uma classe, é provável que esta se torne mais e mais específica para a aplicação, limitando seu potencial reutilização.

7.2. Resultados

Os mesmos 25 casos de teste foram executados antes e depois da refatoração, a fim de se garantir que o software mantém o mesmo comportamento externo. Para cada classe, é calculado um valor para as métricas definidas anteriormente. O objetivo desta subseção é determinar de maneira empiríca se as mudanças observadas no código fonte do software pós-refatoração são significantes e se mudanças redundam em um impacto positivo ou negativo na aplicação.

A tabela Tabela 20 apresenta as estatísticas descritivas correspondentes ao software orientado a objetos e a aspectos. Estas métricas foram computadas usando a ferramenta *Together* [Microfocus, 2018], uma ferramenta comercial escolhida principalmente por oferecer suporte às métricas desejadas. Para cada classe do programa, antes e depois do processo de refatoração, é calculado um valor para as métricas escolhidas.

Tabela 20: Comparação entre as métricas; antes e após a refatoração.

Métrica/Paradigma	Orientada a Objetos (OO)	Orientada a Aspectos (AO)	Variação
LOC	1310	1195	-8.77%
CBO	37	39	+5.40%
LCOM	117	117	0%
DIT	6	6	0%
NOC	21	19	-9.52%
RFC	617	615	-0.32
WMC	21	21	0%
Média	304.14	287.42	-5.5%

Fonte: Autor.

A coluna Variação é o resultado da divisão entre o valor das métricas pré-refatoração pelo valor das métricas pós-refatoração menos um. De acordo com a coluna que indica a variação das métricas, há um decréscimo de 8.77% no número de linhas de código, principalmente devido à extração do tratamento de exceções. A métrica que contabiliza o número de classes também sofre um decréscimo em consequência da extração de classes e inserção destas classes em um aspecto. A única métrica que sofre acréscimo, CBO, está relacionada ao acoplamento entre as classes. Este acréscimo ocorre principalmente devido à implementação de um controle de transações mais completo e seguro. Comparando as médias, entretanto, há um decréscimo de 5.5% no valor das métricas coletadas. Isto mostra que há um ganho positivo global referente à complexidade da aplicação.

8. EXECUTANDO OS TESTES

Esta seção reporta os resultados obtidos durante a execução do Teste de Mutação na aplicação pós-refatoração com a finalidade de avaliar a praticabilidade desta técnica em relação usabilidade e o esforço necessário para aplicá-la. A seção 8.1 retrata o resultado dos testes mutações aplicadas à aplicação logo após o término do processo de refatoração. A seção 8.2 analisa o status dos mutantes resultantes obtidos logo após o Teste de Mutação.

8.1. Conjunto inicial de testes

O conjunto de inicial de testes foi construído baseado na experiência prévia do autor em testes e intenta estressar os principais fluxos de execução da aplicação pós-refatoração com 25 casos de teste. A cada ciclo de teste (que compreendia a execução de todos os casos de testes) é aplicado cada um dos mutantes gerados pela ferramenta *Proteum/AJ* pertencente a cada um dos três grupos de operadores de mutação sumarizados no capítulo 5, que totalizam 24 operadores.

No total, foram gerados 2490 operadores de mutação (tanto para as classes quanto para os aspectos) como mostrado na Tabela 21. Pode-se analisar mais detalhadamente este resultado tomando os operadores de mutação mortos e dividindo-os pelo número de mutantes compiláveis menos os equivalentes, o que resulta na taxa de efetividade do Teste de Mutação, como visto na última linha da Tabela 21. A taxa de efetividade observada para um primeiro teste representa um valor excelente e muito próximo do desejável - 100% de efetividade. À medida que mais mutantes são mortos, por exclusão, há menos mutantes vivos que necessitam de uma análise rigorosa de sua interação como o código fonte da aplicação. Isso resulta em uma maior facilidade e menor esforço necessário para cobrir o restante dos mutantes que permanecem vivos.

Tabela 21. Resultados do caso teste Manter Categoria.

Mutantes anômalos	529
Mutantes compiláveis	1961
Mutantes vivos	60
Mutantes mortos	1768
Mutantes equivalentes	133
Total	2490
Taxa de Mutação	[1768÷(1961−133)] = 0,967177243

Fonte: Autor.

A Figura 21 contém os dados presentes na Tabela 21 em forma de porcentagem para facilitar a interpretação do resultado dos testes. De todos os mutantes gerados pela aplicação, 21,2% são classificados como não compiláveis (anômalos). O restante dos operadores válidos (78,8%) foram aplicados ao software pós-refatoração como a finalidade de simular erros não triviais. Destes operadores, a maior parte, cerca 90%, provocou na aplicação pós-refatoração um comportamento diferente do esperado. O restante é classificado como equivalente, isto é, P(*t*) = m(*t*) para todos os casos de teste com entrada pertencente ao domínio de P. Por fim, o menor conjunto, os 3,1% finais representados em vermelho, são os alvos da análise na próxima subseção. Esta análise, como já informado no capítulo 4, tem a finalidade de avaliar se o operador de mutação realmente representa um fator para atualização do conjunto de casos de teste, ou o operador pode ser considerado equivalente (veja o último exemplo no capítulo 4, seção 4.2). O motivo para um operador de mutação ser considerado equivalente, mesmo após ter sido classificado como vivo, depende de uma investigação a respeito de como este operador de mutação interage com código fonte da aplicação.

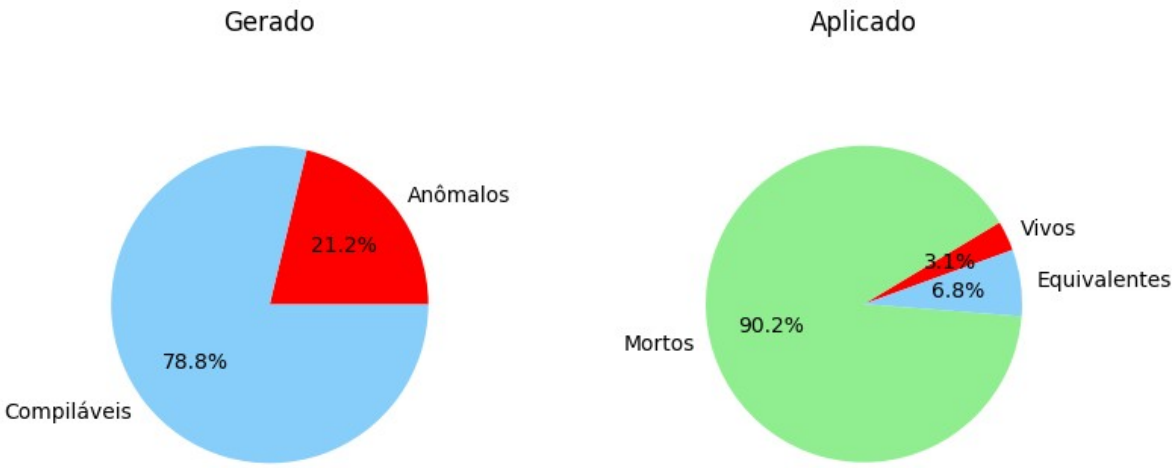

Figura 21: Resultado geral do Teste de Mutação.

8.2. Análise dos mutantes vivos

A ferramenta *Proteum/AJ* possibilita a geração de um relatório em forma de um arquivo de texto preenchido com uma lista de todos os operadores de mutação gerados e seus status recebidos pela ferramenta após a execução do teste. Todos os 60 mutantes vivos estão relacionados com o aspecto *Exceptions.aj*. O Quadro 4 mostra algumas ocorrências contidas neste registro. Nesse cenário, há duas possibilidades: 1) o conjunto de teste é insensível a estes operadores de mutação ou 2) estes operadores são equivalentes.

Quadro 4: Excerto do relatório dos operadores de mutação.

executionFiles/execution/mutationJDB/original/src/aspects/ Operator: **ABAR**
Status: **alive**
58 < after() throwing(NullPointerException ex) : NPExceptions()
58 > before() : NPExceptions()
executionFiles/execution/mutationJDB/original/src/aspects/ Operator: **ABHA**
Status: **alive**
37 < after() throwing(DAOException dex) : DAOExceptions() {
39<JOptionPane.showMessageDialog(null, dex.getMessage(), "DAOException",
40 < JOptionPane.ERROR_MESSAGE);}
executionFiles/execution/mutationJDB/original/src/aspects/ Operator: **ABPR**
Status: **alive**
37 < after() throwing(DAOException dex) : DAOExceptions()
37 > after() throwing(DAOException dex) : CNFExceptions()

Fonte: Autor.

A Tabela 22 mostra que apenas três operadores de mutação geraram mutantes vivos. Comecemos pelo operador ABAR, como apenas uma ocorrência. Este operador altera o momento em que um *advice* é invocado. Aplicando a sugestão ilustrada no Quadro 4 ao código-fonte da aplicação, percebe-se que o operador não aumenta o número de pontos de interesse capturados e afeta somente o momento que a mensagem de erro é mostrada ao usuário, que neste caso é irrelevante. Portanto, este operador de mutação pode ser considerado equivalente.

O próximo operador, ABHA, remove a implementação do corpo do *advice*, provocando a omissão de um ou mais comportamentos. Inserindo o operador de mutação no código-fonte da aplicação e executando-o, percebe-se que, de fato, o comportamento esperado (a exibição de uma mensagem de erro) é omitido. A análise revela que o operador altera o comportamento esperado da aplicação, porém o teste é insensível a este operador por uma limitação técnica do *framework* de teste que não oferece suporte à verificação de uma mensagem lançada pelo aspecto de forma gráfica. Portanto, o mutante deve ser considerado vivo. E o conjunto de teste deve ser atualizado.

O último operador, ABPR, substitui o *pointcut* vinculado a um *advice* por um outro *pointcut* qualquer. Com isso, espera-se que um comportamento indesejado seja executado em resposta a um conjunto de pontos de interesse não intencionado. A análise dos *pointcuts* afetados por esse operador de mutação revela que vários *pointcuts* definidos no aspecto que encapsula as exceções são equivalentes. Uma vez que sua troca não alterou o comportamento do programa. Por essa razão, estes *pointcuts* podem ou não ser substituídos por um único *pointcut,* sem prejuízo à aplicação.

Tabela 22: Resultado detalhado do Teste de Mutação.

Operador	Mutante	Anômalo	Vivo	Compilável	Equivalente	Morto	Taxa
ABAR	10	9	1	1	0	0	0
ABHA	14	0	10	14	0	4	0,285
ABPR	105	2	49	103	0	54	0,524
AJSC	0	0	0	0	0	0	0
APER	0	0	0	0	0	0	0
APSR	6	5	0	1	0	1	1
CGCR	78	57	0	21	0	21	1
CGSR	21	9	0	12	0	12	1
CLCR	943	3	0	940	0	940	1
CLSR	27	5	0	22	0	22	1
DAIC	0	0	0	0	0	0	0
DAPC	0	0	0	0	0	0	0
DAPO	0	0	0	0	0	0	0
DEWC	0	0	0	0	0	0	0
DSSR	3	2	0	1	0	1	1
OASN	38	26	0	12	0	12	1
OEBA	132	132	0	0	0	0	NaN
ORRN	145	80	0	65	0	65	1
PCCC	0	0	0	0	0	0	0

Fonte: Autor.

Tabela 21: Resultado detalhado do Teste de Mutação.

(continuação)

Operador	Mutante	Anômalo	Vivo	Compilável	Equivalente	Morto	Taxa
PCCE	18	2	0	16	0	16	1
PCCR	20	4	0	16	0	16	1
PCGS	0	0	0	0	0	0	0
PCLO	6	4	0	2	0	2	1
PCTT	3	0	0	3	3	0	0
POAC	30	27	0	3	1	2	1
POEC	30	0	0	30	18	12	1
POPL	18	0	0	18	13	5	1
PSDR	0	0	0	0	0	0	0
PSWR	13	0	0	13	1	12	1
PWAR	0	0	0	0	0	0	0
PWIW	131	14	0	117	97	20	1
SDWD	0	0	0	0	0	0	0
SMTC	8	0	0	8	0	8	1
SSDL	556	123	0	433	0	433	1
VDTR	81	15	0	66	0	66	1
VTWD	54	10	0	44	0	44	1

Fonte: Autor.

8.3. Conclusões sobre os resultados obtidos

Com as conclusões obtidas a partir da análise da interação dos operadores de mutação com a aplicação, tem-se um *feedback* robusto a respeito de como melhorar tanto o conjunto inicial de testes quanto o código-fonte da aplicação. A partir desse ponto, trabalha-se para atingir os 100% de taxa de mutação tendo como guia o retorno obtido a cada execução do teste de mutação. Outra possibilidade é definir um limite aceitável para o valor da taxa de mutação, uma vez que há operadores de mutação que geram programas mutantes equivalentes e que têm, por definição, um comportamento indistinguível do programa original.

9. CONCLUSÃO

Este trabalho apresenta um processo de refatoração e uma técnica de teste para software orientado a aspectos. A refatoração do software usado como objeto de estudo foi realizada utilizando-se como referencial um catálogo proposto na literatura científica especificamente para o paradigma de orientação a aspectos. Subsequentemente, a comparação entre ambas as versões do software mostrou que refatoração é positiva no sentido de diminuir a complexidade do software.

A técnica de teste mostrou-se especialmente útil para avaliar a abrangência e eficácia do conjunto inicial de testes, uma vez que a técnica mostra (em forma mutantes vivos) que certas áreas da aplicação não estão sendo devidamente estressadas pelo conjunto inicial de teste. Na experiência realizada neste trabalho, o conjunto inicial de testes usado tanto antes como depois do processo de refatoração mostrou-se insensível aos mutantes gerados pelo operador ABHA. Portanto, o conjunto inicial de casos de teste foi incapaz de discriminar o programa em mutação do programa original (pós-refatoração). Esta situação levou à análise da interação do operador de mutação como o código-fonte e posteriormente ao melhoramento dos casos de teste.

A técnica de Teste de Mutação também se mostra adequada para a avaliação do programa sob teste, apontando quais erros catalogados na taxonomia de falhas para software orientado a aspectos a aplicação pode conter. Este caso ocorreu quando aplicado ao programa pós-refatoração os mutantes gerados pelo operador ABPR. Após a aplicação deste operador, um erro recorrente foi descoberto, o que possibilita a correção deste erro e consequentemente o melhoramento do programa sob teste.

Estas resultados são, fora de qualquer dúvida, impossíveis de serem alcançadas se a prática do teste baseado em falhas for feito manualmente, uma vez que divisar e aplicar todos os possíveis erros que uma aplicação está sujeita, além dolorosamente tedioso, é humanamente impossível se considerado o enorme número de combinações possíveis (compiláveis ou não) para estes defeitos.

Igualmente benéfica, esta técnica de teste previne a mescla inadvertida entre a avaliação das preocupações principais e transversais, permitindo que o testador se ocupe com uma tarefa por vez. A técnica de teste usada aqui também comprova a ausência dos defeitos descritos no capítulo 5 na forma de operadores de mutação mortos ou equivalentes. Isto tudo representa uma métrica robusta com respeito à qualidade do software. Ademais, a avaliação dos mutantes vivos mostra que não é necessário um alto custo em tempo e esforço para a obtenção de uma cobertura completa por parte do teste de mutação. Após a execução do teste de mutação, apenas dois tipos de operadores de mutação geraram mutantes vivos. Isto indica que com a ferramenta adequada é possível fazer uso da técnica de teste de mutação de maneira que o custo/benefício torna-se praticável.

9.1. Limitações

A obtenção e configuração da ferramenta de automação de testes representou o maior fator de dificuldade para a conclusão deste trabalho. Uma vez que a ferramenta utilizada não é disponibilizada de forma pública, mas é possível obtê-la apenas entrando em contato como o desenvolvedor da ferramenta. Ainda assim, por dificuldades de configuração, a ferramenta foi enviada ao autor deste trabalho pré-configurada, em uma máquina virtual, o que redundou em perda de performance e mais tempo gasto na execução dos testes.

Outra limitação encontrada aqui neste trabalho está ligada a restrição de tempo para a comparação da técnica de Teste de Mutação com outras técnicas de teste (como as técnicas que foram retornadas durante a fase de mapeamento). Isto seria especialmente útil para contrastar critérios objetivos observados durante o uso da técnica de teste de Mutação como, por exemplo, o esforço empreendido para avaliação da abrangência do conjunto inicial de teste e sua efetividade.

9.2. Trabalhos futuros

Este trabalho deve seguir adiante futuramente revisitando as limitações descritas na subseção anterior. A escolhida da aplicação como objeto de experimento deveu-se, sobretudo, por motivos de simplicidade e didática, porém há a necessidade de estender o experimento realizado para aplicações robustas e de grande porte, isto é, aplicações comerciais, como sistemas web em Java, por exemplo. Ademais, a implementação mais popular e a mais investigada do paradigma de orientação a aspectos, AspectJ, possui duas versões; a usada neste trabalho (versão tradicional) e a sintaxe alternativa ou @AspectJ que usa anotações para expressar elementos transversais. Esta última sintaxe é usada pelo *framework* Spring dentro de sua estrutura AOP baseada em *proxy* sem precisar de um *weaver* AspectJ. Portanto, da mesma maneira feita anteriormente neste trabalho com uma aplicação pequena, é preciso entender o custo necessário para realizar o processo de teste para os cenários aqui descritos.

REFERÊNCIAS

AHMAD, S., Ghani, A. A. A. and Sani, F. M. (2014). Dependence Flow Graph for analysis of aspect-oriented Programs. *International Journal of Software Engineering & Applications (IJSEA)*, v. 5, n. 6, p. 125–144.

ALEXANDER, R. T., Bieman, J. M. and Andrews, A. a (2004). Towards the Systematic Testing of Aspect-Oriented Programs. *technique Colorado State*, n. C, p. 11.

Apache Foundation, TomCat Documentation, Disponível

em: <http://tomcat.apache.org/tomcat-8.0-doc/>. Acesso em: 09/06/2018.

BAEKKEN, J. S. (2006). A Fault Model for Pointcuts and Advice in AspectJ Programs: Thesis. School of Electrical Engineering and Computer Science, Washington state university.

CHERAIT, H. and Bounour, N. (2015). History-based approach for detecting modularity defects in aspect oriented software. *Informatica (Slovenia)*, v. 39, n. 2, p. 187–194.

CHIDAMBER, S. and Kemerer, C. (1994). A metrics suite for OO design. *IEEE Transactions on Software Engineering,v. 20, n. 6, p. 476–493.*

DEMILLO, R. A., Lipton, R. J. and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, v. 11, n. 4, p. 34–41.

FERRARI, F. C., Nakagawa, E. Y., Rashid, A. and Maldonado, J. C. (2010). Automating the Mutation Testing of Aspect-oriented Java Programs. *Proceedings of the 5th Workshop on Automation of Software Test*, n. Section 3, p. 51–58.

FERRARI, F. C., P. Cafeo, B. B., Levin, T. G., et al. (20 dec 2015). Testing of aspect-oriented programs: difficulties and lessons learned based on theoretical and practical experience. *Journal of the Brazilian Computer Society*, v. 21, n. 1, p. 20.

FERRARI, F. C., Rashid, A. and Maldonado, J. C. (2011). Design of mutant operators for the AspectJ language. *University of Sao Carlos, Sao Carlos*, n. 1, p. 38.

FERRARI, F. C., Rashid, A. and Maldonado, J. C. (2013). Towards the practical mutation testing of AspectJ programs. *Science of Computer Programming*, v. 78, n. 9, p. 1639–1662.

FOWLER, M. (1999). *Refactoring: Improving the Design of Existing Code*. 1. ed. Boston: Addison-Wesley Professional.

GAMMA, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

GHANI, A. A. A. and Parizi, R. M. (2013). Aspect-oriented program testing: An annotated bibliography. *Journal of Software*, v. 8, n. 6, p. 1281–1300.

HANENBERG, S., Oberschulte, C. and Unland, R. (2003). Refactoring of aspect-oriented software. *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays).*, p. 19–35.

HILSDALE, E., Hugunin, J., Isberg, W., Kiczales, G. and Kersten, M. (2001). Aspect-oriented programming with AspectJ; . Mini-course at OOPSLA. https://pdfs.semanticscholar.org/82c6/bb8a70ed98294046a02527c5f4fb52542d34.pdf.

ISTQB, Glosary, Disponível

em: <http://glossary.istqb.org>. Acesso em: 20/06/2018.

JORGENSEN, P. C. (2013). *Software Testing : A Craftsman's Approach*. 4. ed. Boca Raton: CRC Press.

KICZALES, G., Hilsdale, E., Hugunin, J., et al. (2001). An Overview of AspectJ. *ECOOP 2001—Object-Oriented Programming*, v. 2072, n. September, p. 207–235.

KICZALES, G., Lamping, J., Mendhekar, A., et al. (1997). Aspect-oriented programming. *ECOOP'97 — Object-Oriented Programming*, v. 1241/1997, n. June, p. 220–242.

LADDAD, R. (2009). *Aspectj in Action Enterprise AOP with Spring Applications*. 2. ed. Manning Publications Co.

LEMOS, O. A. L., Ferrari, F. C. and Lopes, P. C. M. C. V (2006). Testing aspect-oriented programming Pointcut Descriptors. *WTAOP ’06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, p. 33–38.

LIEBERMAN, H. and Hewitt, C. (1983). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, v. 26, n. 6, p. 419–429.

LINDSTRÖM, B., Offutt, J., Sundmark, D., Andler, S. F. and Pettersson, P. (jan 2017). Using mutation to design tests for aspect-oriented models. *Information and Software Technology*, v. 81, p. 112–130.

MAREW, T., Kim, J. and Bae, D. H. (feb 2007). Systematic Mapping Studies in Software Engineering. *International Journal of Software Engineering and Knowledge Engineering*, v. 17, n. 01, p. 33–55.

MARTIN, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River: Prentice Hall.

MICROFOCUS, Together Tool Documentation, Disponível em

<https://www.microfocus.com/products/requirements-management/together/>, Acesso em: 18/07/2018.

MONTEIRO, M. P. (2004). Catalogue of Refactorings for AspectJ. *Universidade do Minho, Tech. Rep. UM-DI-GECSD-200402*, p. 46.

OFFUTT, A. J. and Pan, J. (1998). Automatically detecting equivalent mutants and infeasible paths. *Software Testing Verification and Reliability*, v. 7, n. 3, p. 165–192.

ORACLE, Java Documentation, Disponível

em:

<

https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.

html>. Acesso em: 09/06/2018.

PRESSMAN, R. S. (2016). *Software Engineering: A Practitioner’s Approach*. 8. ed. Palgrave Macmillan.

RIBEIRO, M., Reis, R. Q. and Abelem, A. J. G. (2015). How to automatically collect oriented object metrics: A study based on systematic review. *Proceedings - 2015 41st Latin American Computing Conference, CLEI 2015*,

SANTOS, R. R. Dos (2014). *Programação de computadores em Java*. Rio de Janeiro: NovaTerra.

SILVEIRA, P. (2012). *Introdução à arquitetura e design de software: uma visão sobre a plataforma Java*. 1. ed. Sao Paulo: Elsevier.

SIMON, L. tollens (2005). Uma experiência na refatoração da arquitetura e código-fonte de software para web desenvolvido com php. Dissertação de Mestrado. Universidade Federal do Pará.

SINGHAL, A., Bansal, A. and Kumar, A. (2013). A Critical Review of Various Testing Techniques in Aspect-oriented Software Systems. *SIGSOFT Softw. Eng. Notes*, v. 38, n. 4, p. 1–9.

THE ECLIPSE FOUNDATION, AspectJ Documentation, Disponível em: <http://www.eclipse.org/aspectj/docs.php>. Acesso em: 09/06/2018.