**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

# MECANISMOS DE BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA PADRÕES

**DISSERTAÇÃO DE MESTRADO**

**Thiago Cassio Krug**

**Santa Maria, RS, Brasil**

**2019**

# MECANISMOS DE BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA PADRÕES

**Thiago Cassio Krug**

Dissertação apresentada ao Curso de Mestrado Programa de
Pós-Graduação em Ciência da Computação (PPGCC), Área de Concentração
em Computação, da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
**Mestre em Ciência da Computação**

**Orientador: Prof. Dr. Eduardo Kessler Piveta**

**Santa Maria, RS, Brasil**

**2019**

Krug, Thiago Cassio

Mecanismos de Busca por Oportunidades de Refatoração para Padrões / por Thiago Cassio Krug. – 2019.
125 f.: il.; 30 cm.

Orientador: Eduardo Kessler Piveta
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação, RS, 2019.

1. Refatoração para Padrões de Projeto. 2. Refatoração. 3. Padrões de Projeto. I. Piveta, Eduardo Kessler. II. Título.


E-mail: thiagockrug@gmail.com

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Programa de Pós-Graduação em Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

**MECANISMOS DE BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA PADRÕES**

elaborada por
**Thiago Cassio Krug**

como requisito parcial para obtenção do grau de
**Mestre em Ciência da Computação**

**COMISSÃO EXAMINADORA:**

**Eduardo Kessler Piveta, Dr.**
(Presidente/Orientador)

**Lisandra Manzoni Fontoura, Dr.** (UFSM)

**Maicon Bernardino da Silveira, Dr.** (UNIPAMPA)

Santa Maria, 30 de agosto de 2019.

*À minha família, esposa, colegas e amigos.*

# AGRADECIMENTOS

Meu agradecimento a todas as pessoas que contribuíram para a realização deste trabalho.

Agradeço em primeiro lugar, ao meu orientador, professor Eduardo Kessler Piveta, por me dar a oportunidade de realizar o mestrado.

Em segundo lugar, agradeço aos meus pais, à minha esposa, aos meus amigos e colegas, pelo incentivo e suporte durante todo esse período.

Agradeço também ao Instituto Federal Farroupilha pelo Programa de Incentivo à Qualificação, através do afastamento total e parcial, do qual tive a oportunidade de fazer parte.

Por fim, agradeço o apoio financeiro dado pela CAPES, o qual me proporcionou estadia em Santa Maria e demais custos de vida.

*“A mudança de foco (para padrões) vai ter um efeito profundo e duradouro no modo pelo qual escrevemos programas.”*

— WARD CUNNINGHAM E RALPH JOHNSON

# RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Santa Maria

**MECANISMOS DE BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA PADRÕES**

AUTOR: THIAGO CASSIO KRUG
ORIENTADOR: EDUARDO KESSLER PIVETA
Local da Defesa e Data: Santa Maria, 30 de agosto de 2019.

Sistemas de software devem evoluir para evitar sua degradação. A técnica de refatoração com a aplicação de padrões de projeto auxiliam na evolução consistente de software. Uma oportunidade de refatoração consiste em um trecho de código que pode ser alterado via refatoração para melhorar características de qualidade deste trecho. Dessa forma, este trabalho apresenta uma proposta de definição de mecânicas para a busca por oportunidades de refatoração para Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command*. Nele são descritos alguns indícios e refatorações que auxiliam na solução de tais problemas. Um estudo de caso foi conduzido para avaliar as oportunidades de refatoração identificadas pelas mecânicas.

**Palavras-chave:** Refatoração para Padrões de Projeto. Refatoração. Padrões de Projeto.

# ABSTRACT

Master's Dissertation
Post-Graduate Program in Computer Science
Federal University of Santa Maria

**SEARCH MECHANISMS FOR REFACTORING TO PATTERNS OPPORTUNITIES**

AUTHOR: THIAGO CASSIO KRUG
ADVISOR: EDUARDO KESSLER PIVETA

Defense Place and Date: Santa Maria, August 30th, 2019.

Software systems must evolve to avoid its degradation. The refactoring technique associated with the application of design patterns help software consistent evolution. A refactoring opportunity consists of a source code that can be altered by refactoring to improve its quality. In this sense, this study presents a set of mechanics to search refactoring opportunities to apply design patterns. Some clues and refactoring steps which aid to solve the indicated problems are described. We conducted a case study with a set of open-source projects to evaluate the opportunities identified by the mechanics.

**Keywords:** Refactoring to Design Patterns, Refactoring, Design Patterns.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ANEXOS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AJDT | *AspectJ Development Tools* |
| API | *Application Programming Interface* |
| AQL | *Aspect Query Language* |
| AST | *Abstract Syntax Tree* |
| DSL | *Domain Specific Language* |
| GoF | *Gang of Four* |
| GPA | *Grande Projeto Antecipado* |
| IDE | *Integrated Development Environment* |
| JDT | *Java Development Tools* |
| LoC | *Linhas de Código (Lines of Code)* |
| OA | *Orientação a Aspectos* |
| OO | *Orientação a Objetos* |
| OR | *Oportunidade de Refatoração* |
| UC | *Unidade de Compilação* |
| UML | *Unified Modeling Language* |
| XML | *eXtensible Markup Language* |

# SUMÁRIO

# 1 INTRODUÇÃO

Uma característica que se pode observar em sistemas de software é a necessidade de evolução. Ao adaptar, melhorar e modificar um sistema, seu projeto pode se afastar de sua concepção original, diminuindo sua qualidade (MENS; TOURWÉ, 2004). A alteração de um componente de software nem sempre está vinculada à uma inserção ou remoção de requisitos do sistema. Um trecho de código pode ser modificado para que sua legibilidade seja melhorada ou mesmo para facilitar futuras implementações. A refatoração é uma técnica que pode ser utilizada para esse fim, pois através dela os desenvolvedores podem melhorar as estruturas internas de um sistema de software sem modificar seu comportamento externamente observável (FOWLER et al., 1999; MENS; TOURWÉ, 2004).

Um objetivo para a aplicação de uma refatoração é a inserção de padrões de projeto, que são soluções reutilizáveis para problemas recorrentes em projetos de desenvolvimento software (GAMMA et al., 2000). A utilização de refatorações para padrões de projeto permite que a definição de estruturas de padrões em estágios antecipados do projeto de desenvolvimento não seja obrigatória. Dessa maneira, de acordo com a evolução dos requisitos de software e de seu projeto e implementação, podem ser inseridos novos padrões, modificados padrões existentes, e retirados padrões não mais úteis para a aplicação (KERIEVSKY, 2008).

Uma oportunidade de refatoração é um local de um artefato de software passível de ser melhorado de acordo com algum critério de qualidade. Dessa forma, a busca e identificação de oportunidades de refatoração considera deficiências e inadequações que um artefato pode possuir. Assim, uma oportunidade de refatoração é uma associação entre um artefato de software, uma inadequação, e uma refatoração (PIVETA, 2009).

Nesse contexto, o objetivo deste trabalho é propor três mecânicas para a busca de oportunidades de refatoração para padrões de projeto: Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command*. Para isso, é utilizado um processo de definição e implementação de heurísticas de busca para oportunidades de refatoração para padrões (PAULI, 2014) para guiar a criação de cada mecânica. É discutida a alteração do metamodelo (DE FAVERI, 2013; TEIXEIRA JÚNIOR, 2014) de informações necessárias para as buscas, e a realização da implementação do *plug-in Pattern Refactoring* desenvolvido. Neste contexto, as principais contribuições deste trabalho são:

- a definição das três mecânicas para a busca de oportunidades de refatoração para padrões

de projeto: Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command*;

- a definição de indícios, positivos e negativos, para as mecânicas de busca, que representam as características que uma oportunidade de refatoração para um respectivo padrão possui;

- a criação de funções heurísticas parametrizáveis para o cálculo de cada oportunidade de refatoração, baseada nos valores dos indícios extraídos do código-fonte;

- a extensão do metamodelo do *plug-in* AOPJungle, carregando mais informações pertinentes do código-fonte; e

- a implementação de um *plug-in*, chamado *Pattern Refactoring*, para a IDE Eclipse que auxilia na execução automatizada das mecânicas propostas.

Para avaliar as mecânicas propostas, um estudo de caso foi conduzido utilizando o *plug-in Pattern Refactoring* para a identificação e ordenação das oportunidades de refatoração candidatas. Em seguida, as três primeiras oportunidades de cada projeto sob teste foram aplicadas e avaliadas a aplicabilidade de cada candidata.

O restante do documento está organizado da seguinte forma:

- Capítulo 2, *Revisão de Literatura*, apresenta os conceitos necessários para o entendimento do escopo desta dissertação. São descritas as definições de refatoração e oportunidade de refatoração, assim como os padrões de projeto *Factory Method*, *Composite* e *Command*. Apresenta ainda as refatorações para padrões Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command*. Após, apresenta os trabalhos relacionados.

- Capítulo 3, *Mecânicas de Busca para Padrões*, apresenta a definição das mecânicas para a busca de oportunidades de refatoração para padrões de projeto, assim como cada um de seus indícios. São descritas as funções heurísticas de cada mecanismo e as funções de normalização.

- Capítulo 4, *Estudo de Caso*, apresenta os resultados obtidos através da avaliação das mecânicas em um estudo de caso com quatro projetos de código aberto.

- Capítulo 5, *Conclusão*, apresenta as contribuições deste trabalho e aponta algumas sugestões para trabalhos futuros.

# 2 REVISÃO DE LITERATURA

Este capítulo apresenta a revisão da literatura que trata dos conceitos e tecnologias que foram utilizadas para o desenvolvimento deste trabalho. A Seção 2.1 apresenta os principais conceitos de refatoração. A Seção 2.2 descreve as principais características dos padrões de projetos, incluindo aqueles que são utilizados neste trabalho. A Seção 2.3 aborda os fundamentos das refatorações para padrões e descreve aquelas que são utilizadas neste trabalho. A seguir, a Seção 2.4 descreve os trabalhos correlatos a este trabalho. Por fim, a Seção 2.5 realiza o fechamento do capítulo, retomando os conceitos abordados e evidenciando suas características.

## 2.1 Refatoração

Refatoração é o processo de mudança de um sistema de software de tal maneira que não altere o comportamento externo do código, melhorando a sua estrutura interna. É uma forma disciplinada de organizar código que minimiza as chances de introdução de defeitos (FOWLER et al., 1999), podendo ajudar a melhorar de maneira incremental os atributos de qualidade de um sistema (KATAOKA et al., 2001).

As refatorações são organizadas em catálogos de refatoração, sendo compostas por um nome, um contexto que pode ser aplicado, um conjunto de passos para sua aplicação, e um ou mais exemplos de sua aplicação (FOWLER et al., 1999; FOWLER, 2019).

No processo de refatoração são identificadas e removidas a duplicidade de código ou função, simplificadas lógicas condicionais, e alterados trechos de código que não estão claros. Tais modificações podem ser tão grandes quanto a unificação de hierarquias ou tão pequenas quanto a troca do nome de uma variável (KERIEVSKY, 2008).

Para garantir que o comportamento externo não foi modificado ou nenhuma falha foi introduzida, é necessária a realização de testes, sejam manuais ou automatizados. Para isso, a refatoração realizada em pequenos passos auxilia de maneira a dividir o problema em pequenas partes e testá-las de forma isolada (KERIEVSKY, 2008).

Um motivo para se refatorar um código-fonte é torná-lo mais propenso a receber novas funcionalidades. No momento de adição de uma funcionalidade, pode-se programar sem se preocupar com a adequação da nova funcionalidade ao código-fonte, ou pode-se refatorar a estrutura de maneira a acomodar essa funcionalidade. No primeiro caso, a adição da funcionalidade pode deixar um débito de projeto, o qual deverá ser pago mais tarde através do uso da

refatoração (KERIEVSKY, 2008).

O processo de refatorar continuamente um projeto pode tornar mais simples e fácil de trabalhar com ele. Buscar constantemente por problemas e solucioná-los logo após serem descobertos ajuda na organização e clareza do projeto. Assim, o código-fonte será mais fácil de manter e de estender. Além disso, códigos que não estão claros precisam ser refatorados. Dessa maneira, o conhecimento que está expresso na funcionalidade será mais acessível a outros desenvolvedores, facilitando a comunicação (KERIEVSKY, 2008).

Os processos de refatoração (MENS; TOURWÉ, 2004; PIVETA, 2009) comumente possuem quatro grupos de atividades: (i) a identificação de onde aplicar uma refatoração, (ii) a avaliação dos efeitos da refatoração na qualidade, (iii) a garantia de que a refatoração preserve o comportamento de um sistema de software, e (iv) a manutenção da consistência dos artefatos de um sistema de software refatorado.

A identificação de qual local de um artefato de software pode ser refatorado consiste em buscar por locais em um projeto em que refatorações possam ser aplicadas, levando em consideração deficiências, inadequações ou incompletudes. Dessa forma, uma oportunidade de refatoração é uma relação de associação entre uma deficiência ou limitação e uma refatoração (PIVETA, 2009).

Já a avaliação dos efeitos da refatoração na qualidade pode ser medida por meio de funções de impacto, as quais predizem as mudanças nas métricas de qualidade quando uma refatoração é aplicada (DU BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; DU BOIS, 2006; PIVETA, 2009). Du Bois (2006) e Piveta (2009) apresentam abordagens para a criação de funções de impacto em código-fonte orientado a objetos e a aspectos, respectivamente.

Existem algumas técnicas que auxiliam a garantir a preservação do comportamento observável de software. Uma maneira é a verificação dos resultados dos testes por meio de pré-condições. Entretanto, isso não é suficiente para alguns sistemas como de tempo real, embutidos, ou críticos, as quais necessitam de outros valores (regras temporais, memória, e proteção, por exemplo) (MENS; TOURWÉ, 2004).

A última etapa é a manutenção da consistência dos artefatos de software alterados pela refatoração. Tipicamente no desenvolvimento de software são criados artefatos além do código-fonte, como documento de requisitos, arquiteturas de software, modelos de projeto. Quando o código for modificado, tais artefatos devem seguir as modificações para que toda a documentação esteja consistente (MENS; TOURWÉ, 2004).

## 2.2 Padrões de Projeto

Os padrões de projeto são soluções reutilizáveis para problemas recorrentes em projetos de desenvolvimento de software (GAMMA et al., 2000). Muitos desenvolvedores quando se deparam com um problema em particular buscam resolvê-lo utilizando boas soluções que foram úteis para problemas similares, adaptando o código-fonte para o novo problema. Dessa maneira, a solução de software vai se tornando cada vez mais madura, atacando de forma mais eficiente o problema. Em consequência, padrões de classes e de comunicação entre objetos, são frequentemente identificados em sistemas orientados a objetos. Um desenvolvedor que conheça tais padrões poderá aplicá-los aos problemas, sem a necessidade de reinventar uma nova solução. Segundo Gamma *et al.* (2000, p. 20), padrões de projeto "*são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular.*"

Uma vantagem da utilização de padrões de projeto é sua estabilidade em relação à propagação de mudanças no código-fonte. Ampatzoglou *et al.* (2015) discute a probabilidade de modificação das classes que participam de instâncias de padrões de projeto quando há uma alteração no código de um sistema. A conclusão é que as classes que compõem os padrões de projeto sofrem um menor impacto quando há alterações. Dessa maneira, é possível identificar classes que necessitam de um menor esforço para a sua manutenção, pois são mais resistentes à propagação de mudanças.

A utilização de padrões de projeto auxilia o desenvolvedor a obter um projeto de software adequado mais rapidamente. Afinal, padrões de projeto são técnicas testadas e aprovadas que ajudam na escolha de soluções que vão tornar o sistema reutilizável, além de evitar escolhas que prejudiquem a reutilização. Por meio de padrões de projeto é possível obter uma melhora na documentação e manutenção de sistemas, pois fornecem uma especificação clara das interações de classes e objetos e o seu objetivo (GAMMA et al., 2000).

Geralmente, um padrão de projeto é constituído por quatro elementos essenciais: *i) o nome do padrão*, o qual descreve em uma ou duas palavras o problema, suas soluções e consequências; *ii) o problema*, que representa as situações para a aplicação do padrão; *iii) a solução* que denota os componentes, relacionamentos, responsabilidades e colaborações que integram o padrão de projeto; e *iv) as consequências*, isto é, as implicações positivas ou negativas da aplicação do padrão (GAMMA et al., 2000).

Nem sempre é trivial realizar a decomposição de um sistema em objetos. Vários aspectos devem ser considerados, tais como: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização e assim por diante. Tais aspectos podem ser conflitantes e influenciam diretamente na decomposição. Além disso, muitos objetos são identificados através do modelo de análise do problema.

Entretanto, alguns objetos não são derivados do mundo real, sendo abstrações de alto ou baixo nível, como vetores, por exemplo. Ademais, um modelo de sistema fidedigno ao mundo real apenas refletirá as necessidades atuais, mas não precisamente as futuras. Dessa maneira, as abstrações de um projeto são a chave para torná-lo flexível. Em se tratando de abstrações, os padrões de projeto podem auxiliar no seu reconhecimento, assim como na identificação de objetos que podem capturá-las. Objetos que retratam um processo ou algoritmo não se manifestam na natureza, entretanto são elementos essenciais de projetos flexíveis. Um exemplo é o padrão *Strategy*, que relata a implementação de famílias de algoritmos intercambiáveis. Tais objetos são identificados frequentemente apenas durante o esforço de tornar o projeto mais flexível, e raramente no processo de análise (GAMMA et al., 2000).

Dentre os aspectos a serem considerados na decomposição de um sistema, o nível de granularidade que os objetos devem possuir também é um desafio. Alguns padrões de projeto auxiliam nesse tema, como o *Flyweight*, que oferece suporte eficiente a grandes quantidades de objetos de granularidade fina. Já os padrões *Abstract Factory* e *Builder* também proveem objetos que apenas criam outros objetos. Dessa maneira, são descritas formas específicas de decomposição de objetos em objetos menores.

A definição das interfaces dos objetos é um passo relevante na construção de sistemas OO. A interface de um objeto é o conjunto de todas as operações que lhe podem ser solicitadas. Cada interface específica é denotada como um tipo diferente. Entretanto, é possível que objetos possuam interfaces semelhantes porém, com implementações diferentes. A definição das interfaces e o relacionamento entre elas podem ser auxiliadas pela escolha de padrões de projeto. Um exemplo é o *Visitor*, o qual define a interface que todas as classes de objetos em que visitantes podem visitar (GAMMA et al., 2000).

Nesse processo, a distinção entre tipo e classe é importante. Enquanto o tipo remete somente à interface de um objeto, a classe remete ao estado interno do objeto e sua implementação. Assim, é possível que um objeto possua vários tipos, e objetos de classes diferentes possuam o mesmo tipo. Isso influencia diretamente a herança de classe e a herança de interface.

A herança de classe trará à subclasse todas as características de estado interno e comportamento da superclasse. Entretanto, na herança de interface o objeto se compromete na implementação das operações por ela definidas (GAMMA et al., 2000).

Através da herança de interface pode-se obter dois benefícios exclusivos na criação de projetos reutilizáveis: os objetos clientes não necessitam conhecer os tipos dos objetos que eles utilizam, nem das classes que implementam estes objetos. Dessa maneira, a dependência de implementação é reduzida e é considerada um princípio de projetos reutilizáveis OO: "*Programe para uma interface, não para uma implementação*" (GAMMA et al., 2000, p. 33).

Vários padrões de projetos são restritos à diferença de tipo e classe. O *Chain of Responsability* define que os objetos devem possuir um tipo em comum, entretanto não utilizam a mesma implementação. Já no *Composite* é definida uma interface comum para o participante *Component*, e com frequência o participante *Composite* define uma codificação em comum. Já outros padrões, como *Command*, *Observer* e *Strategy* são implementados através de interfaces (GAMMA et al., 2000).

Em seguida são descritos os padrões de projeto abordados através das refatorações para padrões que foram utilizados por este trabalho: *Factory Method*, *Composite* e *Command*.

### 2.2.1 *Factory Method*

O *Factory Method* é um dos cinco padrões de criação de objetos relatados no catálogo de padrões do *Gang of Four* (GoF) (GAMMA et al., 2000). Esse padrão de projeto tem o objetivo de delegar para as subclasses a decisão de qual objeto criar baseado em uma interface. Assim, a instanciação do objeto é adiada para a subclasse correspondente. Uma utilização eficaz desse padrão é quando a classe não pode antecipar a classe de objetos que necessita ser criada, ou mesmo quando há o desejo que suas subclasses indiquem os objetos a serem criados.

Um benefício do emprego do *Factory Method* é a flexibilidade. Uma classe cliente que instancia seus objetos requeridos através de um *Factory Method* está apenas dependente da interface comum aos objetos gerados pelas subclasses. Dessa maneira, é eliminada a necessidade de utilizar as classes específicas no código do cliente. Assim, a classe cliente pode trabalhar com qualquer objeto das subclasses definidas. Além disso, é possível com a utilização das subclasses fornecer versões estendidas do objeto requerido pelo cliente.

A Figura 2.1 apresenta a estrutura do padrão *Factory Method*, que define os seguintes participantes:

- ***Product***: os objetos criados pelo *Factory Method* são definidos pela interface *Product*.
- ***ConcreteProduct***: implementa a interface *Product*.
- ***Creator***: declara o *Factory Method*, que retorna um objeto do tipo *Product*.
- ***ConcreteCreator***: redefine o *Factory Method* retornando uma instância do *ConcreteProduct*.

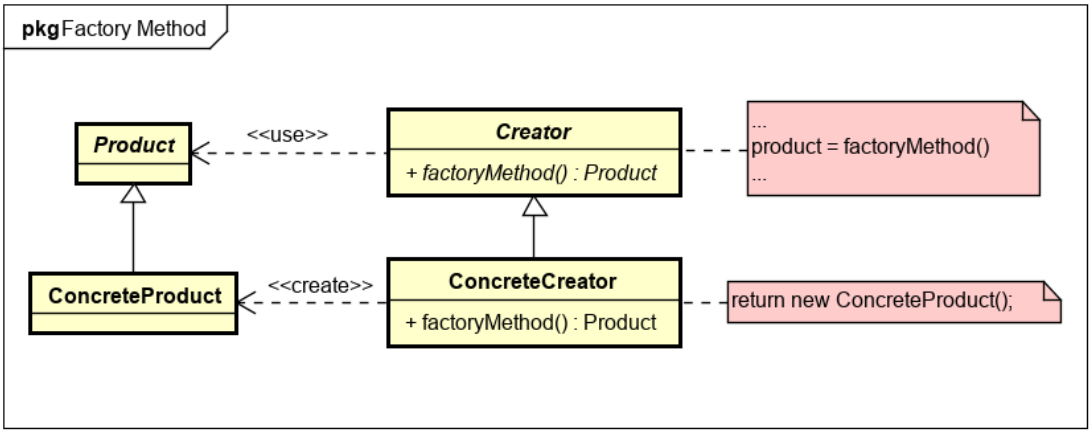


Figura 2.1 – Estrutura do Padrão de Projeto *Factory Method*.

A hierarquia de *Product* define a interface que os métodos fábrica utilizarão para a construção dos objetos necessários. Assim, é possível a inserção de novas subclasses de *Product* com comportamentos diferentes necessários para possíveis novas implementações de acordo com a demanda. Apesar de o cliente que utiliza o *Factory Method* não depender das classes concretas dos produtos, ele pode utilizar um *Factory Method* específico para a sua demanda. Assim, cada *Factory Method* é dependente de uma implementação específica de *Product*.

Existem algumas variações na implementação do *Factory Method*. Em um caso, a classe *Creator* é concreta e implementa o *Factory Method*, e as subclasses apenas implementam alguma classe de objetos que a classe *Creator* não instancia. Outro caso ocorre quando a classe *Creator* é abstrata e não fornece uma implementação para o *Factory Method*. Neste caso, é obrigatória a utilização de subclasses para a implementação do *Factory Method*. Além disso, é possível a utilização de *Factory Method* parametrizado, o qual define o tipo do objeto a ser retornado de acordo com os valores enviados por parâmetro. Por fim, existem métodos fábrica que podem retornar apenas a classe do objeto a ser criado, e não o objeto em si. Nesse caso, é necessária a utilização de reflexão para a instanciação posterior do objeto.

### 2.2.2 *Composite*

*Composite* é um dos sete padrões estruturais descritos no catálogo de padrões GoF (GAMMA et al., 2000). Por meio do padrão *Composite* é possível compor objetos em estruturas de árvore que representam hierarquias de parte-todo. A estrutura da composição é uniforme. Assim, os objetos que formam a estrutura são tratados de maneira semelhante.

O que torna a estrutura uniforme é uma classe abstrata que representa tanto uma classe primitiva quanto uma classe compositora. Assim, atributos e operações são compartilhados entre todos os membros da hierarquia. A Figura 2.2 mostra a estrutura do padrão *Composite*. Em um *Composite*, uma classe *Composite* possui vários objetos *Component*. Os objetos *Component* podem ser tanto *Leafs* como outros *Composites*. Assim, é possível a criação de uma estrutura em árvore de maneira recursiva. Os membros participantes deste padrão são:

- ***Component***: define a interface uniforme para todos os objetos da composição;
- ***Leaf***: representa os objetos-folha na composição (que não possuem objetos filhos);
- ***Composite***: representa o comportamento dos objetos que possuem filhos, além de armazená-los; e
- ***Client***: manipula os objetos da composição através da interface definida por *Component*.

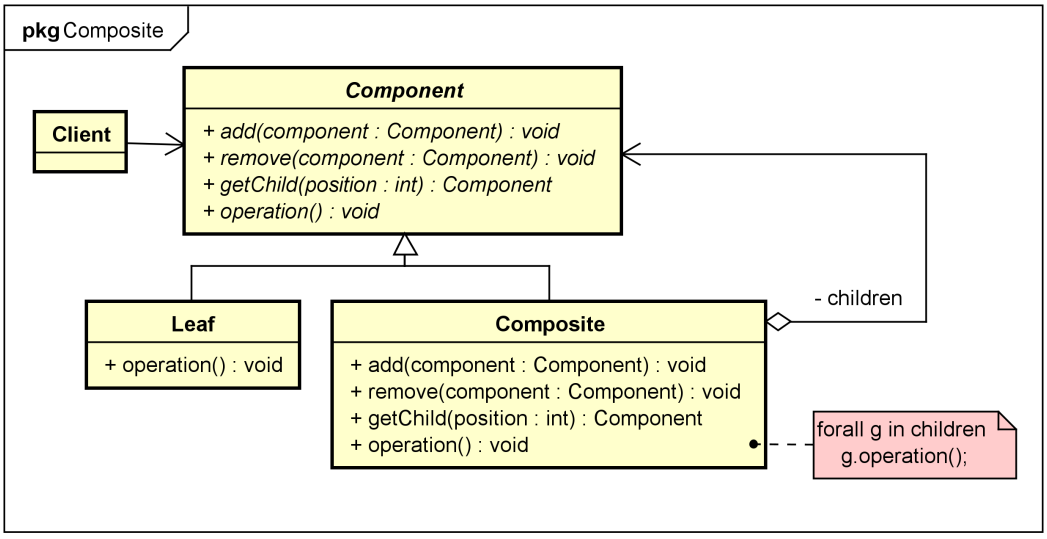


Figura 2.2 – Estrutura do Padrão de Projeto *Composite*.

Como a classe *Component* é a interface comum aos objetos da composição, ela necessita definir o conjunto de comportamentos para gerenciar os componentes-filhos. Além disso, é

possível que seja definida uma interface para o acesso ao pai de um componente da estrutura. De acordo com a necessidade, a classe *Component* pode implementar um comportamento padrão para todas as classes da composição.

A utilização do padrão *Composite* é realizada através de chamadas aos métodos definidos na interface da classe *Component*. Assim, os clientes podem realizar a manipulação dos objetos na estrutura. Caso a solicitação do *Client* seja realizada a uma classe *Leaf*, tal solicitação é realizada imediatamente pela classe. Se a solicitação for realizada a uma classe *Composite*, a mesma processa a solicitação possibilitando que tal chamada possa ser repassada para os componentes-filhos na estrutura.

Um *Client* que use uma composição poderá estar manipulando objetos primitivos (*Leaf*) ou mesmo objetos compostos (*Composite*) sem que necessariamente saiba com qual tipo está lidando. O código-fonte no *Client* pode se tornar mais enxuto com o uso da composição, pois como a composição compartilha a mesma interface, as estruturas serão tratadas de maneira uniforme.

Em um *Composite*, a inserção de novas estruturas requer um menor esforço de implementação. Como as estruturas novas devem seguir a mesma interface *Component*, a estrutura já codificada permanecerá a mesma, apenas com a necessidade da adição do código-fonte dos novos componentes.

### 2.2.3 *Command*

O padrão *Command* é um padrão comportamental do catálogo GoF (GAMMA et al., 2000). Este padrão pode ser usado para encapsular uma solicitação como um objeto, sendo possível a parametrização de acordo com a solicitação, assim como enfileirar solicitações, fazer registro (*log*), e dar suporte a operações que possam ser desfeitas.

Este padrão permite um desacoplamento das ações a serem executadas de seus clientes. Assim, o desenvolvedor pode definir um tipo de comando diferente de acordo com valores de parâmetros ou mesmo de acordo com o tipo da solicitação. Em seguida, a classe cliente apenas realiza a chamada do comando informando o tipo da solicitação.

Em casos onde uma função a ser executada não é conhecida, ou mesmo não é de responsabilidade da estrutura cliente, o padrão *Command* pode ser implementado. Assim, não há a necessidade que o desenvolvimento de uma ação seja colocado dentro do cliente, permitindo inclusive que essas ações possam ser chamadas de maneira desacoplada no restante do projeto.

Caso deseje-se que uma ação seja desfeita, é necessário armazenar o estado atual do sistema que será alterado dentro do *ConcreteCommand*. Assim, caso seja solicitada a volta ao estado anterior, esse estado armazenado no *ConcreteCommand* será utilizado.

A Figura 2.3 apresenta a estrutura do padrão *Command*. Podem ser identificados os seguintes participantes:

- ***Command***: define uma interface para a execução de uma operação, onde suas subclasses implementarão, e as classes clientes utilizarão;
- ***ConcreteCommand***: implementa a ação concreta a ser chamada pelo *Invoker*, ou realizada a chamada da ação do *Receiver*. Para isso, recebe o *Receiver* como parâmetro, onde o *ConcreteCommand* utilizará para realizar a ação;
- ***Client***: representa um cliente que vai instanciar um *ConcreteCommand* e passar um *Receiver* como parâmetro, armazenado na classe *Invoker*;
- ***Invoker***: solicita ao *Command* a execução da solicitação; e
- ***Receiver***: tem o conhecimento da ação a ser realizada, ou contém as informações necessárias para que a ação seja realizada.

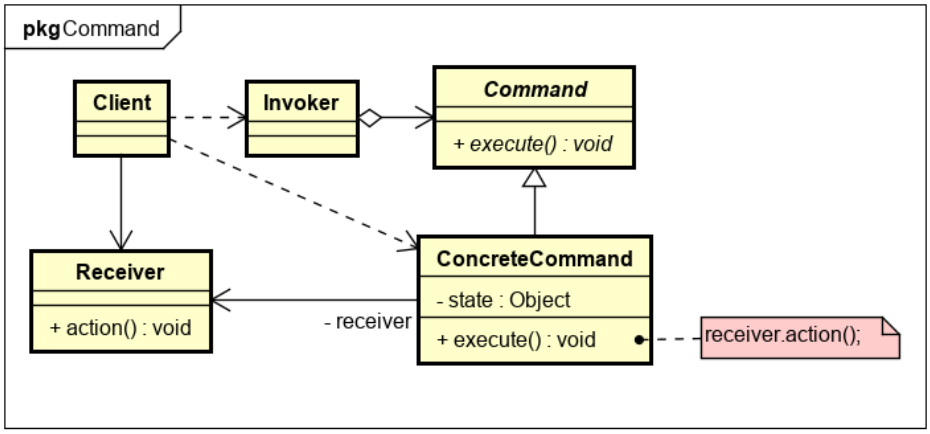


Figura 2.3 – Estrutura do Padrão de Projeto *Command*.

Com o uso do padrão *Command* é possível criar comandos compostos, no qual se tem alguns *CommandConcrete* que fazem ações simples, e um comando composto que utiliza esses comandos simples para realizar ações complexas. Além disso, a adição de novos comandos é simples, apenas adicionando a classe de comando concreto e estendendo a classe *Command*.

O *Command* também permite especificar, enfileirar e executar solicitações em tempos diferentes. O tempo de vida de um objeto *Command* pode ser diferente da solicitação original. Logo, é possível que o mesmo objeto atenda a diferentes solicitações de diferentes clientes.

## 2.3 Refatorações para Padrões de Projeto

Os motivos para utilizar uma refatoração para um padrão de projeto podem ser semelhantes aos motivos para utilizar uma refatoração de baixo nível, como *Extrair Método* (FOWLER et al., 1999). Tais motivos podem incluir a remoção de código duplicado, comunicar melhor a intenção de uma classe e simplificar um trecho de código.

A utilização de refatorações para padrões de projeto auxilia no sentido de não ser obrigatória a definição de estruturas de padrões em estágios antecipados do projeto de desenvolvimento. Dessa maneira, de acordo com a evolução dos requisitos de software e da implementação do código-fonte, podem ser inseridos novos padrões, modificados padrões existentes, e retirados padrões não mais úteis para a aplicação. Assim, as refatorações para padrões servem como uma alternativa a um grande projeto antecipado (GPA) (KERIEVSKY, 2008).

Um projeto de software necessita, frequentemente, ser reorganizado ou refatorado. Os padrões de projeto auxiliam nessa atividade pois determinam como deve ser a reorganização, reduzindo o volume de refatorações a serem aplicadas. Dessa forma, os padrões de projeto atendem muitas das estruturas resultantes da aplicação de refatorações. Assim, os padrões se tornam alvos desejáveis da utilização das refatorações (GAMMA et al., 2000).

Fowler (1999, p. 98) define sucintamente o objetivo das refatorações para padrões: "*Há uma relação natural entre padrões e refatorações. Padrões são onde você quer estar; refatorações são modos de chegar lá.*"

Kerievsky (2008) define um catálogo com 27 refatorações para padrões de projeto, dividido em refatorações de criação, simplificação, generalização, proteção, acumulação e utilitárias. Tais refatorações possuem uma estrutura semelhante à utilizada por Fowler et al. (1999). Tal estrutura é composta por (KERIEVSKY, 2008):

- Nome: define o nome da refatoração para padrão de projeto;
- Resumo: define a estrutura da transformação de um trecho de código para um padrão de projeto. Tal estrutura é representada através de diagramas da UML (*Unified Modeling Language*);

- Motivação: descreve alguns motivos pela qual a refatoração poderia ser utilizada;
- Benefícios e desvantagens: descreve de forma sucinta algumas vantagens e desvantagens de se utilizar a refatoração;
- Mecânica: define os passos que devem ser realizados para a utilização da refatoração;
- Exemplo: apresenta a aplicação da refatoração em um exemplo prático; e
- Variações: documenta algumas possíveis variações da refatoração.

As seções a seguir descrevem as refatorações para padrões de projeto que serão utilizadas por este trabalho. As refatorações consideradas são *Encapsular Classes com Factory* (Seção 2.3.1), *Substituir Árvore Implícita por Composite* (Seção 2.3.2) e *Substituir Envio Condicional por Command* (Seção 2.3.3).

### 2.3.1 Encapsular Classes com Factory

A refatoração *Encapsular Classes com Factory* (KERIEVSKY, 2008) tem como objetivo substituir uma ou várias instanciações diretas a classes por uma *Factory*, que possui um conjunto de *Factory Methods* (GAMMA et al., 2000). Sua aplicação consiste em encapsular cada classe tornando seus construtores não-públicos, e levando o cliente a criar as instâncias necessárias através de uma *Factory*.

A Figura 2.4 apresenta um exemplo da refatoração *Encapsular Classes com Factory*. O problema mostrado na figura consiste em uma hierarquia de classes que auxiliam no mapeamento de atributos do banco de dados para variáveis de instância. Estão contidas a classe abstrata *AttributeDescriptor* e as subclasses *BooleanDescriptor* (descreve atributos boleanos), *DefaultDescriptor* (descreve qualquer atributo) e *ReferenceDescriptor* (descreve referências a outras tabelas). As três subclasses são usadas diretamente pela classe *Client*.

A Figura 2.5 mostra o exemplo após a aplicação da refatoração. Os construtores das subclasses foram alterados para a visibilidade apenas dentro do pacote. Para cada funcionalidade exercida pelas subclasses foi criado um método público na classe *AttributeDescriptor*. Dessa maneira, a dependência de implementação é reduzida, pois a classe *Client* nesse momento somente usa os métodos fábrica disponíveis em *AttributeDescriptor*. Além disso, houve uma melhora na intenção de cada função.

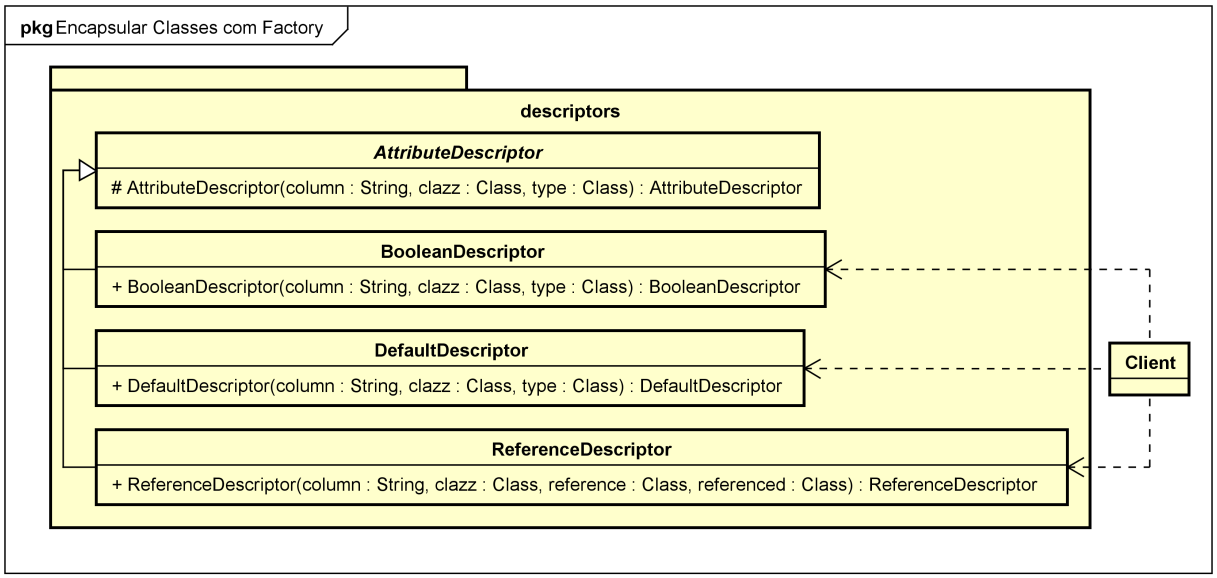


Figura 2.4 – Refatoração Encapsular Classes com *Factory*, antes.

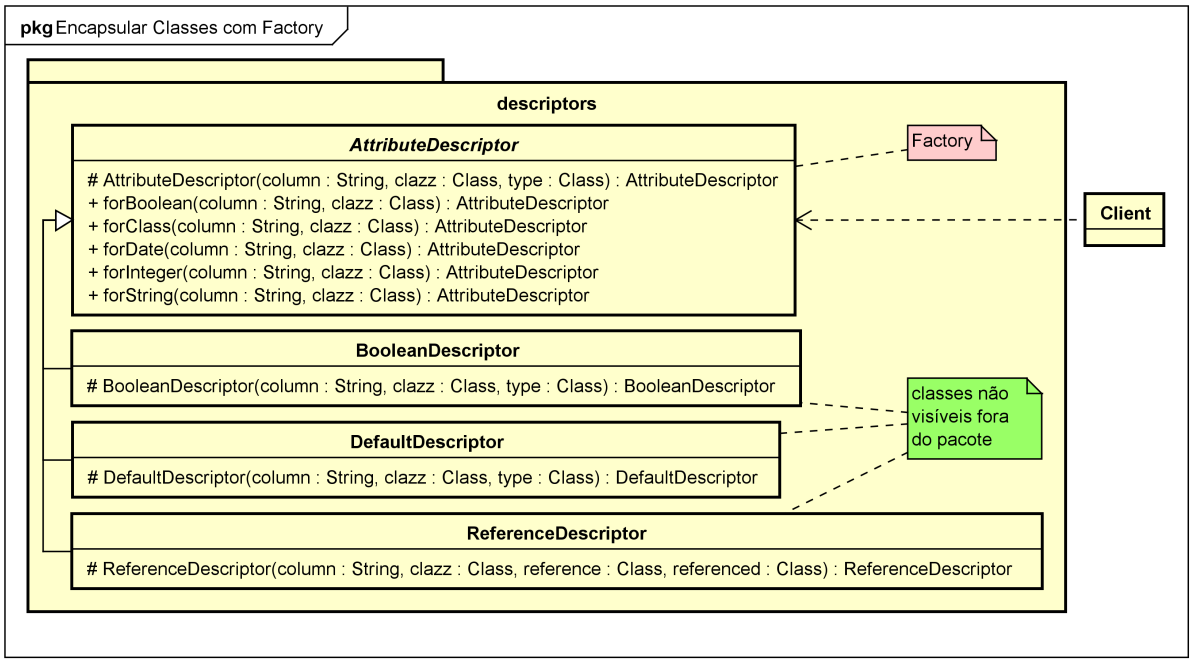


Figura 2.5 – Refatoração Encapsular Classes com *Factory*, depois.

Classes agrupadas em um mesmo pacote e que implementam a mesma interface comumente possuem um propósito geral único, entretanto cada classe implementa alguma característica única. Um cliente instancia diretamente uma classe se ele necessita saber da sua existência. Entretanto, neste cenário de classes agrupadas, talvez seja interessante não saber a existência das classes, mas apenas da interface que elas implementam.

Esta refatoração reduz a complexidade na criação de tipos de instâncias ao disponibilizar

o conjunto de opções através de métodos que revelam a sua intenção. Além disso, reduz a quantidade de informação do pacote ao ocultar as classes que não necessitam ser públicas. Ademais, auxilia na redução de dependência de implementação, um dos princípios de projetos reutilizáveis OO. Entretanto, é necessário criar um método de criação novo para cada nova classe.

### 2.3.2 Substituir Árvore Implícita por Composite

A refatoração *Substituir Árvore Implícita por Composite* (KERIEVSKY, 2008) objetiva a troca de uma estrutura de árvore por um *Composite* (GAMMA et al., 2000). Uma árvore implícita é um trecho de código que utiliza uma estrutura lógica em árvore, embora tal estrutura não seja representada como uma árvore. A Figura 2.6 apresenta uma árvore implícita composta por *tags* no formato XML (*eXtensible Markup Language*).

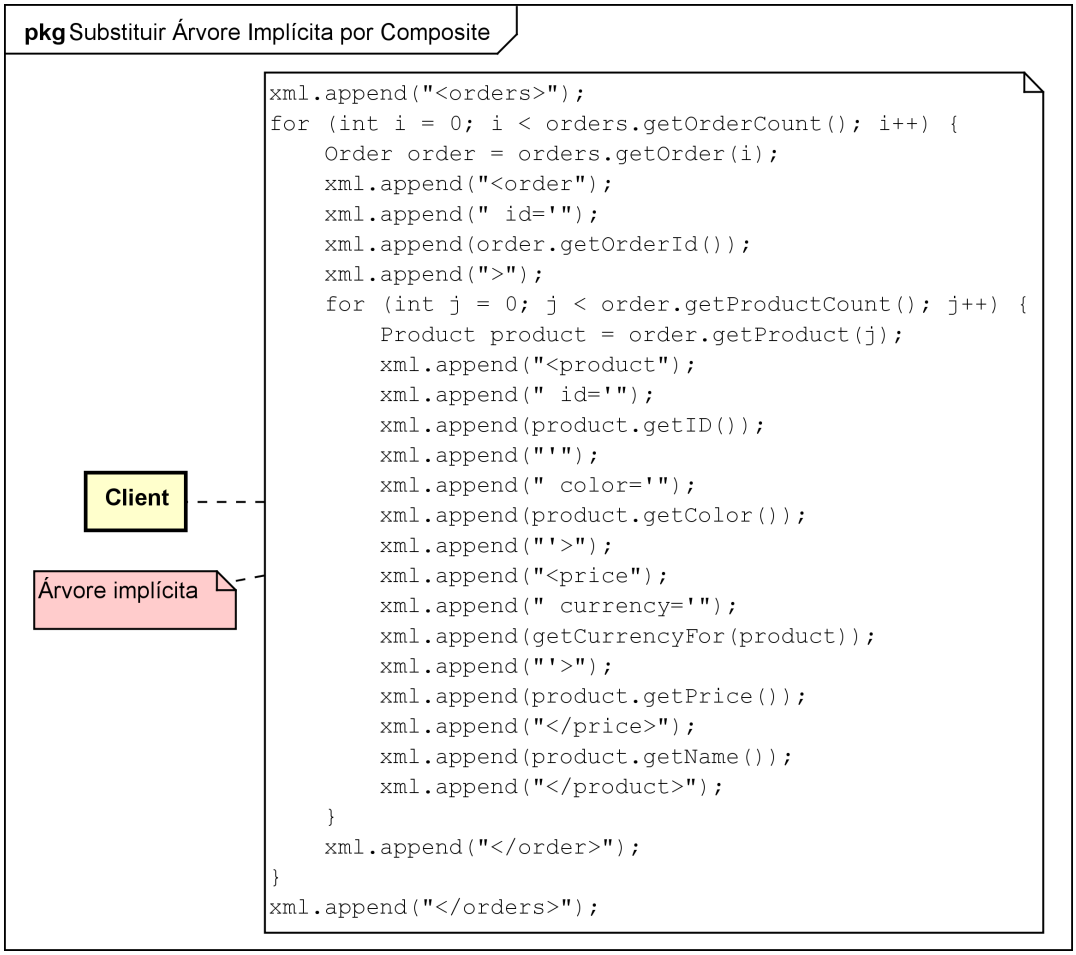


Figura 2.6 – Refatoração Substituir Árvore Implícita por *Composite*, antes.

A alteração de uma árvore implícita para um *Composite* pode tornar o código mais en-

xuto e com um número menor de linhas. Dessa forma, reduz o acoplamento do *Client* com a construção da árvore, e utiliza no lugar uma estrutura de composição de nós para a construção da árvore. Com *Composite* é possível encapsular sentenças repetitivas para adição, remoção e formatação de nós. Além disso, a manipulação de novos nós é realizada de maneira generalizada, sem um grande acoplamento. Assim, a construção das árvores pelos *Clients* é realizada com a inserção de poucas linhas de código.

Entretanto, as vantagens da utilização do *Composite* se justificam quando há uma quantidade considerável de árvores implícitas. Caso contrário, o uso do *Composite* pode causar um aumento da complexidade do código.

A Figura 2.7 apresenta uma solução para o problema de árvore implícita mencionado na Figura 2.6. É criada a estrutura *TagNode* que implementa as características de um *Composite*. A partir desta estrutura cada nó da árvore implícita é substituída por uma instância de *TagNode* e criada uma relação entre nós pai e filhos. Assim, a responsabilidade da criação e montagem da árvore implícita é delegada para o *Composite*, desacoplando a lógica de construção do *Client*. Entretanto, a classe *Client* se torna dependente do *Composite*.

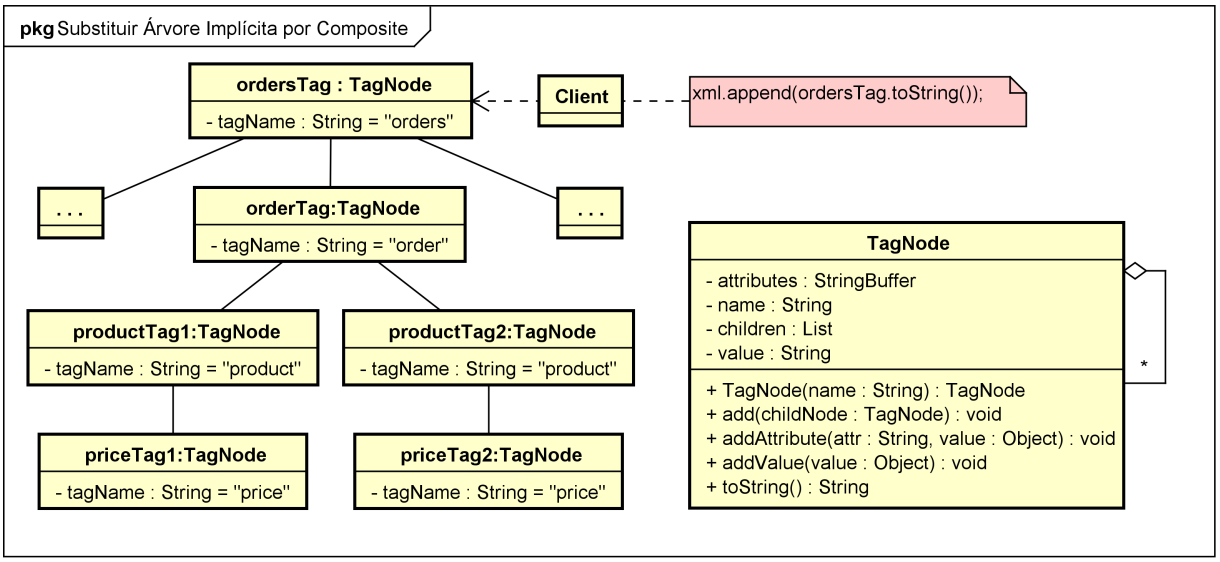


Figura 2.7 – Refatoração Substituir Árvore Implícita por *Composite*, depois.

### 2.3.3 Substituir Envio Condicional por Command

A refatoração *Substituir Envio Condicional por Command* (KERIEVSKY, 2008) tem o objetivo de substituir uma lógica condicional que envia requisições ou executa ações por um conjunto de comandos que as executam. Para isso, é necessário transformar cada ação em um

comando, armazená-los, e substituir a lógica condicional por um código que obtenha e execute os comandos.

A Figura 2.8 apresenta uma oportunidade para a aplicação dessa refatoração. Há nesse código da classe *CatalogApp* um conjunto de várias sentenças condicionais (“else if”), onde cada condição serve para realizar um roteamento de requisições e manipulação.

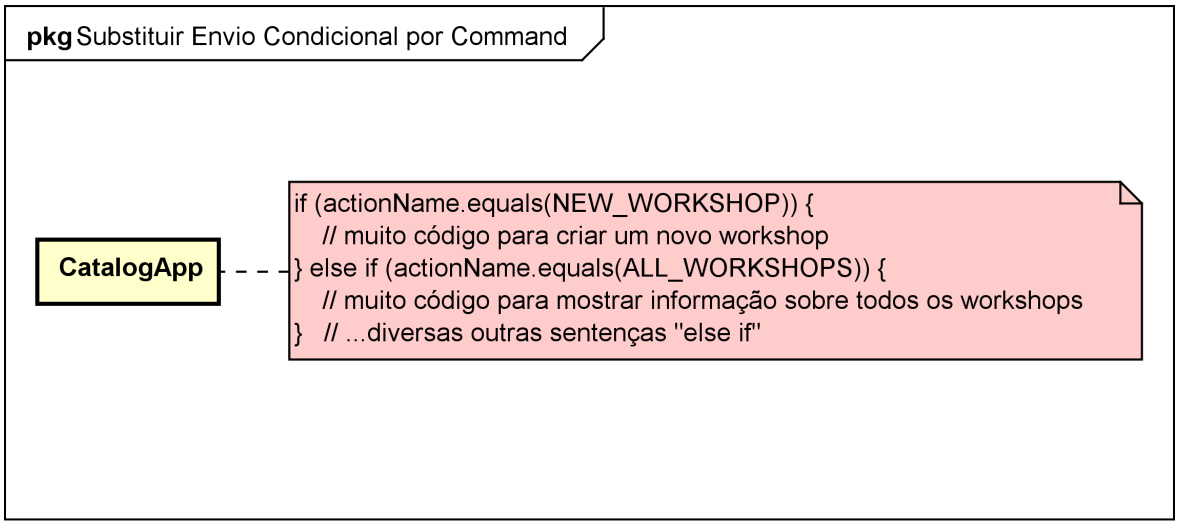


Figura 2.8 – Refatoração Substituir Envio Condicional por *Command*, antes.

A Figura 2.9 mostra o exemplo anterior refatorado. É criada a classe abstrata *Handler*, que representa um comando de manipulação genérico. E cada ação contida em uma sentença condicional foi extraída para uma classe de comando concreta. No caso do exemplo, as classes *NewWorkshopHandler* e *AllWorkshopHandler* foram criadas, ambas sendo subclasses de *Handler*. Na classe *CatalogApp*, é criado um *Map* de *handlers* com as instâncias de cada comando concreto associado com o seu valor para rotamento. Quando a classe *CatalogApp* realizar o roteamento, a mesma buscará o objeto de comando no *Map* e chamará o método *execute*;

A refatoração *Substituir Envio Condicional por Command* auxilia na redução de um corpo de código extenso, transformando as várias sentenças condicionais em classes com comportamentos bem definidos. Além disso, ela dispõe de mecanismos simples para adicionar novos comandos, sendo necessário apenas a implementação de uma nova classe para cada novo comando desejado.

Essa refatoração permite que o usuário realize ações diferentes em tempo de execução num mesmo contexto. O usuário pode escolher qual(is) comando(s) ele deseja e solicitar a execução deles. Assim, cada comando concreto será escolhido em tempo de execução automa-

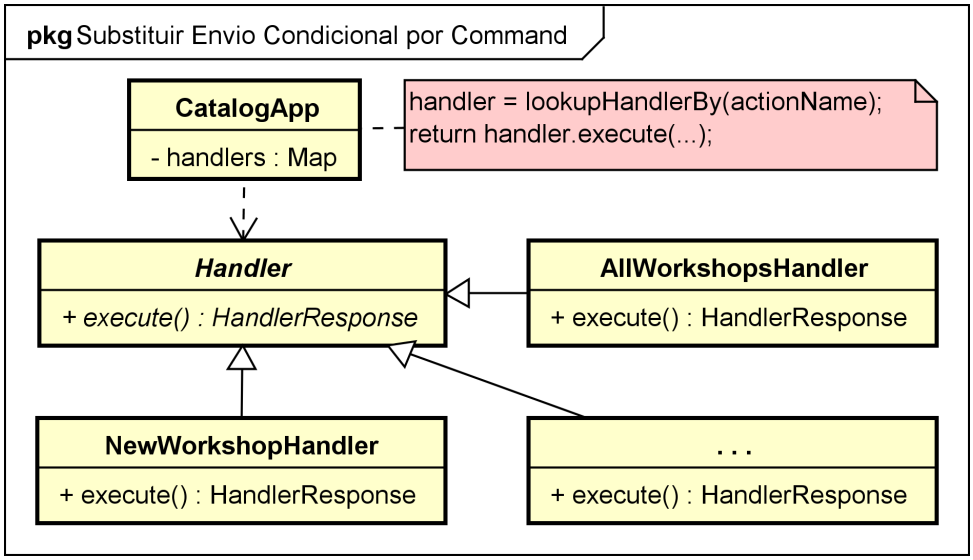


Figura 2.9 – Refatoração Substituir Envio Condicional por *Command*, depois.

ticamente pelo seu valor de roteamento.

Entretanto, utilizar a refatoração em locais onde as sentenças condicionais são suficientes pode apenas complicar o projeto. Caso a quantidade de linhas de código (*lines of code*, LoC) ou a complexidade das ações realizadas forem triviais essa refatoração pode não ser útil.

## 2.4 Trabalhos Relacionados

Existem vários trabalhos na literatura de refatoração e de busca de oportunidades de refatoração. Mens e Tourwe (2004) apresentam uma visão geral dos trabalhos até então existentes no campo de refatoração. Os trabalhos são comparados por aspectos como as atividades de refatoração suportadas, as técnicas que apoiam tais atividades, os artefatos de software que foram refatorados, características das ferramentas desenvolvidas, e o efeito do uso de refatoração no processo de desenvolvimento.

Mais recentemente, Abebe e Yoo (2014) aplicam uma revisão sistemática da literatura para a verificar as tendências, oportunidades e desafios na área de refatoração de software. É realizada a classificação dos trabalhos em dez grupos distintos. Os principais esforços realizados no grupo de padrões de projeto é a utilização de padrões como auxílio na identificação de uma refatoração, na utilização de um padrão na ferramenta de refatoração, e a busca por antipadrões de projeto.

A seguir, Al Dallal (2015) realiza uma revisão sistemática da literatura por trabalhos

que buscam identificar oportunidades de refatoração em códigos orientados a objetos. Os resultados indicam que o campo de pesquisa em identificação de oportunidades de refatoração está altamente ativa. As refatorações mais utilizadas fazem parte do catálogo criado por Fowler (FOWLER et al., 1999), as quais se destacam *Extrair Classe* e *Mover Método*. As abordagens de identificação de oportunidades de refatoração orientada a métricas de qualidade, a pré-condições, e a *clustering* foram as mais utilizadas. Para a avaliação das abordagens utilizadas pelos autores dos trabalhos pesquisados, as técnicas mais utilizadas foram avaliações baseadas na intuição dos avaliadores, e avaliações baseadas na qualidade.

É possível verificar que mais de 60% dos trabalhos avaliados por Al Dallal (2015) realizam os testes das abordagens em até dois sistemas. Dentre os sistemas mais utilizados se destacam o JHotDraw[1], Apache Ant[2] e ArgoUML[3]. A grande utilização do JHotDraw se justifica pelo seu conhecido bom projeto utilizando padrões de projeto.

Na pesquisa por trabalhos ligados à refatoração de sistemas orientados a objetos e a aplicação de padrões de projeto, três trabalhos foram considerados em destaque.

No primeiro trabalho é apresentado o *Reflective Refactoring* ($R^2$) (KIM; BATORY; DIG, 2015), um pacote que auxilia na automação de inserção de padrões de projeto. Para cada padrão de projeto é criado um *script* para sua criação. $R^2$ usa os conceitos da reflexão: os elementos da linguagem Java se tornam instâncias das classes definidas no pacote $R^2$. A partir dos elementos carregados, o *script* adiciona e altera as informações do código-fonte de acordo com o padrão de projeto escolhido. Os testes realizados revelaram que o $R^2$ pode ser aplicado em sistemas não-triviais e ainda auxilia na produtividade para a inserção de padrões de projeto.

No segundo trabalho é relatada a ferramenta AROS[4] (PAULI, 2014; PAULI; PIVETA, 2014), que identifica oportunidades de refatoração para padrões. São criados mecanismos para a busca de oportunidades para os padrões *Strategy*, *Factory Method*, *Template Method*, *Creation Method* e *Chain Constructors*. Através da avaliação realizada em três ferramentas é possível verificar que as oportunidades identificadas incrementaram atributos de qualidade, caracterizando uma precisão relevante nas mecânicas criadas. O presente trabalho apresenta mais três refatorações para padrão, contribuindo para a pesquisa na área. Pauli (PAULI, 2014) define as

[1] JHotDraw é um *framework* GUI em Java para a criação de software gráfico. http://www.jhotdraw.org/

[2] Apache Ant é uma biblioteca e uma ferramenta de linha de comando que objetiva a automação de compilação e construção de software através de arquivos com alvos e pontos de extensão dependentes entre si. http://ant.apache.org/

[3] ArgoUML é uma ferramenta de modelagem de diagramas UML, com suporte a todos os padrões da UML 1.4. http://argouml.tigris.org/

[4] http://sourceforge.net/projects/aros2dp/

mecânicas de maneira rígida, em que não é possível que o usuário informe os parâmetros para a personalização dado o contexto de seu projeto. Entretanto, este trabalho define alguns indícios para cada mecânica, que podem contribuir positivamente ou negativamente na busca das oportunidades. Ademais, são definidas funções heurísticas parametrizáveis pelo usuário que possibilitam uma melhor ordenação das oportunidades candidatas de acordo com as características do projeto sendo refatorado.

Por fim, o terceiro trabalho apresenta o *plug-in* Auto-SCST (VEDURADA; NANDIVADA, 2018), o qual busca por oportunidades de refatoração para *Subclass* e *State*. É introduzido o conceito de campos de controle, que dependendo do comportamento da classe, pode levar a uma ou a outra refatoração citada. A ferramenta é avaliada em oito projetos onde identifica oportunidades relevantes para serem aplicadas as refatorações. As mecânicas para a busca das oportunidades de refatoração são estáticas. Enquanto o presente trabalho permite o desenvolvedor informar os valores de pesos e parâmetros que seja necessário.

## 2.5 Fechamento do Capítulo

Este capítulo apresentou os conceitos necessários para o entendimento do trabalho realizado. A refatoração é uma técnica que auxilia na evolução de um sistema existente evitando que o mesmo se degrade. As refatorações estão organizadas em catálogos que descrevem suas características e mecanismos. Além disso, foram descritas as atividades necessárias para a aplicação de uma refatoração. Os padrões de projeto constituem estruturas que auxiliam na resolução de problemas recorrentes no desenvolvimento de software. Dentre eles foram destacados os padrões de projeto *Factory Method*, *Composite* e *Command*, que serão utilizados durante este trabalho. Em seguida, conceitos relacionados a refatorações para padrões foram discutidos, e após apresentadas as refatorações que serão utilizadas neste trabalho: *Encapsular Classes com Factory*, *Substituir Árvore Implícita por Composite* e *Substituir Envio Condicional por Command*. Por fim, foram descritos os trabalhos relacionados no que tange o estado da arte de refatorações, e também os trabalhos considerados mais relevantes em relação a este trabalho. O capítulo seguinte trata sobre a definição e especificação das mecânicas de busca para cada refatoração para padrões.

# 3 MECÂNICAS DE BUSCA PARA PADRÕES

A proposta deste trabalho é definir mecanismos para a busca de oportunidades de refatoração para padrões de projeto. O primeiro passo para a realização desta proposta é a absorção e documentação dos conceitos fundamentais que regem as áreas de refatoração, padrões de projeto, e refatorações para padrões. Tais conceitos, técnicas e atividades foram apresentadas no Capítulo 2.

Em seguida, é necessário seguir um processo que defina como uma busca por oportunidades de refatoração deve ser realizada. Pauli (2014) apresentada um conjunto de atividades para a busca de oportunidades de refatoração. A Figura 3.1 mostra o diagrama de atividades do processo. O processo é dividido em duas etapas: definição da oportunidade de refatoração e a implementação e avaliação da mesma.

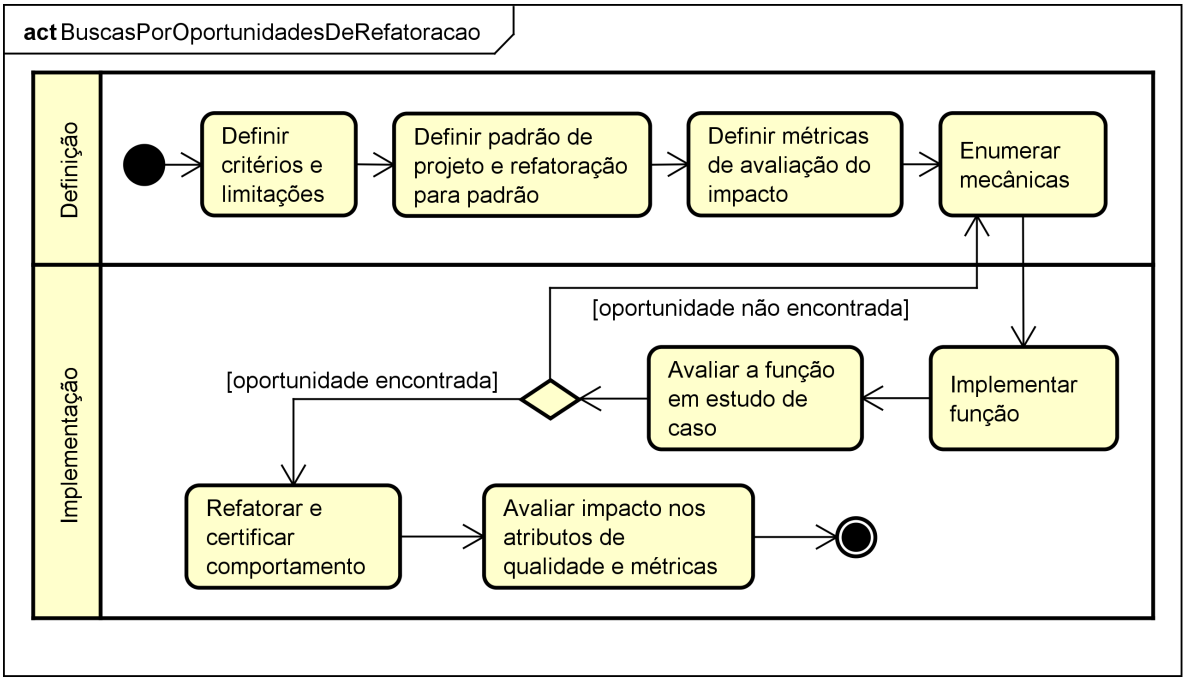


Figura 3.1 – Atividades envolvidas no processo de buscas por oportunidades de refatoração (PAULI, 2014).

De acordo com a Figura 3.1, em primeiro lugar é necessário definir indícios, critérios ou limitações que podem caracterizar uma oportunidade de refatoração para um padrão de projeto. A partir de cada oportunidade definida, são identificadas as refatorações para padrão que solucionam ou auxiliam na melhoria do código-fonte. Em seguida, deve ser escolhido um conjunto de métricas para avaliar se a aplicação da refatoração melhorará atributos de qualidade de

software. Após, uma série de mecanismos deve ser criada para encontrar os indícios definidos anteriormente.

As mecânicas de busca por oportunidades de refatoração para as refatorações Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command* foram criadas baseadas em tal processo.

## 3.1 Mecânica de busca para Encapsular Classes com *Factory*

Quando uma classe cliente instancia diretamente classes que situam-se em um pacote com uma interface em comum, pode-se caracterizar uma oportunidade para a refatoração Encapsular Classes com *Factory* como a Figura 3.2 apresenta. Nesta estrutura, a classe *Client* referencia todas as subclasses de *AttributeDescriptor*, utilizando-as para a descrição de dados provenientes de um banco de dados.

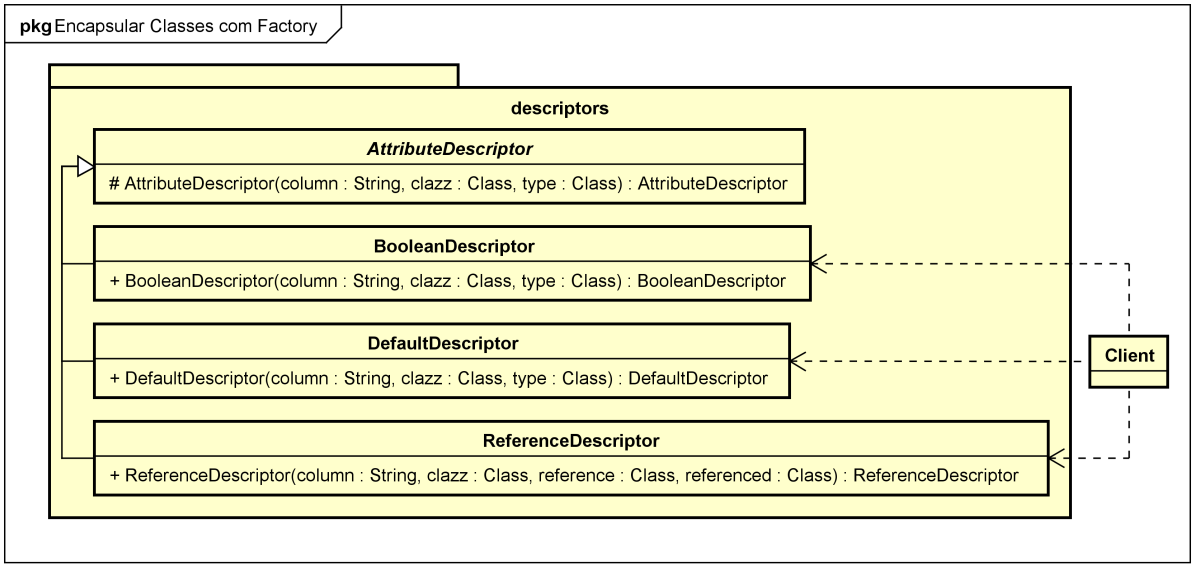


Figura 3.2 – Indícios de uma oportunidade para a refatoração *Encapsular Classes com Factory*.

Com a aplicação da refatoração Encapsular Classes com *Factory*, a classe *Client* ficará dependente apenas da classe *AttributeDescriptor*, que instanciará objetos de suas subclasses de acordo com as chamadas realizadas na *Client*. Assim, os métodos disponíveis na classe de *Factory* possuirão uma melhor descrição de sua funcionalidade. Além disso, é possível diminuir a quantidade de classes explícitas sem necessidade de um mesmo pacote. Por fim, auxilia no princípio de projetos: "programe para uma interface" (GAMMA et al., 2000).

Para a implementação da busca das oportunidades de refatoração para esse padrão foi

criada a mecânica descrita no Algoritmo 1.

**Algoritmo 1:** MECÂNICA DE BUSCA PARA ENCAPSULAR CLASSES COM *Factory*

**Entrada:** $CIs$ = {conjunto das classes instanciadas em um projeto}
**Saída:** $Ss$ = {candidatos a oportunidades de refatoração}

```
1  início
2      para cada classe CI do conjunto CIs faça
3          SCs + ← obtém superclasse de CI;
4      fim
5      para cada superclasse SC do conjunto SCs faça
6          numeroSubtipos ← quantidade de subtipos de SC;
7          se numeroSubtipos >= 2 então
8              visitor ← valor para características do padrão Visitor em SC;
9              numeroSubtiposNoMesmoPacote ← quantidade de subtipos de SC
                que estão no mesmo pacote;
10             numeroModificadoresPublicos ← quantidade de modificadores
                públicos dos subtipos de SC que não fazem parte da interface de SC;
11             numeroDeChamadasSubtipos ← quantidade de chamadas diretas aos
                subtipos de SC
12             resultado ← aplica a Função Heurística 3.1 para Encapsular Classes
                com Factory;
13             Ss + ← SC e resultado; (adiciona a superclasse SC e o resultado ao
                conjunto de candidatos a oportunidades de refatoração Ss)
14         fim
15     fim
16     retorna Ss
17 fim
```

De acordo com o algoritmo, são obtidas todas as classes de um projeto que são instanciadas. Entretanto, são adicionadas ao conjunto $CIs$ somente classes instanciadas que estão fora do pacote da classe cliente, ou seja, da classe que realiza a instanciação. Isso ocorre porque a refatoração Encapsular Classes com *Factory* prevê que as classes encapsuladas serão visíveis apenas para o pacote ao qual pertencem. Se elas são utilizadas apenas dentro do seu próprio pacote, não há necessidade da aplicação da refatoração.

O Algoritmo 1 inicia extraindo a superclasse de cada classe $CI$ e armazenando-a na lista de superclasses $SCs$. Após, a lista $SCs$ é iterada e, para cada uma das superclasses $SC$, é contada a quantidade de subtipos que possui. Caso a classe $SC$ possua dois ou mais subtipos, é verificado se essa classe possui características do padrão *Visitor*.

Durante a execução se verificou a presença de muitas ocorrências de falsos positivos de hierarquia de classes que implementam o padrão *Visitor* (GAMMA et al., 2000). As carac-

terísticas que a mecânica proposta busca são semelhantes às da implementação desse padrão. Assim, tornou-se necessário verificar durante a busca se o conjunto de classes implementa o padrão.

Quando desenvolvedores criam um conjunto de classes que implementam o *Visitor* é comum o uso da palavra *"Visitor"* no nome da classe e *"visit"* em algum método. Aproveitando-se dessa forma de nomenclatura é feita a verificação buscando as palavras em seus respectivos lugares. Caso seja encontrada a palavra *"Visitor"*, é somado 2 ao resultado, e é somado 1 para cada método que inicie com a palavra *"visit"*. O estudo de caso foi realizado com a implementação deste viés mais simples, porém pode ser usado outro algoritmo ou forma de verificação que resulte em uma detecção mais eficiente.

Em seguida, são contados quantos dos subtipos da superclasse $SC$ se encontram no mesmo pacote ou abaixo na hierarquia de pacotes como é apresentado no Algoritmo 1. Uma maneira de organizar as classes em um projeto é de acordo com a sua funcionalidade. Essa organização vai priorizar o agrupamento de classes com funções semelhantes e que atuam no mesmo contexto. Assim, é mais comum encontrar famílias de classes que acumulam funções semelhantes dentro de uma mesma hierarquia de pacotes. Caso uma subclasse esteja em um pacote diferente, pode significar que a mesma possui um comportamento que não é mais tão compatível com o comportamento da superclasse.

Após, é contada a quantidade de métodos públicos que as subclasses da classe $SC$ possuem que não são herdadas. Um número maior de métodos públicos que não fazem parte da interface da classe $SC$ significa que essa subclasse está realizando mais funcionalidades que as definidas em $SC$. Sendo assim, a publicidade da subclasse pode ser importante para o restante do projeto, não sendo interessante ser refatorada e ocultada das demais classes, como a refatoração Encapsular Classes com *Factory* prevê. Vale notar que os métodos construtores não são considerados na contagem, pois os mesmos podem ser públicos para que a refatoração seja realizada.

Caso uma classe seja chamada de maneira literal, como *Pessoa.class* por exemplo, pode significar que a sua existência pública é necessária. Dessa forma, são contadas quantas vezes as subclasses da classe $SC$ são chamadas literalmente e armazenadas na variável $numeroDeChamadasSubtipos$, como é apresentado no Algoritmo 1.

Depois de aferidos os valores para cada variável, é aplicada a função heurística conforme apresentado pela Função 3.1.

$$
\begin{aligned}
&resultado(sc : Classe, p : ParametrosEPesos) = \\
&\quad \big(normalizarReflexao(numeroSubtipos(sc), p.parametro1) * p.peso1 \\
&+ normalizarReflexao(numeroSubtiposNoMesmoPacote(sc), p.parametro2) * p.peso2\big) \\
&\quad - \\
&\big(normalizarRetilineo(numeroModificadoresPublicos(sc), p.parametro3) * p.peso3 \\
&\quad + normalizarRetilineo(visitor(sc), p.parametro4) * p.peso4 \\
&+ normalizarRetilineo(numeroDeChamadasSubtipos(sc), p.parametro5) * p.peso5\big)
\end{aligned}
\tag{3.1}
$$

Onde:

- *sc*: é uma superclasse $SC$ do conjunto de superclasses $SCs$;
- *p*: é um registro que contém os parâmetros e pesos necessários para aplicar a função.
- *numeroSubtipos*: é a quantidade de subtipos, ou seja, subclasses de uma dada classe;
- *numeroSubtiposNoMesmoPacote*: é a quantidade de subtipos que estão no mesmo pacote ou abaixo da hierarquia de pacotes da sua classe;
- *numeroModificadoresPublicos*: é a quantidade de métodos públicos das subclasses que são diferentes dos métodos públicos da superclasse;
- *visitor*: é uma medição para características do padrão *Visitor*. Esse padrão, quando implementado, comumente possui as características que o algoritmo busca;
- *numeroDeChamadasSubtipos*: é a quantidade de chamadas literais, *Pessoa.class* por exemplo, das subclasses de $SC$.

Sendo $p$ de acordo com a Listagem 3.1:

Listagem 3.1 – Registro que contém os parâmetros e pesos.

```
ParametrosEPesos {
  parametro1: real;
  parametro2: real;
  parametro3: real;
  parametro4: real;
  parametro5: real;
  peso1: real;
  peso2: real;
```

```
9    peso3: real;
10   peso4: real;
11   peso5: real;
12 }
```

Os valores dos pesos da Função 3.1 podem ser definidos utilizando algum método de decisão multicritério, como o Método de Análise Hierárquica (*Analytic Hierarchy Process*, AHP (SAATY, 1982, 1988)). Além disso, cada parâmetro e peso poderia ter um valor inicial sugerido para o desenvolvedor. Mas para isso é necessária a realização de um estudo mais amplo, com mais objetos de estudo e validação dos resultados com profissionais da área.

O valor de cada indício medido pode variar muito. Além disso, é necessário dar a possibilidade de ser definido um valor ideal para cada indício de acordo com sua peculiaridade. Assim, para que exista uma maior equivalência entre os valores de cada indício foram criadas as funções *normalizarRetilineo* e *normalizarReflexao*.

A função *normalizarRetilineo* recebe o valor do indício e um limiar para o qual o indício deve se aproximar para ser considerado ideal, conforme é apresentado na Função 3.2.

$$normalizarRetilineo(indicio, limiar) = indicio * (1/limiar) \tag{3.2}$$

Já a função *normalizarReflexao* utiliza a função *normalizarRetilineo* e caso o resultado ultrapasse o valor 1, é diminuído do resultado o valor excedente, como é apresentado através do Algoritmo 2.

**Algoritmo 2:** FUNÇÃO NORMALIZARREFLEXAO

**Entrada:** $indicio = \{\text{indício}\}$
$limiar = \{\text{limiar}\}$
**Saída:** $resultado$

```
1 início
2 |  resultado ← indicio * (1/limiar);      // <- normalizarRetilineo
3 |  se resultado > 1 então
4 |  |  resultado ← 1 − (resultado − 1);
5 |  fim
6 |  retorna resultado
7 fim
```

Por exemplo, o desenvolvedor define o valor ideal de certo indício para ser uma oportunidade de refatoração como *oito*. A Figura 3.3 apresenta o resultado das Funções *normalizarRetilineo* e *normalizarReflexao* com o valor de limiar definido anteriormente. Durante a execução da mecânica é aferido o valor *seis* para este indício. O resultado será *0,75*, independente da

função utilizada. Ou seja, está próximo de vir a ser uma oportunidade de refatoração.

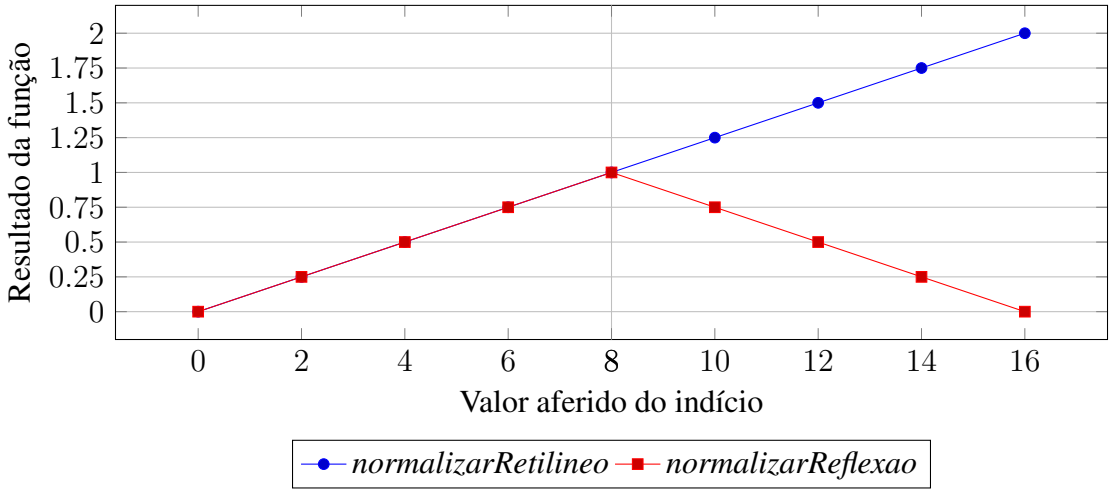


Figura 3.3 – Exemplo de resultado das Funções *normalizarRetilineo* e *normalizarReflexao*

Em outra medição, o valor do início aferido é *16*, por exemplo. Nesse caso, se a função utilizada foi a *normalizarRetilineo*, significa que o valor do indício é muito melhor que o definido pelo desenvolvedor. Em outras palavras, quanto maior o valor do indício, melhor a chance de ser uma oportunidade de refatoração. Entretanto, se foi utilizada a Função *normalizarReflexao*, o resultado será *zero*. Logo, significa que o valor do indício talvez esteja distante de ser uma oportunidade de refatoração aplicável. Assim, essa função pode ser usada quando um valor de indício muito alto signifique que o mesmo não é uma oportunidade válida. A Tabela 3.1 apresenta os parâmetros da Função Heurística 3.1 relacionados aos limiares das Funções *normalizarRetilineo* e *normalizarReflexao*.

A heurística apresentada através da Função 3.1 pode ser dividida em duas partes. Na primeira parte é utilizada a função *normalizarReflexao* para calcular o valor ideal do número de subtipos, e esse resultado é multiplicado pelo *Peso1*. O produto da multiplicação é então somado com o valor resultante da função *normalizarReflexao* aplicada ao número de subtipos no mesmo pacote da superclasse $SC$ multiplicado pelo *Peso2*. Essa primeira parte da função heurística define os critérios que contribuem positivamente para que uma dada hierarquia de classes possa ser considerada uma oportunidade de refatoração para Encapsular Classes com *Factory*.

Na segunda parte da Função Heurística 3.1 é aplicada a função *normalizarRetilineo* sob o valor do número de modificadores públicos, e após multiplicado pelo *Peso3*. Assim quanto mais modificadores públicos diferentes da superclasse, menor a chance de ser uma oportunidade de refatoração para Encapsular Classes com *Factory*. Em seguida, se aplica a função *norma-*

Tabela 3.1 – Relação dos parâmetros e limiares na Função Heurística 3.1

| Parâmetro | Limiar | Significado |
|---|---|---|
| *parametro1* | Limiar da Função *normalizarReflexao* | O valor do indício $numeroSubtipos$ que se aproximar ao valor do *parametro1* terá o resultado mais perto de *um*. |
| *parametro2* | Limiar da Função *normalizarReflexao* | O valor $numeroSubtiposNoMesmoPacote$ que se aproximar ao valor do *parametro2* terá o resultado mais perto de *um*. |
| *parametro3* | Limiar da Função *normalizarRetilineo* | O valor do indício $numeroModificadoresPublicos$ que se aproximar ao valor do *parametro3* terá o resultado mais perto de *um*, se o indício for maior, o resultado também será maior que *um*. |
| *parametro4* | Limiar da Função *normalizarRetilineo* | O valor do indício $visitor$ que se aproximar ao valor do *parametro4* terá o resultado mais perto de *um*, se o indício for maior, o resultado também será maior que *um*. |
| *parametro5* | Limiar da Função *normalizarRetilineo* | O valor do indício $numeroDeChamadasSubtipos$ que se aproximar ao valor do *parametro5* terá o resultado mais perto de *um*, se o indício for maior, o resultado também será maior que *um*. |

*lizarRetilineo* ao valor de $visitor$, multiplicada pelo *Peso4*. Da mesma maneira que acontece com o indício anterior, quanto maior o valor de $visitor$ menor a chance de ser uma oportunidade de refatoração. Por fim, é utilizada a função *normalizarRetilineo* sob o valor da variável $numeroDeChamadasSubtipos$, multiplicada pelo *Peso5*. Ou seja, quanto maior o número de chamadas diretas às subclasses, menor a chance de ser uma oportunidade de refatoração. Ao final das multiplicações, os três resultados são somados, para serem subtraídos do valor da primeira parte da função heurística, conforme é apresentado na Função 3.1.

Por fim, o conjunto de oportunidades de refatoração $Ss$ é ordenado em ordem decrescente em relação ao valor da aplicação da Função Heurística e exibido para que o usuário avalie quais oportunidades podem ser utilizadas no projeto de acordo com as suas necessidades, vantagens ou mesmo desvantagens da refatoração para padrão Encapsular Classes com *Factory*.

## 3.2 Mecânica de busca para Substituir Árvore Implícita por *Composite*

Um indício para a aplicação de Substituir Árvore Implícita por *Composite* pode ser encontrado em um trecho de código que possua uma árvore, descrita de maneira implícita. Uma árvore implícita é um trecho de código que utiliza uma estrutura lógica em árvore, entretanto tal estrutura não é representada como uma árvore. É possível verificar uma árvore implícita através

da Figura 3.4, onde é realizada a construção de um texto no formato XML.

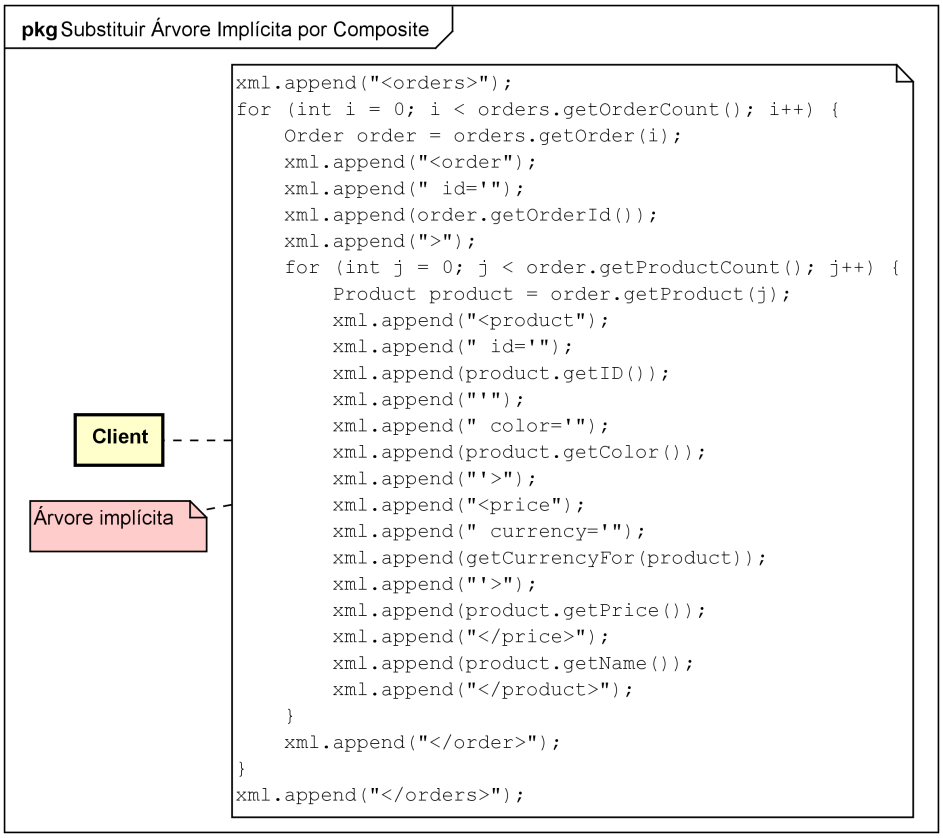


Figura 3.4 – Indício de uma oportunidade para a refatoração *Substituir Árvore Implícita por Composite*.

Uma solução para o problema de árvore implícita mostrado na Figura 2.6 é a refatoração para o padrão *Composite*. Assim, a responsabilidade da criação e montagem da árvore implícita é delegada para a Composição, desacoplando a lógica de construção do *Client*. Entretanto, a classe *Client* se torna dependente do *Composite*.

A alteração de uma árvore implícita para um *Composite* pode tornar o código mais enxuto e com um número menor de linhas. Dessa forma, reduz o acoplamento do *Client* com a construção da árvore, e utilizando no lugar uma estrutura de composição de nós para a construção da árvore. Com o uso do *Composite* é possível encapsular sentenças repetitivas para adição, remoção e formatação de nós. Além disso, a manipulação de novos nós é realizada de maneira generalizada, sem um grande acoplamento. Assim, a construção das árvores pelos *Clients* é realizada com a inserção de poucas linhas de código.

O Algoritmo 3 apresenta uma mecânica para a busca das oportunidades de refatoração para Substituir Árvore Implícita por *Composite*.

O algoritmo possui como entrada o conjunto de todos os $Metodos$ do projeto sendo

**Algoritmo 3:** MECÂNICA DE BUSCA PARA SUBSTITUIR ÁRVORE IMPLÍCITA POR *Composite*

**Entrada:** $Metodos =$ {conjunto de todos os métodos do projeto}
$TAGsDoProjeto =$ {conjunto de todas as *tags* do projeto com suas respectivas quantidades}
**Saída:** $Ss =$ {candidatos a oportunidades de refatoração}

```
1  início
2      para cada Metodo dos Metodos faça
3          TAGs ← obtém todas as tags presentes no Metodo;
4          quantidadeDeTags ← total de TAGs identificadas dentro do Metodo;
5          se quantidadeDeTags > 1 então
6              quantidadeDeTagsDeFechamento ← quantidade de TAGs de
                fechamento; (</p> por exemplo)
7              se quantidadeDeTagsDeFechamento > 1 então
8                  TAGsDoProjeto + ← TAGs; (o quantitativo de cada tag é
                    adicionado no contador de tags do projeto)
9                  resultado ← aplica a Função Heurística 3.3 para Substituir Árvore
                    Implícita por Composite;
10                 Ss + ← Metodo, TAGs e resultado; (adiciona o Metodo, as
                    TAGs do método e o resultado ao conjunto de candidatos a
                    oportunidades de refatoração Ss)
11             fim
12         fim
13     fim
14     para cada oportunidade S do conjunto de oportunidades Ss faça
15         para cada TAG da oportunidade S faça
16             somatorioDosTotais + ← obtém a quantidade total da TAG do
                conjunto das TAGsDoProjeto e adiciona no somatório;
17         fim
18         resultadoSomatorio ← aplica a Função Heurística 3.4 para Substituir
            Árvore Implícita por Composite;
19         S ← resultado de S + resultadoSomatorio;
20     fim
21     retorna Ss
22 fim
```

investigado. Além disso, é recebido um conjunto que conterá todas as *tags* do projeto com suas respectivas quantidades. Nesse momento, o conjunto $TAGsDoProjeto$ é vazio. Sua principal função é manter a contagem de cada *tag*, para que ao final seja calculado o somatório das quantidades de cada *tag* pertencente ao método sendo avaliado. Tal algoritmo tem como saída o conjunto de oportunidades de refatoração.

Uma característica do padrão de projeto *Composite* na refatoração para substituir uma árvore implícita escrita com *tags* em XML é que cada *tag* encontrada pode valer para todo o projeto. Assim, uma *tag* que aparece recorrentemente no projeto tem maior potencial para ser refatorada para esse padrão. Podem ocorrer casos em que um método possua uma quantidade maior de *tags*, entretanto, tais *tags* são pouco utilizadas. Já outro método possui uma quantidade menor de *tags*, onde essas *tags* são mais recorrentes. Assim, o segundo método provavelmente possuirá um valor de resultado maior que o primeiro método.

Essa contagem de cada *tag* pode ser mais vantajosa onde a implementação da composição de cada nó seja realizada separadamente de outros nós. Assim, é importante destacar qual *tag* é mais utilizada no decorrer do projeto, para que essa refatoração que visa a substituição dessa *tag* possua uma maior aplicabilidade.

Seguindo o Algoritmo 3, o primeiro passo é iterar todos os métodos do projeto. Para cada método, são verificadas as *tags* escritas em Strings literais (`doc.append("<b>");` por exemplo) e adicionadas em uma lista de $TAGs$ do método. Após, se obtém o total de *tags* identificadas no interior do $Metodo$ sendo verificado. O algoritmo continuará a verificar somente se houver pelo menos mais de uma $TAG$ presente no método.

O próximo passo é contar a quantidade de *tags* de fechamento (`</p>`, por exemplo) presentes no método. É comum em XML, ou mesmo em linguagens derivadas do XML, a presença de *tags* de fechamento. A presença de tais *tags* pode ser um indicativo que há uma presença de árvore implícita dentro do método sendo analisado. Além disso, evita a presença de falsos positivos no conjunto de candidatos a oportunidades de refatoração. Caso o método possua ao menos duas *tags* de fechamento, o Algoritmo 3 continua sua execução.

Dada a presença de mais de uma *tag* de fechamento, as quantidades de cada $TAG$ do $Metodo$ são adicionadas às $TAGsDoProjeto$, para que ao final seja possível saber o total de cada *tag* encontrada. Em seguida, é aplicada a Função Heurística 3.3, onde os valores dos pesos e dos parâmetros são informados anteriormente à execução do algoritmo pelo usuário.

$$resultado(m : Metodo, p : ParametrosEPesos) =$$
$$(normalizarRetilineo(quantidadeDeTags(m), p.parametro1) * p.peso1 +$$
$$normalizarRetilineo(quantidadeDeTagsDeFechamento(m), p.parametro2) * p.peso2) \quad (3.3)$$

Onde:

- *m*: é o método que se está buscando por *tags*;
- *p*: contém os parâmetros e pesos necessários para aplicar a função.
- *quantidadeDeTags*: a quantidade de *tags* encontrada dentro do método sendo verificado;
- *quantidadeDeTagsDeFechamento*: é a quantidade de *tags* de fechamento presentes no método.

A Função 3.3 vai priorizar as oportunidades de refatoração de acordo com a quantidade de *tags* presentes no método. A presença de poucas *tags* pode implicar em falsos positivos, não sendo interessante dessa forma um ordenamento mais complexo.

Continuando a execução do Algoritmo 3, o $Metodo$, suas $TAGs$ e o resultado obtido da Função Heurística 3.3 são adicionados ao conjunto de oportunidades de refatoração para Substituir Árvore Implícita por *Composite*.

Após todos os métodos serem verificados, são iteradas as oportunidades de refatoração $Ss$, e para cada $TAG$ da oportunidade de refatoração $S$ é obtida a quantidade total dessa *tag* no projeto de software. Ou seja, caso o método da oportunidade $S$ possua as *tags* `<p>` e `<b>`, onde `<p>` é encontrado em 3 ocasiões em todo o projeto, e `<b>` em 4 casos, a variável $somatorioDosTotais$ possuirá o valor 7 (3+4).

Subsequente, é aplicada a Função Heurística 3.4:

$$resultadoSomatorio(t : TAG, p : ParametrosEPesos) =$$
$$(normalizarRetilineo(somatorioDosTotais(t), p.parametro3) * p.peso3) \quad (3.4)$$

Onde:

- *t*: é uma *tag* pertencente a uma oportunidade de refatoração;

- *p*: é um registro que possui os pesos e parâmetros para a aplicação da função;
- *somatorioDosTotais*: é o total de vezes que cada $TAG$ do método é encontrada em todo o projeto.

A Tabela 3.2 apresenta os parâmetros das Funções Heurísticas 3.3 e 3.4 relacionado aos limiares da Função *normalizarRetilineo*. Como em todos os casos é utilizada a Função *normalizarRetilineo*, todos os resultados podem ultrapassar o valor *um*, denotando que quanto maior o valor do indício a partir do parâmetro respectivo, melhor.

Tabela 3.2 – Relação dos parâmetros e limiares nas Funções Heurísticas 3.3 e 3.4.

| **Parâmetro** | **Limiar** | **Significado** |
|---|---|---|
| *parametro1* | Limiar da Função *normalizarRetilineo* | O valor do indício $quantidadeDeTags$ que se aproximar ao valor do *Parâmetro1* terá o resultado mais perto de *um*, se o indício for maior, o resultado também será maior que *um*. |
| *parametro2* | Limiar da Função *normalizarRetilineo* | O resultado será *um* quando o valor do indício $quantidadeDeTagsDeFechamento$ for igual ao valor do limiar que é fornecido pelo *Parâmetro2* |
| *parametro3* | Limiar da Função *normalizarRetilineo* | O valor $somatorioDosTotais$ próximo ao valor informado pelo *Parâmetro3* terá um valor resultante próximo a *um*. |

Assim como a Heurística 3.3, quanto maior a presença de *tags* que foram encontradas várias vezes dentro do projeto, maior a chance do método ser uma oportunidade de refatoração aplicável.

Por fim, o $resultado$ obtido anteriormente pela Função 3.3 é somado ao valor do *resultadoSomatorio*, e adicionado à oportunidade de refatoração correspondente, conforme é apresentado no Algoritmo 3.

Para que seja possível a execução da mecânica com valores iniciais para os parâmetros e pesos é necessário realizar um amplo estudo que possa determinar, ou mesmo sugerir, tais valores. Este estudo não faz parte do escopo deste trabalho, mas convém ser realizado em um trabalho futuro. Em relação aos pesos, o desenvolvedor pode utilizar algum método de decisão multicritério (como o AHP) caso deseje.

## 3.3 Mecânica de busca para Substituir Envio Condicional por *Command*

Locais no código-fonte com várias sentenças condicionais e várias linhas de código podem ser refatoradas utilizando refatorações como *Extrair Método* (FOWLER et al., 1999).

Entretanto, se essas sentenças condicionais estejam realizando um roteamento para a execução de ações, a refatoração Substituir Envio Condicional por *Command* pode ser utilizada. A Figura 3.5 apresenta essa situação, onde a classe *CatalogApp* possui uma extensa lista de sentenças condicionais que realizam o roteamento de requisições para a execução de ações.

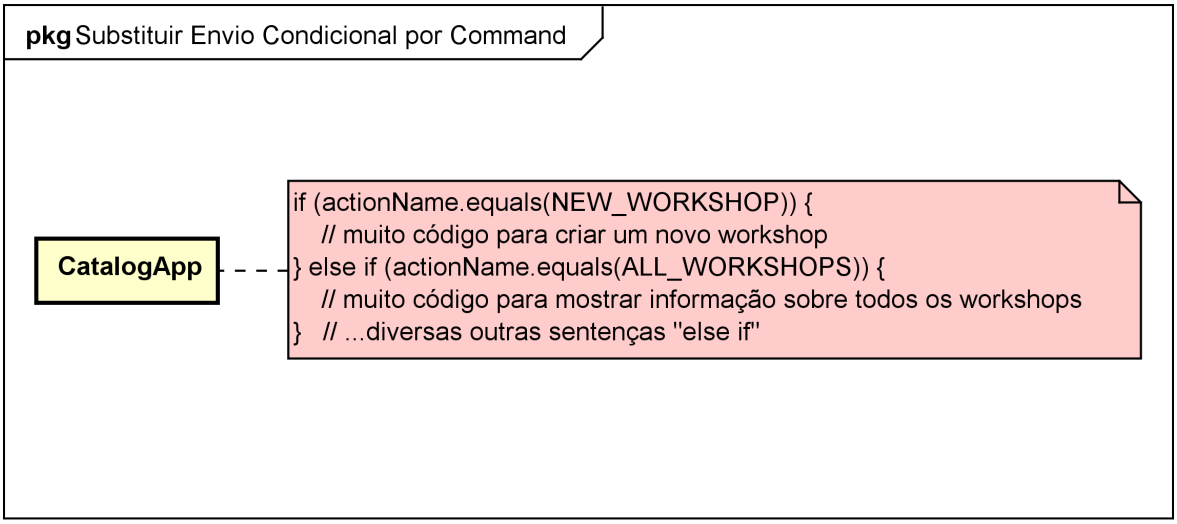


Figura 3.5 – Indícios de uma oportunidade para a refatoração *Substituir Envio Condicional por Command*.

Com a aplicação da refatoração Substituir Envio Condicional por *Command* terão sido criadas as classes *Handler*, que representa um comando genérico; *NewWorkshopHandler* e *AllWorkshopHandler*, entre outras para cada condicional, que representam cada comando concreto a ser executado. Na *CatalogApp*, é definido um *Map* que armazena os comandos concretos com o seu respectivo identificador. A partir desse momento, as requisições são enviadas para o *Map handlers*, que chama a execução da ação respectiva. Dessa forma, as ações ficam desacopladas da classe *CatalogApp*, que podem ser utilizadas por outras classes, enfileiradas para a execução, ou mesmo serem compostas para a construção de comandos mais complexos.

O Algoritmo 4 apresenta uma mecânica para a busca das oportunidades de refatoração para Substituir Envio Condicional por *Command*. Segundo o algoritmo, são obtidos todos os blocos condicionais `if` do código-fonte e inseridos na lista $IFs$.

A lista de bloco condicionais $IFs$ é, então, iterada. É contada a quantidade de sentenças condicionais que o bloco condicional $IF$ possui. Uma sentença condicional pode ser um `if (condição) ...`, `else if (condição) ...`, ou `else ...`. Blocos condicionais internos às sentenças condicionais não são contados.

É verificada então se a quantidade de condicionais é maior ou igual a três. Blocos con-

**Algoritmo 4:** MECÂNICA DE BUSCA PARA SUBSTITUIR ENVIO CONDICIONAL POR *Command*

**Entrada:** $IFs = \{$conjunto das blocos condicionais `if`$\}$
**Saída:** $Ss = \{$candidatos a oportunidades de refatoração$\}$

```
1  início
2      para cada bloco condicional IF do conjunto IFs faça
3          quantidadeCondicionais ← quantidade de if, else if's e else
             contido no bloco condicional IF;
4          se quantidadeCondicionais >= 3 então
5              quantidadeLoC ← quantidade de linhas de código (LoC) presente no
                 IF;
6              mediaLoCPorCondicao ← quantidadeLoC ÷
                 quantidadeCondicionais; (média de LoC por condição no IF)
7              se mediaLoCPorCondicao >= 3 então
8                  quantidadeExpressoesSimples ← quantidade de expressões
                     simples no bloco IF;
9                  mediaCondicoesSimples ← quantidadeExpressoesSimples ÷
                     quantidadeCondicionais; (média de condições simples por IF)
10                 resultado ← aplica a Função Heurística 3.5 para Substituir Envio
                     Condicional por Command;
11                 Ss + ← IF e resultado; (adiciona o bloco condicional IF e o
                     resultado ao conjunto de candidatos a oportunidades de
                     refatoração Ss)
12             fim
13         fim
14     fim
15     retorna Ss
16 fim
```

dicionais com poucas condições não caracterizam uma oportunidade para a refatoração atual. Caso essa condição seja satisfeita, é contada a quantidade de linhas de código de todo o bloco condicional. A forma de contagem de linhas de código pode ser vista através do Anexo B.

Em seguida, é calculada a média de linhas de código por condição. Essa aferição auxilia no filtro de blocos condicionais que possuem, para cada condição, uma a duas linhas de código. Afinal, a refatoração Substituir Envio Condicional por *Command* é aplicada em blocos de código inchados, tornando cada trecho de código dentro de cada condição uma ação diferente. Caso o bloco condicional possua uma média de linhas de código por condição inferior a três, esse bloco é desconsiderado.

Ainda, o Algoritmo 4 apresenta a contagem da quantidade de expressões simples do bloco $IF$. As expressões booleanas de cada sentença condicional são verificadas e são contadas caso sejam considerada simples. Uma expressão pode ser considerada simples quando for uma

variável primitiva, uma invocação de um método com apenas um parâmetro, uma verificação de instância (`instanceof`), uma expressão com prefixo (`!variavel`), e uma expressão de comparação (==, >=, <=, etc.). Caso for uma invocação de método, é verificado se o argumento é uma variável de um tipo primitivo (`char`, `int`, `long`, `double`, etc.), `String` ou `Class`. O mesmo é feito para ambos os lados da expressão de comparação e do conteúdo da expressão com prefixo. Satisfeitas essas verificações, a expressão condicional é então contada.

Blocos condicionais que realizam roteamento de ações possuem uma ou poucas verificações condicionais dentro de uma expressão de uma sentença condicional. Verificações complexas em sentenças condicionais podem indicar que não são oportunidades de refatoração para Substituir Envio Condicional por *Command*.

Após é realizado o cálculo da média de condições simples presentes no bloco condicional $IF$. Assim, quanto maior a média de sentenças condicionais simples, maior a chance de ser uma oportunidade de refatoração.

Dados esses indícios, é aplicada a Função Heurística 3.5, onde os valores de parâmetros e pesos são informados pelo usuário.

$$
\begin{aligned}
&resultado(bc : BlocoCondicional, p : ParametrosEPesos) = \\
&\quad \big(normalizarRetilineo(quantidadeCondicionais(bc), p.parametro1) * p.peso1 \\
&\qquad + normalizarRetilineo(quantidadeLoC(bc), p.parametro2) * p.peso2 \\
&\quad + normalizarRetilineo(mediaLoCPorCondicao(bc), p.parametro3) * p.peso3 \\
&+ normalizarRetilineo(quantidadeExpressoesSimples(bc), p.parametro4) * p.peso4 \\
&+ normalizarRetilineo(mediaCondicoesSimples(bc), p.parametro5) * p.peso5\big) \quad (3.5)
\end{aligned}
$$

Onde:

- *bc*: é o bloco condicional sendo verificado;
- *p*: é um registro com os pesos e parâmetros que serão utilizados na aplicação da função;
- *quantidadeCondicionais*: é a quantidade de `if`, `else if`s e `else` de um bloco condicional $IF$;
- *quantidadeLoC*: é o número de linhas de código do bloco $IF$;
- *mediaLoCPorCondicao*: é a média de linhas de código por sentença condicional;

- *quantidadeExpressoesSimples*: é a quantidade de expressões simples das sentenças condicionais;
- *mediaCondicoesSimples*: é a média de condições simples por sentenças condicionais do bloco $IF$.

Assim como as funções anteriores, a determinação de valores iniciais para pesos e parâmetros dependem de um estudo mais amplo para a aferição de cada variável. Especificamente quanto aos pesos, eles podem ser calculados pelo desenvolvedor com o auxílio de algum método de decisão multicritério.

A Função 3.5 apresenta o cálculo para a presente refatoração. De forma geral, quanto maior o valor de cada indício, maior a chance da oportunidade candidata ser uma oportunidade de refatoração. Sendo assim, foi utilizada a função *normalizarRetilineo* em cada indício com seu respectivo valor de parâmetro. Todos os resultados da aplicação dos indícios na função *normalizarRetilineo* são multiplicados pelos seus respectivos pesos, definindo assim a relevância entre cada indício. A Tabela 3.3 apresenta os parâmetros das Função Heurística 3.5 relacionado ao limiar de cada Função *normalizarRetilineo* utilizada.

Tabela 3.3 – Relação dos parâmetros e limiares nas Funções Heurísticas 3.5.

| **Parâmetro** | **Limiar** | **Significado** |
|---|---|---|
| *parametro1* | Limiar da Função *normalizarRetilineo* | Caso a $quatidadeCondicionais$ seja igual ao valor do *parametro1* o resultado será *um*, superior caso a $quantidadeCondicionais$ for maior, e menor caso contrário. |
| *parametro2* | Limiar da Função *normalizarRetilineo* | O resultado será *um* quando o valor do indício $quantidadeLoC$ for igual ao valor do *parametro2* |
| *parametro3* | Limiar da Função *normalizarRetilineo* | O valor $mediaLoCPorCondicao$ próximo ao valor informado pelo *parametro3* terá um valor resultante próximo a *um*. |
| *parametro4* | Limiar da Função *normalizarRetilineo* | O valor $quantidadeExpressoesSimples$ igual ao valor informado pelo *parametro4* terá o resultado *um*. |
| *parametro5* | Limiar da Função *normalizarRetilineo* | O valor $mediaCondicoesSimples$ aproximado ao valor informado pelo *parametro3* terá um valor resultante próximo a *um*. |

Por fim, o $IF$ e o resultado da função heurística são adicionados ao conjunto de candidatos a oportunidades de refatoração para Substituir Envio Condicional por *Command*. O conjunto $Ss$ é classificado em ordem decrescente e exibido para o usuário, que avaliará quais oportunidades podem ser refatoradas de acordo com o contexto do projeto, vantagens e desvantagens dessa refatoração.

## 3.4 Fechamento do Capítulo

Este capítulo apresentou a definição de três mecânicas de busca para padrões de projeto. Foi descrito um processo de busca de oportunidades de refatoração. Foram propostas as mecânicas de busca para Encapsular Classes com *Factory*, Substituir Árvore Implícita por *Composite* e Substituir Envio Condicional por *Command*. Para cada mecânica foram identificados e apresentados seus indícios para serem consideradas oportunidades, e em seguida o conjunto de passos para a busca desses indícios. Cada mecânica possui uma Função Heurística que permite a ordenação das oportunidades identificadas para que o desenvolvedor possa aplicar as refatorações nas oportunidades de mais interesse. Para tornar os valores dos indícios mais homogêneos, foram criadas as funções *normalizarRetilineo* e *normalizarReflexao*. O próximo capítulo trata de um estudo de caso utilizando as mecânicas de busca propostas, além da implementação do *plug-in Pattern Refactoring*.

# 4 ESTUDO DE CASO

A fim de verificar as mecânica propostas neste trabalho foram analisados quatro projetos de código aberto de diferentes números de classes e de linhas de código (LoC). A Tabela 4.1 apresenta os projetos com seus respectivos links para a página do projeto. Foram selecionados projetos que atendessem a áreas de aplicação diversas, o que auxilia numa maior cobertura da avaliação das mecânicas, pois oferecem uma maior variedade de estilos de codificação, estruturas de código e de limitações. Dessa maneira, as mecânicas podem ser aferidas sob diferentes condições e verificada a eficácia de cada mecânica.

Tabela 4.1 – Projetos utilizados no Estudo de Caso.

| Projeto | Descrição | Versão | Nº Classes | Nº LoC |
|---|---|---|---|---|
| **ADempiere** `www.adempiere.net` | Suite Empresarial ERP & CRM | 3.9.0 | 3705 classes | 622132 LoC |
| **Apache-Ant** `ant.apache.org` | Construtor de aplicações Java | 1.10.6 | 801 classes | 83593 LoC |
| **EclipseLink** `www.eclipse.org/eclipselink` | Biblioteca de Persistência | 2.6 | 3013 classes | 403844 LoC |
| ***Pattern Refactoring*** `bitbucket.org/ThiagoKrug/patternrefactoring/` | *Plug-in* de busca por oportunidades de refatoração para padrões | 1.0 | 258 classes | 12112 LoC |

O projeto ADempiere é um software ERP (*Enterprise Resource Planning*), com suporte a CRM (*Customer Relationship Management*), análise de desempenho financeira e gestão da cadeia de suprimentos. É um software web e multi-plataforma, com relatórios e interface de usuário inteligentes.

O Apache-Ant é uma ferramenta para a construção de aplicações, que pode ser utilizado para qualquer tarefa que possua alvos e tarefas. É um sistema muito flexível, que possibilita a construção de programas e gerenciamento de dependências.

O EclipseLink é uma biblioteca de persistência que possibilita a comunicação de um sistema com banco de dados com um alto nível de abstração. Implementa os padrões JPA (*Java Persistence API*), SDO (*Service Data Objects*) e JAXB (*Java Architecture for XML Binding*).

O *plug-in Pattern Refactoring* auxilia na busca de oportunidades de refatoração para padrões de projeto. Ele utiliza o *framework* AOPJungle para a extração das informações do código-fonte e em seguida retorna as oportunidades candidatas ordenadas pelo valor da respectiva função heurística.

A realização das buscas de oportunidades de refatoração se deu através da implementação do *plug-in Pattern Refactoring*, que contém os algoritmos e funções de cada mecânica

proposta. O *Pattern Refactoring* localiza os indícios e calcula os valores baseado nas informações do código-fonte extraídas pelo *plug-in* AOPJungle.

Todos os quatro projetos foram submetidos a cada mecânica de busca por oportunidades de refatoração. As três primeiras oportunidades candidatas de cada projeto foram descritas atentando aos pontos positivos e negativos de sua aplicação. É apresentada a primeira oportunidade candidata de cada projeto de forma mais detalhada, com diagramas e código-fonte quando necessário.

Cada candidata a oportunidade de refatoração possui um valor do resultado da respectiva função heurística da mecânica. Esse valor pode ser tanto positivo quanto negativo. Em seguida, os autor deste trabalho verificou a aplicabilidade da refatoração sobre a oportunidade candidata, classificando-a de "muito alta", com uma alta probabilidade de ser uma oportunidade de refatoração, até "muito baixa", sendo um possível falso positivo.

Ao final dos resultados de cada mecânica é feita uma avaliação das oportunidades de refatoração obtidas dos projetos. É feita uma sintetização das características encontradas das oportunidades candidatas com recomendações do que as mecânicas poderiam melhorar, assim como do que funcionou para a detecção.

## 4.1 Resultados para a mecânica de busca para Encapsular Classes com *Factory*

Para buscar por oportunidades de refatoração para Encapsular Classes com *Factory* é necessário informar valores de parâmetros e de pesos para a Função Heurística 3.1 apresentada no Capítulo 3. Para a execução de três projetos foram utilizados os valores de parâmetros e pesos apresentados pela Função 4.1. Para o projeto Apache-Ant se utilizou o valor três para o parâmetro do indício *numeroModificadoresPublicos*.

$$
\begin{aligned}
resultado = \big( & normalizarReflexao(numeroSubtipos, 7) * 1 \\
& + normalizarReflexao(numeroSubtiposNoMesmoPacote, 7) * 1 \big) \\
& - \\
\big( & normalizarRetilineo(numeroModificadoresPublicos, 6) * 0.7 \\
& + normalizarRetilineo(visitor, 3) * 2 \\
& + normalizarRetilineo(numeroDeChamadasSubtipos, 2) * 1 \big) \quad (4.1)
\end{aligned}
$$

Os valores dos parâmetros foram definidos de acordo com as execuções realizadas em cada projeto. Eram verificadas as primeiras oportunidades de refatoração resultantes da mecânica com a função heurística na medida em que cada teste era executado. A cada execução, as oportunidades candidatas eram avaliadas pelo autor deste trabalho e redefinidos os valores dos parâmetros e pesos para priorizar as oportunidades que eram mais relevantes e aplicáveis. Dessa forma, chegou-se nos valores de pesos e parâmetros apresentados pela Função 4.1 e tabulados como a Tabela 4.2 apresenta. Vale ressaltar que para o projeto Apache-Ant, o valor do parâmetro para o indício *numeroModificadoresPublicos* foi três, pois trouxe resultados melhores que com os valores anteriores.

Tabela 4.2 – Valores dos parâmetros dos indícios.

| **Indício** | **Valor do parâmetro** | **Significado** |
|---|---|---|
| Número de subtipos | 7 | Privilegia as oportunidades com uma quantidade de subclasses próxima a 7. |
| Número de subtipos no mesmo pacote | 7 | Privilegia as oportunidades que possuem cerca de 7 subtipos no mesmo pacote. |
| Número de modificadores públicos | 6 | As oportunidades são desprestigiadas a partir de 6 modificadores públicos |
| Visitor | 3 | A partir de 3 vezes identificadas as palavras-chave para o padrão *Visitor*, a oportunidade é desprestigiada |
| Número de chamadas literais à subtipos | 2 | É desfavorecida a oportunidade que contiver a partir de duas chamadas literais. |

Os valores dos pesos indicam o grau de importância do resultado daquele indício para o valor final da função. Quanto maior o valor do peso, maior será a relevância do indício. Recomenda-se utilizar valor *um* para indicar que o valor do indício é relevante. Mas isso pode ser personalizado pelo desenvolvedor, caso ele queira que o valor de um indício seja *dez* vezes mais importante que outro, ele pode usar o valor *dez* e peso *um* para o menos relevante.

Poderia haver uma definição de valores "padrão" para os pesos e parâmetros, para que o desenvolvedor apenas executasse a mecânica. Entretanto, para criar uma definição para cada valor é necessária a realização de mais estudos e testes que extraiam características de cada indício que possam ser consideradas valores "padrão". Do mesmo modo é preciso mais estudos para a definição de valores dos limiares que possam determinar a aplicabilidade ou não de uma refatoração em uma oportunidade candidata. Já para os pesos uma abordagem utilizando um método de decisão multicritério para as suas valorações pode ser utilizada pelo desenvolvedor caso ele sinta necessidade.

O parâmetro do número de subtipos de uma superclasse foi definido como *sete* pois durante as execuções se verificou que hierarquia de classes que possuem por volta desse valor podem ser caracterizadas como uma oportunidade de refatoração, como a Tabela 4.2 apresenta. Um número abaixo desse pode não valer a pena aplicar a refatoração, e acima dele pode caracterizar uma hierarquia de classes em que os subtipos exerçam uma função de *State* (GAMMA et al., 2000) por exemplo. Para a execução em outros projetos, pode-se utilizar valores em um intervalo de *cinco* a *14*. A relevância do resultado da função *normalizarReflexao* sobre o número de subtipos com esse parâmetro foi considerado importante, assim, recebeu o valor de peso um.

Da mesma forma que o indício anterior, o número de subtipos no mesmo pacote recebeu o valor *sete* como parâmetro. Ou seja, quanto mais próximo de *sete* subtipos da superclasse que estão na mesma hierarquia de pacotes, maior a possibilidade de ser uma oportunidade de refatoração. Ambos indícios possuem parâmetros similares pois, caso uma superclasse tenha *n* subtipos, é interessante que esses *n* subtipos estejam na mesma hierarquia de pacotes para caracterizar uma oportunidade de refatoração. E, de maneira semelhante ao indício anterior, podem ser utilizados valores no intervalo de *cinco* a *14* para o parâmetro. Como foi considerado relevante as subclasses estarem no mesmo pacote o peso desse indício recebeu o valor *um*.

Na segunda parte da função heurística estão os indícios que contribuem negativamente para a oportunidade candidata ser uma oportunidade de refatoração real. Quanto maior for a quantidade de métodos que são públicos das subclasses, menor a chance dessa hierarquia poder ser refatorada. Assim, foi dado o valor *seis* para o parâmetro do número de modificadores públicos, pois podem existir métodos públicos que ocultados não prejudicam as demais classes do projeto. No caso do projeto Apache-Ant, verificou-se que os métodos públicos prejudicavam os resultados. Foi então colocado o valor *três* para o parâmetro, que trouxe resultados melhores que os anteriores, como apresenta a Tabela 4.2. Um valor do intervalo de *um* a *dez* pode ser utilizado para o valor desse parâmetro. O peso de sua relevância para o cálculo não é tão grande, recebendo o valor *0,7*.

Quanto ao indício *visitor*, recebe o valor *três* como parâmetro pois caso exista algumas instâncias das palavras *"Visitor"* e *"visit"* é provável que seja a implementação do padrão *Visitor* (GAMMA et al., 2000). O intervalo recomendado para esse parâmetro é de *três* a *nove*. O peso para esse indício foi considerado muito alto, para que quando encontradas as classes pertencentes ao padrão, elas não sejam consideradas oportunidades de refatoração. Assim, o

valor do peso inserido foi *dois*.

Por fim, o número de chamadas literais recebeu *dois* como parâmetro da função, onde quanto mais chamadas literais mais significa que essa classe deve ser mantida como pública. O intervalo para o valor do parâmetro pode ser de *um* a *cinco*. O peso dessa avaliação recebeu valor *um* pois é um indício importante de ser aferido.

A seguir estão descritos os resultados das execuções da mecânica para a busca de oportunidades de refatoração para Encapsular Classes com *Factory* aplicada aos projetos apresentados anteriormente, utilizando a Função 4.1. Lembrando que o valor do indício *numeroModificadoresPublicos* foi *três* para o projeto Apache-Ant.

### 4.1.1 ADempiere

A Tabela 4.3 apresenta os três principais resultados da execução da mecânica de busca por oportunidades de refatoração para Encapsular Classes com *Factory*, de um total de *38* oportunidades candidatas. Duas oportunidades foram consideradas com "alta" aplicabilidade, enquanto a terceira foi considerada de aplicabilidade "média".

Tabela 4.3 – Três primeiras candidatas a oportunidades de refatoração para Encapsular Classes com *Factory* do projeto ADempiere.

| # | **Superclasse** $SC$ | **Resultado** | **Aplicabilidade** |
|---|---|---|---|
| 1 | PopupAction | 0,9095238 | Alta |
| 2 | AbstractExcelExporter | 0,7404761 | Alta |
| 3 | Info | 0,6761904 | Média |

A Figura 4.1 apresenta a condição da família de classes da primeira candidata *PopupAction* antes da aplicação da refatoração Encapsular Classes com *Factory*. A superclasse *PopupAction* possui seis subclasses, entretanto, quatro delas estão localizadas em um pacote diferente da superclasse. Além disso, as subclasses possuem apenas dois métodos públicos diferentes da interface de comunicação da superclasse *PopupAction*.

Há algumas características que tornam essas classes uma oportunidade considerada "alta" para a aplicação da refatoração: há apenas dois métodos diferentes da superclasse, não possuem chamadas literais e não implementarem o padrão de projeto *Visitor*. Entretanto, quatro subclasses estão localizadas fora do pacote da superclasse.

A Figura 4.2 apresenta o resultado da aplicação da refatoração Encapsular Classes com *Factory* na família de classes de *PopupAction*. Todas as subclasses foram movidas para o pacote da superclasse e protegidas de acessos externos. Além disso, seus construtores públicos foram

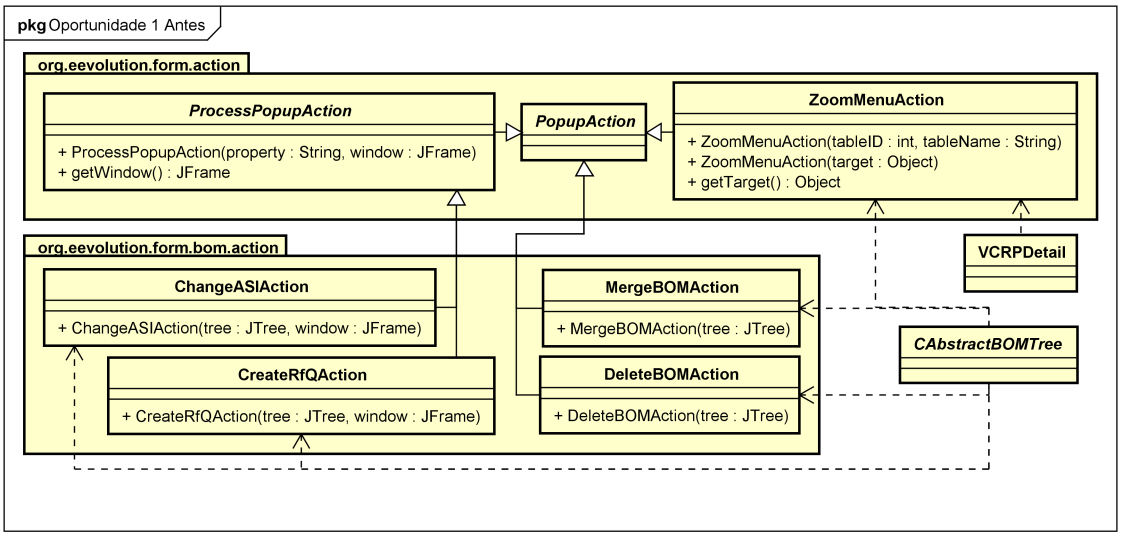


Figura 4.1 – Oportunidade candidata 1 antes da refatoração.

modificados, tornando-os protegidos. Em seguida, foi adicionado um método estático para a instanciação de cada subclasse na classe *PopupAction*. Através desses métodos que agora as classes clientes *CAbstractBOMTree* e *VCRPDetail* realizam as chamadas, utilizando assim a mesma interface de comunicação. Os métodos *getWindow* e *getTarget* foram protegidos e não ocasionaram problemas na compilação do projeto. Enfim, os atributos públicos de classe (*public static final*) das classes foram externados na superclasse para que pudessem ser acessados.

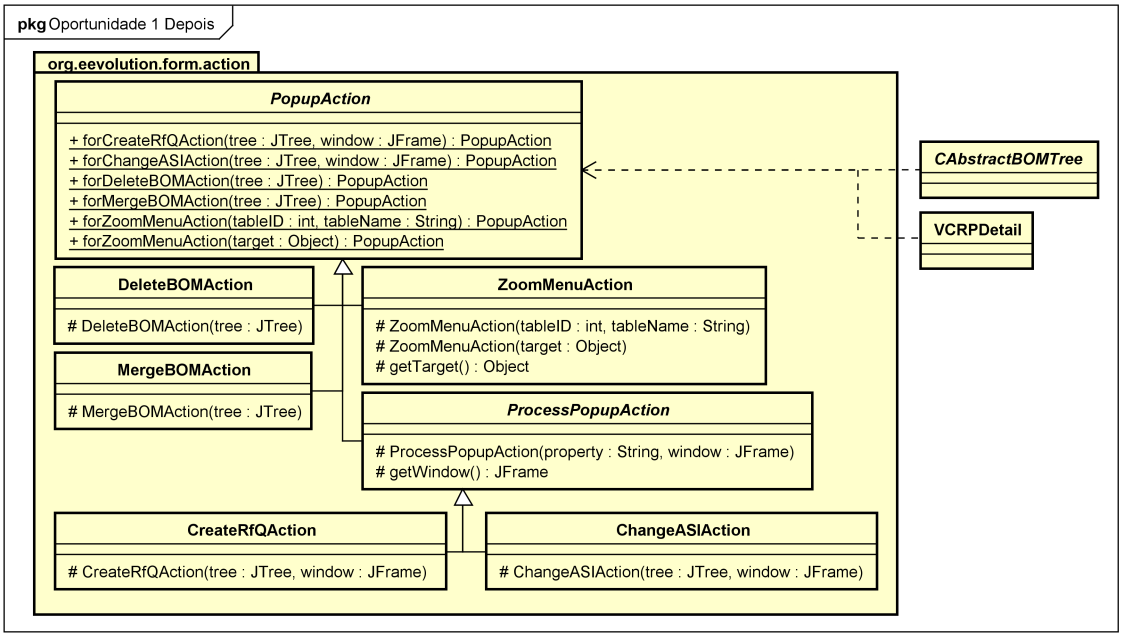


Figura 4.2 – Oportunidade candidata 1 depois da refatoração.

A segunda candidata a oportunidade de refatoração é a hierarquia de classes de *Abs-*

*tractExcelExporter*, que possui quatro subclasses, sendo duas delas no mesmo pacote que a superclasse. Há também o método público *getCtx* da subclasse *ArrayExcelExporter* que difere da interface de comunicação pública da superclasse. Além disso, não existem chamadas diretas às subclasses. Com essas características, essa hierarquia de classe obteve uma classificação "alta".

Após a aplicação da refatoração, todas as classes ficaram na mesma pasta da superclasse *AbstractExcelExporter*, com os seus construtores protegidos. O método *getCtx* já existia na superclasse, entretanto o mesmo era protegido. Para manter a mesma interface, se optou por modificar para público o método na superclasse.

Por fim, a terceira candidata a refatoração Encapsular Classes com *Factory* foi a família de classes de *Info*, recebendo o resultado *0,6761904*. Essa família é composta por dez classes e todas estão no mesmo pacote da superclasse, possuem apenas quatro métodos públicos diferentes da superclasse, não são chamadas literalmente nem implementam o padrão *Visitor*. Entretanto, a subclasse *InfoProduct* necessita ser pública pois é referenciada em uma expressão `instanceof`. Além disso, seis das dez subclasses já possuem os seus métodos construtores protegidos. Isso acontece pois a superclasse *Info* tem o método *create* que é encarregado de criar as instâncias das subclasses de acordo com o valor de uma *String*. Dessa maneira, essa oportunidade foi classificada como "média".

A refatoração foi aplicada nesse conjunto de classes, tornando assim todas as subclasses protegidas no pacote, com exceção da classe *InfoProduct*. O método público *getM_AttributeSet_ID* foi tornado protegido, pois era usado apenas pela classe *InfoPAttribute*, que faz parte do mesmo pacote das classes. O método da superclasse *saveSelectionDetail* foi tornado público, pois duas subclasses implementavam esse método de forma pública. Por fim, o método *stateChanged* da classe *InfoProduct* permaneceu público e sem implementação na superclasse, pois é uma implementação de uma interface que apenas a subclasse usa.

### 4.1.2 Apache-Ant

Durante a execução dos testes para o projeto Apache-Ant, verificou-se que o valor do indício *numeroModificadoresPublicos* com melhores resultados foi *três*. Ao final, foram identificadas *21* candidatas a oportunidades de refatoração para Encapsular Classes com *Factory*. A Tabela 4.4 apresenta os três melhores resultados, onde a aplicabilidade da primeira oportunidade foi considerada "média", da segunda "alta", e da terceira "muito baixa".

Tabela 4.4 – Três primeiras candidatas a oportunidades de refatoração para Encapsular Classes com *Factory* do projeto Apache-Ant.

| # | **Superclasse** $SC$ | **Resultado** | **Aplicabilidade** |
|---|---|---|---|
| 1 | AntClassLoader | 0,8571428 | Média |
| 2 | Definer | 0,8571428 | Alta |
| 3 | PumpStreamHandler | 0,5714285 | Muito baixa |

A oportunidade candidata 1 é a família de classes de *AntClassLoader*, que possui quatro subclasses, sem chamadas literais, sem métodos diferentes da superclasse e não implementam o padrão *Visitor*. A subclasse *TaskConfiguredPathClassLoader* é interna e privada da classe *JUnitLauncherTask*. Já as outras três estão em pacotes abaixo da superclasse. Entretanto, há uma verificação pela classe *SplitClassLoader* com `instanceof`. Assim, essa oportunidade foi classificada como "média". A Figura 4.3 apresenta a estrutura dessa família de classes no projeto Apache-Ant.

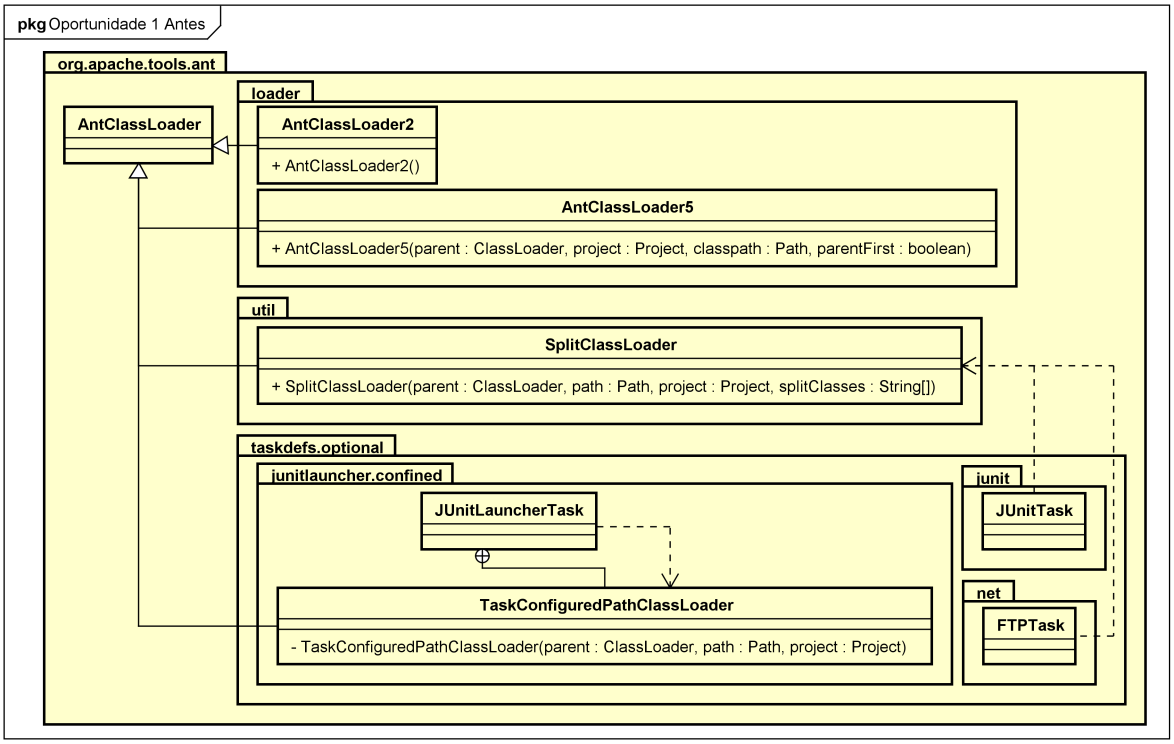


Figura 4.3 – Oportunidade candidata 1 antes da refatoração.

Com a aplicação da refatoração, a subclasse *SplitClassLoader* e a superclasse *AntClassLoader* foram movidas para o pacote *loader*, onde já se encontravam as duas outras subclasses. Foi feito isso pois o nome do pacote representa melhor as classes que ele contém. Os construtores foram protegidos e as classes ocultadas. Se optou por não modificar a subclasse *TaskConfiguredPathClassLoader*, pois ela realiza uma função específica somente para a classe que

a contém. Além disso, os autores das classes as fizeram para serem acessadas somente dessa maneira. Nessa refatoração foi possível realizar a troca da chamada direta pela subclasse *SplitClassLoader* pela chamada à superclasse *AntClassLoader* sem prejudicar os trechos de código. A Figura 4.4 apresenta o resultado da aplicação da refatoração atual.

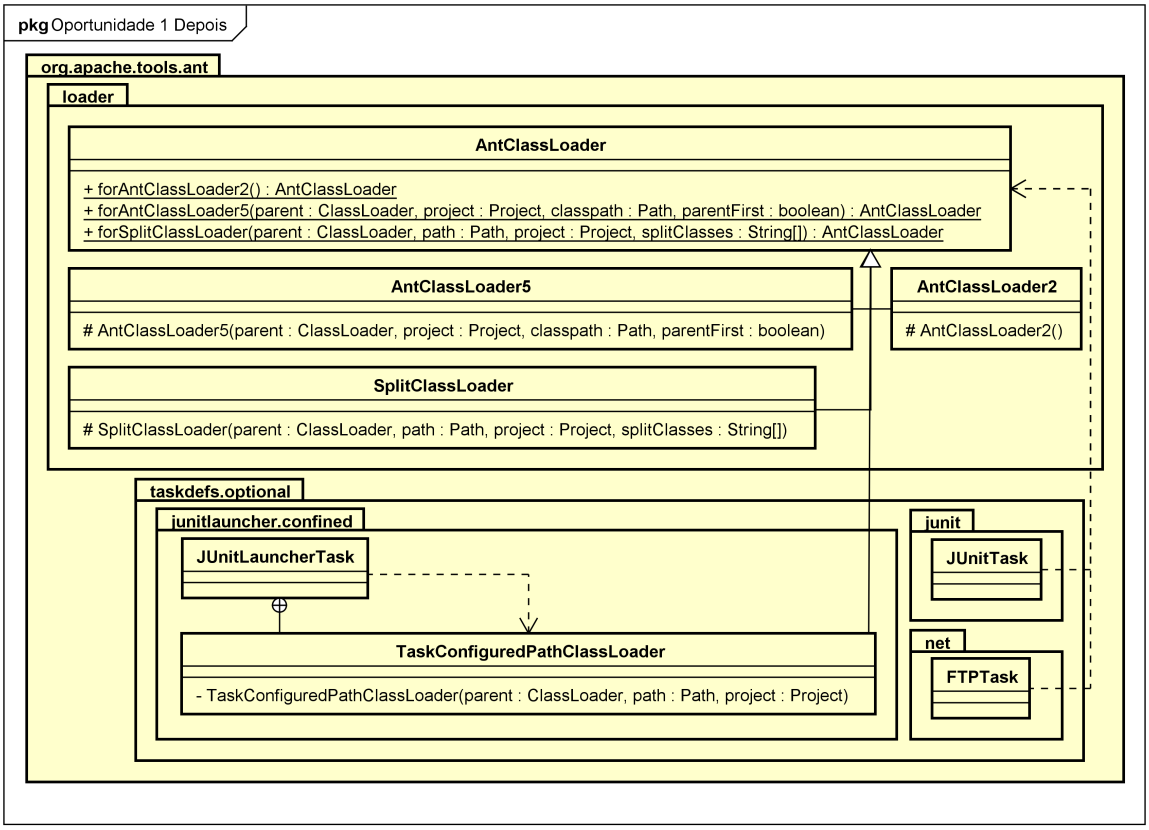


Figura 4.4 – Oportunidade candidata 1 após a refatoração.

Com o resultado *0,8571428*, a oportunidade candidata 2 é a hierarquia de classes de *Definer*. *Definer* é uma classe abstrata que possui três subclasses, todas dentro do seu pacote. As subclasses não implementam nenhum método diferente da superclasse, não implementam o padrão *Visitor* e não são chamadas literalmente. Foi aplicada a refatoração Encapsular Classes com *Factory*, apenas criando novos métodos na superclasse e ocultando os construtores das subclasses. Assim, essa oportunidade foi classificada como “alta”.

Por fim, a oportunidade candidata 3 é a hierarquia de classes de *PumpStreamHandler*, que possui três subclasses, todas abaixo na hierarquia de pacotes, onde há um método diferente da superclasse. Além disso, não implementam o padrão *Visitor* e não possuem chamadas literais. Porém, a subclasse *JUnitLogStreamHandler* é interna e protegida à classe *JUnitTask*, semelhante a primeira oportunidade candidata. Já a subclasse *RedirectingStreamHandler* está

em um pacote em que várias classes que usam e são usadas por essa classe estão protegidas dentro do pacote. Logo, a movimentação dessa subclasse para o pacote acima afetaria a visibilidade de todas as outras classes para que as mesmas continuem funcionando. Assim, apenas uma classe da hierarquia poderia ser refatorada, o que não valeria, nesse momento, o esforço da aplicação da refatoração. Dados esses motivos, essa oportunidade candidata foi classificada como uma oportunidade "muito baixa".

### 4.1.3 EclipseLink

A execução da mecânica para a busca de oportunidades de refatoração para Encapsular Classes com *Factory* sobre o projeto EclipseLink resultou em um total de *205* oportunidades candidatas. A Tabela 4.5 apresenta as três primeiras candidatas a oportunidades de refatoração, onde as duas primeiras foram consideradas "muito altas" para a aplicação da refatoração, enquanto a terceira foi considerada "alta".

Tabela 4.5 – Três primeiras candidatas a oportunidades de refatoração para Encapsular Classes com *Factory* do projeto EclipseLink.

| # | **Superclasse** $SC$ | **Resultado** | **Aplicabilidade** |
|---|---|---|---|
| 1 | IdentityMapAccessor | 1,3119047 | Muito alta |
| 2 | DeferredContentHandler | 1,1309523 | Muito alta |
| 3 | ANTLRStringStream | 0,7666666 | Alta |

A primeira candidata a oportunidade de refatoração é a família de classes a partir de *IdentityMapAccessor*, com o resultado aproximado de *1,311*. Ela possui cinco subclasses, onde todas possuem a mesma interface de comunicação que *IdentityMapAccessor*, com exceção da subclasse *DistributedSessionIdentityMapAccessor*, que possui o método público *initializeIdentityMapsOnServerSession* como é apresentado na Figura 4.5.

É possível verificar que a oportunidade candidata possui várias características que podem torná-la uma oportunidade de refatoração para Encapsular Classes com *Factory*. A maioria das subclasses possui uma interface de comunicação comum com a superclasse, estão no mesmo pacote, não são uma implementação do padrão *Visitor*, e não são chamadas literalmente.

Após a aplicação da refatoração para padrão, todos os construtores públicos das subclasses de *IdentityMapAccessor* ficaram protegidos, e para cada construtor há um método público para a instanciação dos objetos na superclasse, como pode ser observado na Figura 4.6. Vale observar que o método público *initializeIdentityMapsOnServerSession* é usado apenas uma vez em todo o projeto, sendo esse uso na própria classe. Em seu *javadoc* é explicado que o método

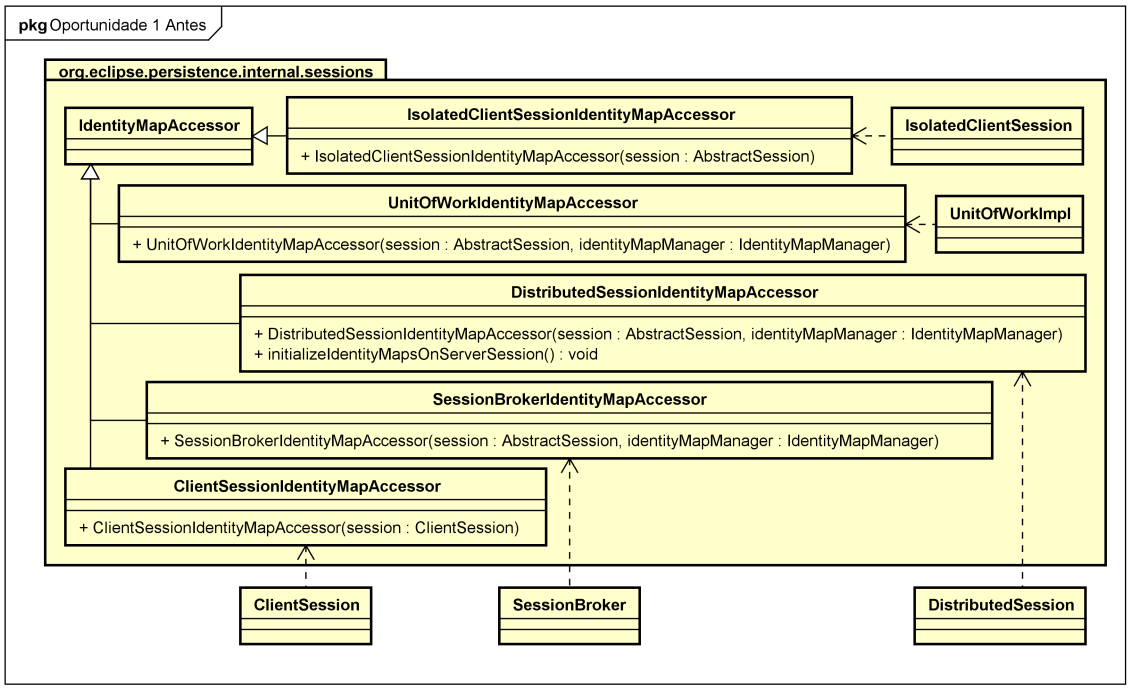


Figura 4.5 – Oportunidade candidata 1 antes da refatoração.

é interno. Dessa forma, o método foi modificado para privado sem afetar as demais classes do projeto.

A segunda candidata a oportunidade de refatoração é a família de classes de *DeferredContentHandler*, que possui seis subclasses e sem nenhuma chamada literal. Uma característica de grande impacto para que essa família de classes fosse considerada uma oportunidade é a baixa quantidade de métodos públicos que não pertencem às superclasses, onde apenas a classe *BinaryMappingContentHandler* contém dois métodos públicos que eram utilizados no próprio projeto. Dessa forma, a solução adotada foi permitir a publicidade da classe *BinaryMappingContentHandler*, mas proteger seus construtores, delegando assim para a superclasse a responsabilidade de instanciar novos objetos. Assim, a aplicabilidade da refatoração nessa família de classes foi considerada "muito alta".

A classe *ANTLRStringStream* que teve a classificação "alta" pois três das quatro de suas subclasses estavam no mesmo pacote que ela, com dois modificadores públicos e sem nenhuma chamada literal às suas subclasses. Os dois métodos públicos se chamam *load*, mas são de subclasses diferentes e tem parâmetros distintos. Porém, esses métodos são usados somente dentro das próprias subclasses e não possuem nenhuma documentação que possa auxiliar sobre sua intenção. Dessa forma, ambos métodos foram protegidos, pois ainda permite que uma classe do mesmo pacote a utilize caso necessário. Após, foram extraídos os atributos públicos

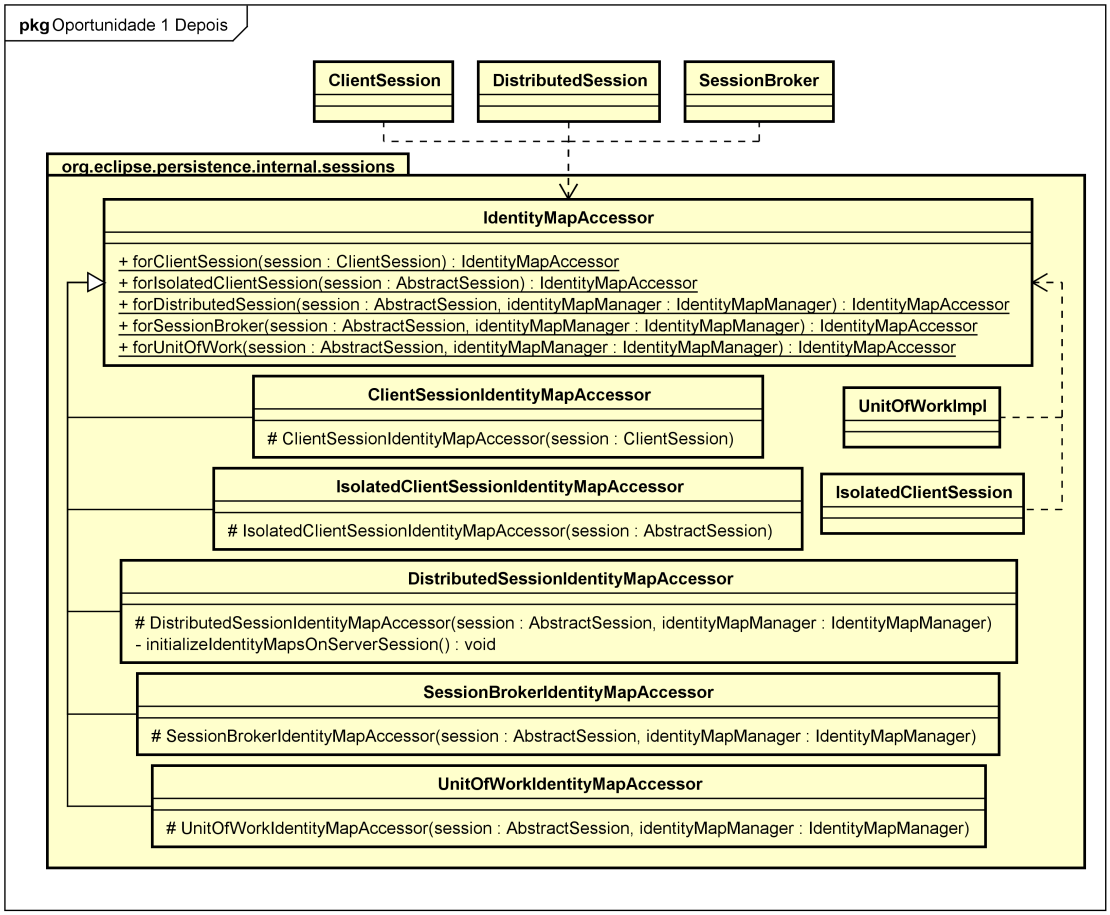


Figura 4.6 – Oportunidade candidata 1 após a refatoração.

de classe (*public static final*) da classe *ANTLRReaderStream* e inseridos na superclasse, para que classes externas possam utilizar. Por fim, a classe *CaseInsensitiveANTLRStringStream* foi movida para o mesmo pacote da superclasse *ANTLRStringStream*.

### 4.1.4 *Pattern Refactoring*

A Tabela 4.6 apresenta os três primeiros resultados de *13* oportunidades candidatas para a aplicação da refatoração. Nesse caso, os três primeiros resultados fazem parte da mesma hierarquia de classes. Receberam então a avaliação "baixa" de aplicabilidade pelos mesmos motivos de toda a hierarquia.

Tabela 4.6 – Três primeiras candidatas a oportunidades de refatoração para Encapsular Classes com *Factory* do projeto *Pattern Refactoring*.

| # | **Superclasse** $SC$ | **Resultado** | **Aplicabilidade** |
|---|---|---|---|
| 1 | Type | 1,5976190 | Baixa |
| 2 | StartLinePosition | 0,8571428 | Baixa |
| 3 | TreeParent | 0,2738095 | Baixa |

Para melhor relatar os motivos da avaliação das três oportunidades candidatas, foi escolhido discutir a aplicação da refatoração na hierarquia como um todo, e não apenas a aplicação sobre a primeira oportunidade. A Figura 4.7 apresenta a estrutura das classes antes da aplicação da refatoração. A primeira candidata se encontra no meio da hierarquia, a segunda abaixo, e a terceira acima na hierarquia.

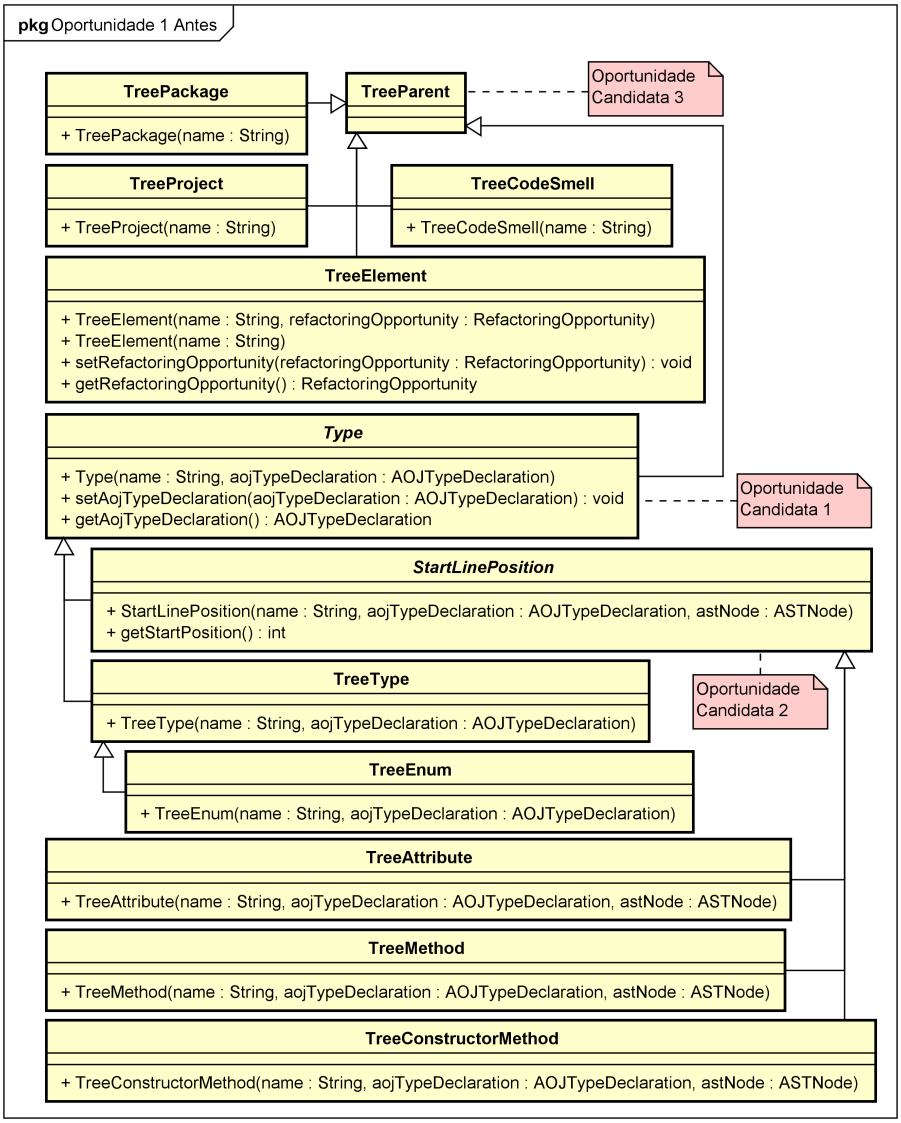


Figura 4.7 – Oportunidade candidata 1 antes da refatoração.

Todas as classes dessa hierarquia estão no mesmo pacote, não implementam o padrão *Visitor*, e não há chamadas literais à elas. Apenas as classes *TreeElement*, *Type*, e *StartLinePosition* possuem métodos públicos diferentes da superclasse *TreeParent*. Entretanto, todas as classes dessa hierarquia são chamadas usando `instanceof`, pois é através do seu tipo que é determinado o ícone que aparecerá no componente de visualização do *Pattern Refactoring* e do

AOPJungle. Além disso, todas as classes *Comparators* e *Providers* necessitam chamar essas classes para realizar o *casting* e utilizar algum método específico da classe.

A Figura 4.8 apresenta a família de classes das oportunidades após a aplicação da refatoração. Diferentemente dos outros resultados, foi escolhido aplicar a refatoração da oportunidade candidata 3, pois ficaria mais didático para explicar quais as alterações foram necessárias.

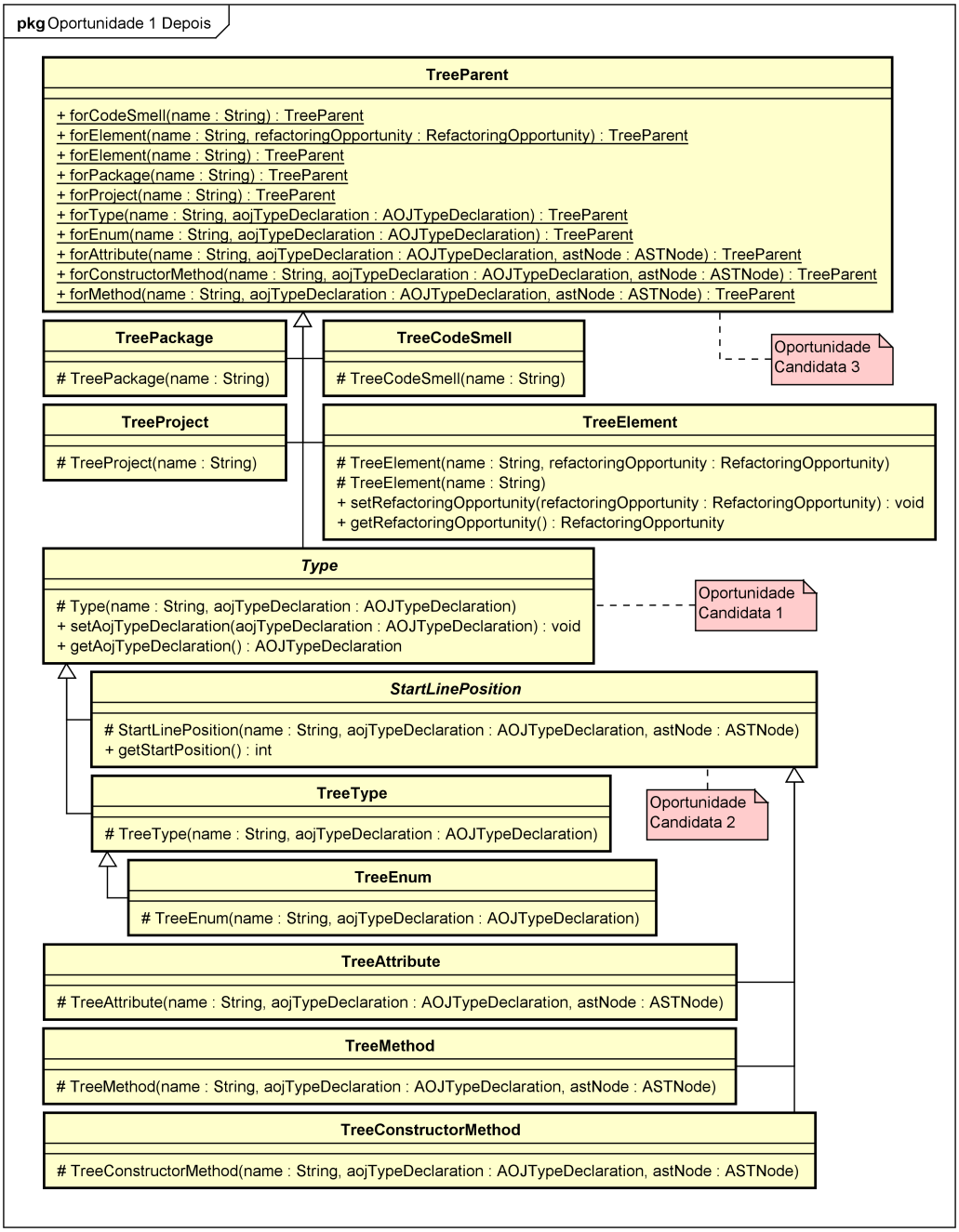


Figura 4.8 – Oportunidade candidata 1 após a refatoração.

Como todos os membros dessa hierarquia de classes são chamados com `instanceof`, não foi possível nem interessante ocultar as subclasses das outras classes do projeto. Uma

maneira de resolver seria definir um atributo na superclasse, e criar uma constante também na superclasse para cada subclasse. Assim, seria possível trocar a verificação do `instanceof` nas classes clientes, pela igualdade do atributo com as constantes. Entretanto, isso adiciona mais complexidade ao projeto, pois a cada nova subclasse, seria necessário um novo método para sua instanciação, assim como mais uma constante que a represente.

Dessa forma, apenas se ocultaram os métodos construtores das subclasses, e se criou um método estático para cada construtor na superclasse *TreeParent*. Após, as instanciações das subclasses nas classes clientes foram substituídas pelos respectivos métodos estáticos.

Essas características descritas servem para as três oportunidades candidatas. Logo, a aplicabilidade das três oportunidades candidatas foi classificada como "baixa". Assim, este autor e desenvolvedor do *plug-in Pattern Refactoring* decidiu não aplicar essa refatoração no projeto.

### 4.1.5 Avaliação dos resultados para a mecânica Encapsular Classes com *Factory*

A Figura 4.9 apresenta os totais de candidatas a oportunidades de refatoração por projeto. O projeto EclipseLink é o que tem a maior quantidade de candidatas (*205*), enquanto o *plug-in Pattern Refactoring* tem a menor quantidade (*13*). Isso é devido ao tamanho de cada projeto. O EclipseLink possui uma quantidade considerável de linhas de código e de classes, enquanto o *Pattern Refactoring* é um projeto de menor tamanho.

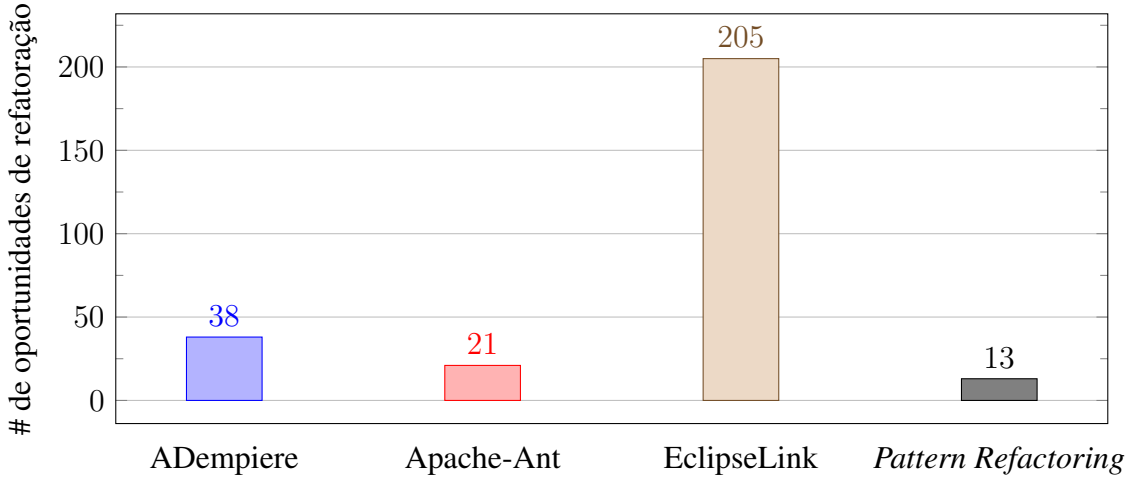


Figura 4.9 – Total de candidatas a oportunidades de refatoração encontradas para Encapsular Classes com *Factory*.

Uma característica que pode ser verificada é a baixa quantidade de oportunidades para o projeto ADempiere em relação a seu tamanho em linhas de código e quantidade de classes. É o

maior projeto desse estudo de caso, entretanto não possui uma estrutura com muitas hierarquias de classes, sendo um projeto mais vertical. Ou seja, contém poucas famílias de classes, mas nessas famílias há muitos membros. Como exemplo, a maior família de classes de *PO* (*Persistence Object*) possui 1393 subclasses. Já o EclipseLink possui uma estrutura mais horizontal, com várias hierarquias de classes, e poucos membros, onde a maior hierarquia possui 150 membros, pertencentes a classe *ORMetadata*.

No projeto EclipseLink foram encontradas várias oportunidades para a aplicação da refatoração Encapsular Classes com *Factory*. O que não aconteceu no projeto Apache-Ant, onde poucas foram encontradas. Isso pode ser uma relação do tamanho do projeto com a sua maturidade. Onde quanto mais maduro e estável um projeto é, menor a chance de aplicação da refatoração nas oportunidades candidatas.

Outra característica é o acoplamento no projeto *Pattern Refactoring*, onde a refatoração não foi possível pela necessidade da publicidade das classes. Nesse caso, talvez seja necessário primeiro reduzir o acoplamento entre as classes do projeto, para que assim seja possível a identificação e refatoração das oportunidades.

A Tabela 4.7 apresenta as dez primeiras oportunidades de refatoração utilizando a Função 4.1 para todos os projetos. As oportunidades estão ordenadas pelo valor do seu resultado, sendo exibidas suas respectivas posições em cada projeto. O EclipseLink é projeto que mais possui oportunidades presentes, e o *Pattern Refactoring* é quem possui a candidata na primeira posição.

Tabela 4.7 – Dez primeiras candidatas a oportunidades de refatoração para Encapsular Classes com *Factory* dos quatro projetos.

| # | # no projeto | Superclasse $SC$ | Resultado | Projeto |
|---|---|---|---|---|
| 1 | 1 | Type | 1,5976190 | *Pattern Refactoring* |
| 2 | 1 | IdentityMapAccessor | 1,3119047 | EclipseLink |
| 3 | 2 | DeferredContentHandler | 1,1309523 | EclipseLink |
| 4 | 1 | BaseResourceCollectionWrapper | 1,0666666 | Apache-Ant |
| 5 | 1 | PopupAction | 0,9095238 | ADempiere |
| 6 | 2 | StartLinePosition | 0,8571428 | *Pattern Refactoring* |
| 7 | 2 | AntClassLoader | 0,8571428 | Apache-Ant |
| 8 | 3 | Definer | 0,8571428 | Apache-Ant |
| 9 | 3 | ANTLRStringStream | 0,7666666 | EclipseLink |
| 10 | 4 | UnmarshalRecordImpl | 0,7666666 | EclipseLink |

A mecânica de busca por oportunidades de refatoração para Encapsular Classes com *Factory* foi avaliada em diferentes projetos. Se verificou a presença de oportunidades de refato-

ração com “alta” probabilidade de serem aplicadas. Sendo assim, é uma mecânica que auxilia o desenvolvedor a identificar oportunidades de refatoração, para que ele possa aplicá-las, melhorando assim a qualidade do código.

A mecânica proposta neste trabalho pode ser melhorada. Em relação ao indício *visitor*, ele atualmente verifica se implementa o padrão *Visitor* através dos nomes da classe e dos métodos. Essa característica pode ser identificada na maioria dos casos da implementação desse padrão, mas não em todos. Se o desenvolvedor utilizar um idioma diferente do inglês, pode ser que o padrão não seja identificado.

Outra melhoria a ser adicionada é o indício à expressão `instanceof`, pois em vários casos as classes precisam ser públicas para que seja realizada uma verificação de tipo. Assim, esse indício auxiliaria a filtrar as hierarquias de classes que necessitam ser visíveis às outras classes.

## 4.2 Resultados para a mecânica de busca para Substituir Árvore Implícita por *Composite*

É necessário informar valores de pesos e parâmetros para as Funções Heurísticas 3.3 e 3.4 para que esta mecânica de busca possa ser executada. As Funções 4.2 e 4.3 apresentam os valores de parâmetros e pesos utilizados em todos os projetos testados.

$$resultado = \big(normalizarRetilineo(quantidadeDeTags, 6) * 1 + normalizarRetilineo(quantidadeDeTagsDeFechamento, 2) * 1\big) \quad (4.2)$$

Cada indício é aplicado em sua respectiva Função *normalizarRetilineo* com um valor de parâmetro. O resultado da aplicação da Função é então multiplicado pelo seu respectivo peso. Essa abordagem permite que cada resultado das funções sobre os indícios possuam uma relevância independente. É possível assim que possa ser priorizado um indício em relação a outro indício nas Funções 4.2 e 4.3. Em ambas as funções da mecânica atual, os resultados são adicionados. Logo, quanto maior o valor do peso de um indício em relação aos outros pesos, maior será sua importância para o resultado final da mecânica.

$$resultadoSomatorio = \big(normalizarRetilineo(somatorioDosTotais, 1) * 1\big) \quad (4.3)$$

Durante a aplicação da mecânica de busca nos projetos deste estudo de caso todos os pesos receberam o valor *um*. Significa que os resultados dos indícios tiveram a mesma importância entre si. O desenvolvedor pode utilizar outro formato ou um método para valorar cada peso, uma maneira é através de métodos multicritérios de decisão.

A Tabela 4.8 apresenta os valores dos parâmetros utilizados nas Funções 4.2 e 4.3 e seus respectivos significados para os resultados.

Tabela 4.8 – Valores dos parâmetros dos indícios.

| **Indício** | **Valor do parâmetro** | **Significado** |
|---|---|---|
| Quantidade de *tags* | 6 | Privilegia as oportunidades com 6 ou mais *tags*. |
| Quantidade de *tags* de fechamento | 2 | Privilegia as oportunidades com 2 ou mais *tags* de fechamento. |
| Somatório dos totais | 1 | Prestigia as oportunidades que possuam uma ou mais *tags*. Ou seja, quanto mais *tags* presentes, melhor. |

Para a execução da mecânica é necessário que o desenvolvedor informe valores de pesos e parâmetros. Não há ainda uma definição para tais valores que possam ser considerados "padrão". É necessário que se faça uma análise com um grupo de dados maior, e se verifique quais valores podem ser definidos como "padrão". Além disso, é preciso investigar os valores dos limiares que definem a aplicabilidade de uma refatoração.

O indício $quantidadeDeTags$ representa a quantidade de *tags* XML e derivadas o método possui. A função *normalizarRetilineo* aplicada a esse indício indica que quanto maior a quantidade de *tags* pertencentes ao método sendo verificado, maior será seu valor. A Tabela 4.8 apresenta o valor *seis* utilizado para o parâmetro do indício $quantidadeDeTags$, ou seja, obterá o valor *um* quando o método possuir *seis tags*. Para a execução em outros projetos, pode-se utilizar valores em um intervalo de *três* a *20*.

Em seguida, presente na Função 4.2 há o indício $quantidadeDeTagsDeFechamento$, que possui o valor da quantidade de *tags* de fechamento identificadas durante a execução da mecânica sob um respectivo método. É também utilizada a função *normalizarRetilineo*, pois quanto maior a quantidade de *tags* de fechamento, maior a chance da oportunidade candidata ser uma oportunidade efetiva de refatoração para Substituir Árvore Implícita por *Composite*. O valor de parâmetro para esse indício é *dois*, como a Tabela 4.8 apresenta, pois a partir de duas *tags* de fechamento o método pode ser uma oportunidade aplicável. Esse valor pode variar de *um* a *dez*.

Depois de executada a primeira parte da mecânica atual que o Algoritmo 3 descreve, é

calculada a quantidade total de cada *tag* que foi utilizada no projeto. A Tabela 4.8 apresenta o valor *um* para o parâmetro do indício $somatorioDosTotais$, que é então utilizado na função *normalizarRetilineo*, significando que a presença de uma *tag* que se repete muito pelo projeto terá um valor muito alto para a oportunidade candidata.

As próximas seções descrevem os resultados da execução da mecânica de busca para Substituir Árvore Implícita por *Composite* aplicada aos projetos ADempiere, Apache-Ant, EclipseLink e *Pattern Refactoring*.

### 4.2.1 ADempiere

A Tabela 4.9 apresenta os três primeiros resultados da execução da mecânica de busca por oportunidades de refatoração para Substituir Árvore Implícita por *Composite*, de um total de *67* oportunidades candidatas. As três oportunidades candidatas foram consideradas com aplicabilidade "muito alta" após a avaliação.

Tabela 4.9 – Três primeiras candidatas a oportunidades de refatoração para Substituir Árvore Implícita por *Composite* do projeto ADempiere.

| # | Classe | Método | Resultado | Aplicabilidade |
|---|---|---|---|---|
| 1 | HelpInfo | *getHelpText* | 10102,666 | Muito alta |
| 2 | MCommissionRun | *createDetail* | 1710,500 | Muito alta |
| 3 | StatusInfo | *doGet* | 1141,833 | Muito alta |

A Listagem 4.1 apresenta o código-fonte do método *getHelpText* pertencente à classe *HelpInfo*. O objetivo deste método é construir uma página HTML com informações de auxílio que serão exibidas a posteriori pelo ADempiere. A linguagem HTML possui sua estrutura baseada em árvore. Logo, sua utilização de forma implícita em Strings caracteriza uma oportunidade de refatoração para Substituir Árvore Implícita por *Composite*.

Listagem 4.1 – Oportunidade candidata 1 antes da refatoração.

```
1  private String getHelpText() {
2    StringBuffer doc = new StringBuffer();
3    // prolog
4    doc.append("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"
       \"http://www.w3.org/TR/html4/strict.dtd\">");
5    // start of document
6    doc.append("<html>");
7    // document header
8    doc.append("<head>");
9    ...
10   doc.append("</head>");
11   // start of body
12   doc.append("<body>");
```

```
13    ...
14    // end of body
15    doc.append("</body>");
16    // end of document
17    doc.append("</html>");
18
19    return doc.toString();
20  }
```

O método possui 287 linhas de código, onde 211 linhas fazem parte da construção da árvore implícita em HTML. Esse método ficou na primeira posição pois possui 400 *tags*, das quais 198 são *tags* de fechamento (“/>”). Além disso, boa parte das *tags* que o método possui são de alta relevância, aumentando assim seu resultado final. Com todas essas características, esta oportunidade recebeu uma avaliação de aplicabilidade “muito alta”.

Para aplicar a refatoração atual, é necessário identificar os nós da composição e criar um conjunto de classes que representem os nós. Esse conjunto de classes ficará disponível para qualquer outra classe cliente do projeto que necessite utilizar tal estrutura. Dessa maneira, a solução desenvolvida foi utilizada nas três candidatas a oportunidades de refatoração. A Figura 4.10 apresenta a estrutura das classes do padrão *Composite* após a aplicação da refatoração.

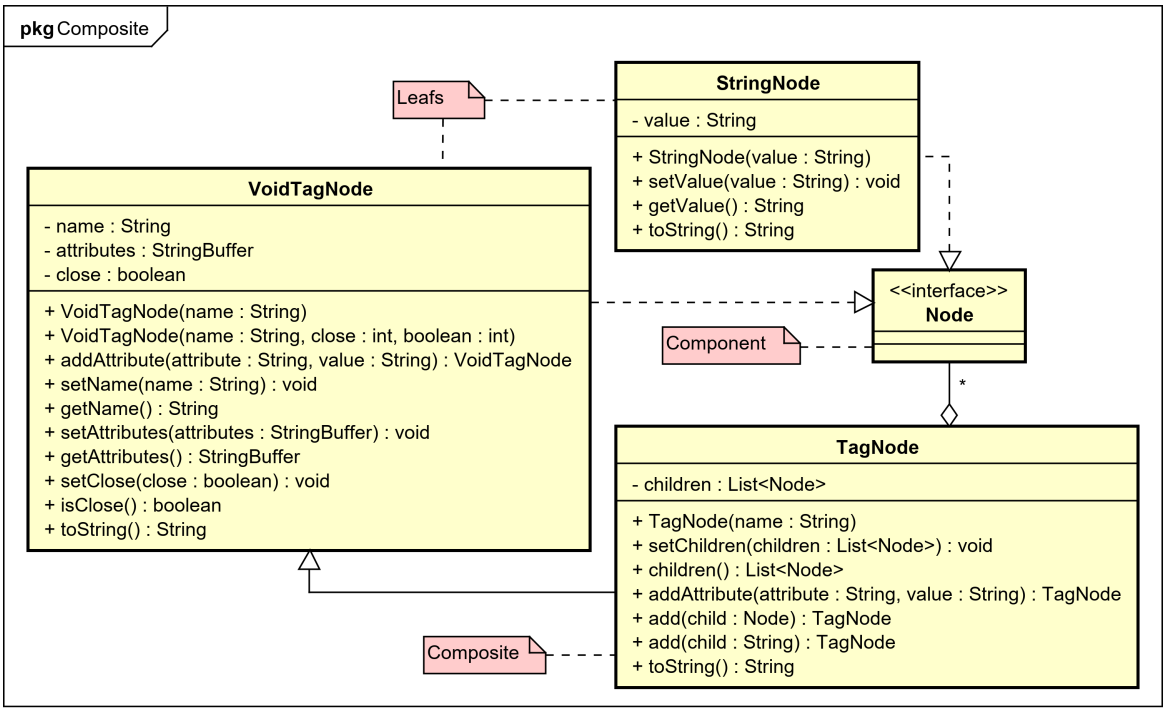


Figura 4.10 – Implementação do Padrão *Composite* para o projeto ADempiere.

Uma forma de codificar o *Composite* seria criar uma classe para cada elemento HTML,

assim fornecendo a estrutura completa que daria suporte à especificação trazida da W3C[5]. Entretanto, tal estrutura não se fez necessária pois apenas poucos elementos foram utilizados, e esses elementos possuíam uma estrutura em comum. Essa estrutura foi abstraída e transformada nas classes *VoidTagNode*, *TagNode* e *StringNode* como é apresentado na Figura 4.10.

A classe *Node* representa qualquer nó pertencente ao HTML. Ela não possui nenhum atributo nem método, pois não há estado nem comportamento comum entre as classes folha. A classe *VoidTagNode* representa os elementos que não possuem *tag* de fechamento nem valor, como o `<br>` e `<meta>`, mas podem possuir um conjunto de atributos. É possível utilizar tais *tags* com um fechamento interno (`<br />` por exemplo), sendo este opcional através do atributo *close*. O restante dos elementos HTML podem ser representados com a classe *TagNode*, que são compostos por outros nós, como `<body>...<body/>`, `<p>...</p>`. Por fim, a classe *StringNode* representa apenas uma String que estará dentro de uma *tag*.

Com essa implementação do padrão *Composite*, o código-fonte do método *getHelpText* foi refatorado, substituindo as inserções em String de *tags* HTML por suas respectivas instâncias dos nós de *Composite*, como a Listagem 4.2 apresenta.

Listagem 4.2 – Oportunidade candidata 1 depois da refatoração.

```
1  private String getHelpText() {
2    StringBuffer doc = new StringBuffer();
3
4    TagNode defaultMarker = s_parameters.isDefaultMigrationIsUpgrade() ?
         getDefaultMarkerTagNode() : null;
5
6    TagNode defaultMarker2 = !s_parameters.isDefaultMigrationIsUpgrade() ?
         getDefaultMarkerTagNode() : null;
7
8    String message = getHtmlMessage("guiButtonViewTraceHelp");
9    message = message.replaceAll("\\{0\\}", new
         Integer(s_parameters.MAXLOGLINES).toString());
10
11   String message2 = getHtmlMessage("guiButtonViewWarningsHelp");
12   message2 = message2.replaceAll("\\{0\\}", new
         Integer(s_parameters.MAXLOGLINES).toString());
13
14   String message3 = getHtmlMessage("guiButtonViewErrorsHelp");
15   message3 = message3.replaceAll("\\{0\\}", new
         Integer(s_parameters.MAXLOGLINES).toString());
16
17   // prolog
18   doc.append(new VoidTagNode("!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML
         4.01//EN\" \"http://www.w3.org/TR/html4/strict.dtd\"", false));
19
20   // start of document
21   doc.append(new TagNode("html").
22     // document header
```

[5] https://www.w3.org/TR/html52/

```
23      add(new TagNode("head").
24        ...
25        ).
26      // start of body
27      add(new TagNode("body").
28        ...
29        // end of body
30        )
31      // end of document
32      );
33
34    return doc.toString();
35  }
```

Algumas verificações que eram realizadas durante a construção do documento foram extraídas e colocadas no início do método como é apresentado através das linhas quatro a 15.

Após a aplicação da refatoração na primeira oportunidade de refatoração houve a simplificação da construção das *tags* HTML, facilitando futuras manutenções no código. Além disso, a adição de novos elementos HTML podem ser realizados utilizando a estrutura *Composite*, que simplifica a implementação e diminui a chance de inserção de falhas. Por fim, instruções repetitivas como as *tags* de fechamento são extraídas do método e isoladas nas classes do padrão.

A segunda oportunidade de refatoração candidata é o método *createDetail* pertencente à classe *MCommissionRun*. Esse método cria, em HTML, os detalhes da comissão dos funcionários. Estão contidos neste método 33 *tags*, sendo oito *tags* de fechamento. Apesar de possuir menos *tags* que a terceira oportunidade candidata, 29 das 33 das *tags* presentes no método *createDetail* estão entre as duas mais recorrentes. Além disso, todas as *tags* presentes podiam ser refatoradas. Dadas essas características, essa oportunidade foi considerada com uma aplicabilidade "muito alta".

Foi aplicada a refatoração sob a segunda oportunidade, substituindo-se todas as *tags* que produziam uma árvore implícita por suas respectivas instâncias da implementação do padrão *Composite* criado para o projeto ADempiere. Para a simplificar o uso da *tag* `<br>`, criou-se apenas uma instância sua (`VoidTagNode br = new VoidTagNode("br", false);`) e se aplicou em todas as ocorrências. Um exemplo de sua aplicação é: `m_comissionLog.append(br + " --------One match found end" + br + br);`.

A terceira oportunidade de refatoração é o método *doGet* da classe Servlet *StatusInfo*. Este método é responsável por receber uma solicitação GET[6], trazer informações do estado de alguns serviços e configurações do servidor que está executando o software ADempiere, e exi-

[6] https://tools.ietf.org/html/rfc2616#page-53

bir para o usuário. Tais informações são exibidas em uma página HTML. Assim, é criada uma árvore implícita com 41 *tags* para estruturar a página. Nem todas as *tags* possuem uma recorrência alta, fazendo com que ficasse na terceira posição. Da mesma forma que as oportunidades anteriores, esta oportunidade de refatoração foi considerada com "muito alta" aplicabilidade.

Todas as *tags* que formavam a árvore implícita foram mudadas para instâncias das classes do padrão *Composite* como resultado da aplicação da refatoração.

### 4.2.2 Apache-Ant

Não foi encontrada nenhuma oportunidade de refatoração no projeto Apache-Ant. Foi verificado manualmente por possíveis falsos negativos, entretanto não foram encontrados.

As únicas árvores implícitas encontradas estavam presentes em algumas documentações Java (Javadoc), mas com poucas instâncias. A pesquisa de árvores implícitas em Javadoc está fora do escopo deste trabalho. Mesmo que estivesse no escopo, as *tags* encontradas são escassas, não sendo interessante sua refatoração.

### 4.2.3 EclipseLink

A Tabela 4.10 apresenta o único resultado obtido da execução desta mecânica de busca. Tal oportunidade foi considerada com aplicabilidade "muito alta".

Tabela 4.10 – Única candidata a oportunidade de refatoração para Substituir Árvore Implícita por *Composite* do projeto EclipseLink.

| # | Classe | Método | Resultado | Aplicabilidade |
|---|---|---|---|---|
| 1 | XMLLogFormatter | *format* | 44,5 | Muito alta |

Foi realizado uma busca manual por oportunidades para verificar se haviam falsos negativos. Porém, não foram encontradas nenhuma oportunidade.

A função do método *format* é formatar uma mensagem de log para um texto em XML. A construção da estrutura em XML é feita diretamente dentro de Strings. A estrutura XML possui 21 *tags*, todas com *tags* de fechamento, totalizando 42 *tags* dentro do método. Desse total, apenas há repetição de três *tags* em duas ocasiões. Assim, o valor da somatório dos totais obteve o valor *27*. A Listagem 4.3 apresenta um recorte do código da oportunidade candidata.

Listagem 4.3 – Oportunidade candidata 1 antes da refatoração.

```
1 public String format(LogRecord record0) {
2   if (!(record0 instanceof EclipseLinkLogRecord)) {
```

```
3       return super.format(record0);
4     } else {
5       EclipseLinkLogRecord record = (EclipseLinkLogRecord)record0;
6
7       StringBuffer sb = new StringBuffer(500);
8       sb.append("<record>\n");
9
10      if (record.shouldPrintDate()) {
11        sb.append("  <date>");
12        appendISO8601(sb, record.getMillis());
13        sb.append("</date>\n");
14
15        sb.append("  <millis>");
16        sb.append(record.getMillis());
17        sb.append("</millis>\n");
18      }
19
20      sb.append("  <sequence>");
21      sb.append(record.getSequenceNumber());
22      sb.append("</sequence>\n");
23
24      ...
25
26      sb.append("</record>\n");
27      return sb.toString();
28    }
29  }
```

O primeiro passo para a aplicação da refatoração foi identificar os nós que farão parte da composição e sua hierarquia. A implementação do padrão *Composite* foi realizada de forma semelhante com a implementação no projeto ADempiere. Entretanto, a oportunidade candidata não possui *tags* vazias (por exemplo `<br />`) e também não possui atributos. Assim, foram implementadas as classes *TagNode* que representa uma *tag* qualquer, e a *StringNode* que representa qualquer valor que será interno à *tag*. A Figura 4.11 apresenta a implementação do padrão para o projeto EclipseLink.

A Listagem 4.4 apresenta o código do método *format* após a aplicação da refatoração. As *tags* presentes nas expressões Strings foram substituídas por objetos da classe *TagNode*, e seus respectivos valores foram colocados em objetos da classe *StringNode*.

Listagem 4.4 – Oportunidade candidata 1 depois da refatoração.

```
1  public String format(LogRecord record0) {
2    if (!(record0 instanceof EclipseLinkLogRecord)) {
3      return super.format(record0);
4    } else {
5      EclipseLinkLogRecord record = (EclipseLinkLogRecord) record0;
6
7      TagNode rec = new TagNode("record").add("\n  ");
8
9      if (record.shouldPrintDate())
10       rec.add(new TagNode("date").add(appendISO8601(record.getMillis()))).
```

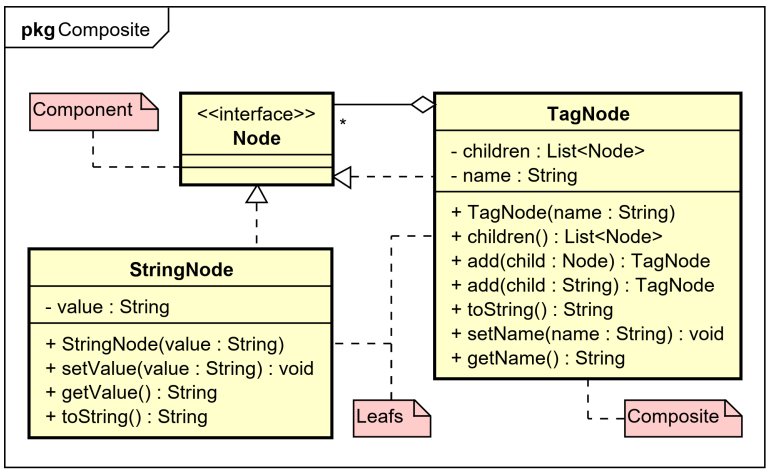


Figura 4.11 – Implementação do Padrão *Composite* para o projeto EclipseLink.

```
11          add("\n  ").
12          add(new TagNode("millis").add(String.valueOf(record.getMillis()))).
13          add("\n  ");
14
15      rec.add(new TagNode("sequence").
16        add(String.valueOf(record.getSequenceNumber()))).
17        add("\n  ");
18
19      ...
20
21      StringBuffer sb = new StringBuffer(500);
22      sb.append(rec + "\n");
23      return sb.toString();
24    }
25  }
```

A aplicação da refatoração Substituir Árvore Implícita por *Composite* na oportunidade candidata simplificou a construção dos nós das *tags* XML. As *tags* utilizadas não fazem parte de uma descrição ou norma, assim, a estrutura de composição construída possibilita a adição de novas *tags* sem ser necessário alterar a estrutura do padrão de projeto. Ou seja, caso seja necessário adicionar uma nova estrutura XML à esta árvore, somente será preciso instanciar objetos da classe *TagNode*.

### 4.2.4 *Pattern Refactoring*

A mecânica de busca foi executada no projeto *Pattern Refactoring* e não foram encontradas oportunidades de refatoração. Foi realizada em seguida uma busca manual por locais no código que pudessem ser consideradas oportunidades, entretanto não foram localizadas.

Essa mecânica busca por indícios de árvore implícita, entretanto o projeto não possui

nenhuma estrutura com as características de uma árvore implícita. Afinal, o projeto não trabalha com nenhuma estrutura semelhante a construção de nós em XML ou derivados.

### 4.2.5 Avaliação dos resultados para a mecânica Substituir Árvore Implícita por *Composite*

A Figura 4.12 apresenta os totais de oportunidades de refatoração de cada projeto. O projeto ADempiere possui a maior quantidade de oportunidades candidatas (*67*), enquanto não foram encontradas oportunidades nos projetos Apache-Ant e *Pattern Refactoring*. Já no projeto EclipseLink foi encontrada apenas *uma* oportunidade. Essa diferença pode ser justificada pelos indícios pesquisados pela mecânica. O principal indício é a presença dos caracteres "<" e ">" em expressões Strings, formando uma *tag*. Tal utilização não é usual em alguns sistemas. Em caso de necessidade, a formação de estruturas em árvore como XML e HTML podem ser delegadas à bibliotecas externas aos projetos.

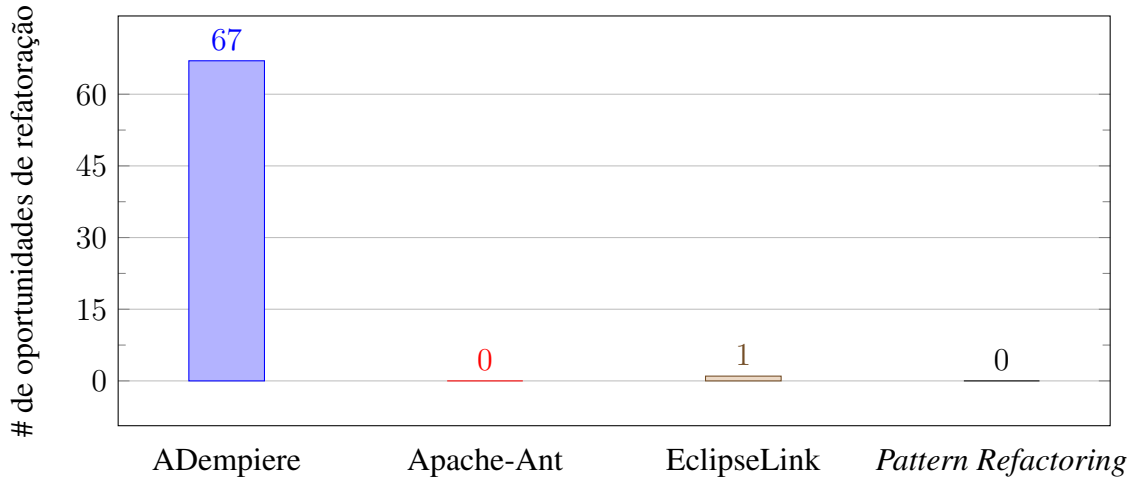


Figura 4.12 – Total de candidatas a oportunidades de refatoração encontradas para Substituir Árvore Implícita por *Composite*.

O projeto ADempiere realiza atividades diferentes dos demais projetos, como exibir resultados em páginas HTML, interagir com banco de dados, gerar relatórios, entre outros, para satisfazer seu propósito de ser uma suíte empresarial ERP e CRM. Assim, nem sempre é interessante criar uma estrutura de composição para atender poucas *tags* XML ou HTML. Entretanto, tais estruturas são incrementadas com passar do tempo, que aumenta o débito técnico, sendo necessário em algum momento que seja aplicada uma refatoração. Dessa maneira, uma possível causa da alta quantidade de candidatas a oportunidades de refatoração em relação aos outros projetos seja o débito técnico relacionado à exibição e registro de conteúdo HTML das classes.

A Tabela 4.11 apresenta as dez primeiras oportunidades de refatoração da execução da

mecânica de busca para Substituir Árvore Implícita por *Composite*. Todas as dez primeiras oportunidades pertencem ao projeto ADempiere, onde metade delas são da classe *MCommissionRun*.

Tabela 4.11 – Dez primeiras candidatas a oportunidades de refatoração para Substituir Árvore Implícita por *Composite* dos quatro projetos.

| # | # no projeto | Classe | Método | Resultado | Projeto |
|---|---|---|---|---|---|
| 1 | 1 | HelpInfo | *getHelpText* | 10102,666 | ADempiere |
| 2 | 2 | MCommissionRun | *createDetail* | 1710,500 | ADempiere |
| 3 | 3 | StatusInfo | *doGet* | 1141,833 | ADempiere |
| 4 | 4 | MCommissionRun | *prepareIt* | 1033,333 | ADempiere |
| 5 | 5 | MRP | *doIt* | 1009,833 | ADempiere |
| 6 | 6 | MCommissionRun | *createMovements* | 930,000 | ADempiere |
| 7 | 7 | HtmlDashboard | *createHTML* | 612,333 | ADempiere |
| 8 | 8 | HtmlDashboard | *goalsDetail* | 504,333 | ADempiere |
| 9 | 9 | MCommissionRun | *processCommissionLine* | 467,166 | ADempiere |
| 10 | 10 | MCommissionRun | *invoiceCompletelyPaid* | 421,833 | ADempiere |

A oportunidade candidata do projeto EclipseLink foi classificada na *37* posição. O método *format*, com o valor *44,5* ficou entre os métodos *stripHtml* (*45,5*) e *createHTMLTable* (*42,5*) das classes HtmlDashboard e OrderReceiptIssue respectivamente.

Vários métodos da classe *MCommissionRun* tiveram uma classificação alta pois a classe como um todo realiza a construção do documento em HTML do cálculo das comissões. A classe pesquisa as informações do banco de dados, realiza os cálculos necessários e monta o documento com as *tags*.

Todos os resultados obtidos da execução da mecânica possuem *tags* HTML ou XML, alguns métodos com poucas e outros com várias *tags*. No momento que é implementado o *Composite* para o projeto, todas as oportunidades candidatas podem ser refatoradas, assim, com uma alta probabilidade de aplicação. Pode-se concluir que a mecânica de busca para Substituir Árvore Implícita por *Composite* criada neste trabalho auxilia o desenvolvedor a identificar oportunidades de refatoração.

É possível implementar o padrão *Composite* de forma genérica, servindo tanto para as *tags* HTML quanto XML. Outra maneira de implementar o padrão é especificando as regras e criando hierarquias entre as *tags* (por exemplo a *tag* `<html>` deve possuir apenas as *tags* `<head>` e `<body>` nessa ordem). O indício do somatório das quantidades totais vem ao encontro dessa segunda maneira de implementação do *Composite*. É possível com o valor desse indício verificar e implementar apenas as *tags* em maior quantidade com suas regras específicas.

Quanto às instâncias dos nós, uma forma diferente de criá-las é através de métodos fábrica ou utilizando o padrão *Strategy*. Essas implementações podem diminuir o acoplamento da classe *Client* que utiliza as *tags* implementadas pelo *Composite*.

Uma melhoria que pode ser adicionada à mecânica é um valor de indício para o projeto, pois o padrão de composição após implementado pode ser utilizado em qualquer lugar dentro do projeto. Dessa forma, seria possível criar e verificar um limiar que, a partir de certo valor, seria positiva a implementação do *Composite* e sua utilização em todo o projeto.

Além disso, a pouca quantidade de oportunidades encontradas pode prejudicar os valores de parâmetros e pesos utilizados, pois não foi possível verificar tais valores aplicados a diferentes oportunidades de outros projetos. Isso significa que o grau de confiança dos valores utilizados para as Funções 4.2 e 4.3 não pode ser considerado alto.

## 4.3 Resultados para a mecânica de busca para Substituir Envio Condicional por *Command*

A realização das buscas por oportunidades de refatoração para Substituir Envio Condicional por *Command* necessita que o usuário informe valores para os parâmetros e pesos da Função Heurística 3.5. Dessa forma, para a execução das buscas em três projetos foram utilizados os valores apresentados pela Função 4.4. Como os valores dos indícios do projeto Apache-Ant são homogêneos, os valores dos pesos foram ajustados para que as oportunidades mais relevantes fossem melhor ordenadas. A Seção 4.3.2 apresentará esses valores.

$$\begin{aligned} resultado = \big(&normalizarRetilineo(quantidadeCondicionais, 5) * 0,1 \\ &+ normalizarRetilineo(quantidadeLoC, 30) * 0,25 \\ &+ normalizarRetilineo(mediaLoCPorCondicao, 4) * 0,9 \\ &+ normalizarRetilineo(quantidadeExpressoesSimples, 5) * 0,5 \\ &+ normalizarRetilineo(mediaCondicoesSimples, 1) * 1\big) \end{aligned} \quad (4.4)$$

A Tabela 4.12 apresenta os valores dos parâmetros de cada indício presente na Função 4.4 e seus respectivos significados.

É relevante que existam valores iniciais para a execução da mecânica de busca para esta refatoração. Porém, esses valores para os pesos e parâmetros devem ser baseados em um estudo

Tabela 4.12 – Valores dos parâmetros dos indícios.

| **Indício** | **Valor do parâmetro** | **Significado** |
|---|---|---|
| Quantidade de condicionais | 5 | Prestigia as oportunidades com 5 ou mais sentenças condicionais. |
| Quantidade de linhas de código | 30 | Privilegia as oportunidades com 30 ou mais linhas no total. |
| Média de linhas por condição | 4 | Prestigia as oportunidades que possuam em média 4 ou mais linhas de código por condição. |
| Quantidade de expressões condicionais simples | 5 | Oportunidades com 5 ou mais expressões condicionais simples são favorecidas. |
| Média de condições simples | 1 | Beneficia oportunidades próximas a uma condição simples por sentença condicional. |

mais abrangente, que envolva mais projetos, e com validação de entidades externas, como outros desenvolvedores ou especialistas. O mesmo vale para a definição das oportunidades como aplicáveis ou não baseadas em um limiar. Tal valor de limiar precisa ser melhor aferido para que possa ser considerado um valor "padrão".

Os valores dos pesos indicam a relevância entre os valores dos indícios. Assim, quanto maior o valor do peso, maior será a relevância do indício para a Função Heurística 3.5. É recomendado utilizar o valor um para indicar que o indício é relevante, mas isso é de escolha do desenvolvedor. Além disso, o desenvolvedor pode optar por utilizar outros métodos de valoração dos pesos, como métodos de decisão multicritérios.

Como a função *normalizarRetilineo* é utilizada em todos os indícios e os resultados ao final são somados, quanto maior for o valor do indício, maior a chance de ser uma oportunidade de refatoração aplicável para Substituir Envio Condicional por *Command*.

A Tabela 4.12 apresenta o parâmetro da quantidade de condicionais, que recebeu o valor *cinco* pois um número abaixo dessa quantidade pode não valer a pena a aplicação da refatoração, e acima não há problema. Um intervalo de valores que pode ser aplicado à esse parâmetro vai de *três* a *15*. O peso desse indício para a Função 4.4 não foi relevante, logo recebeu o valor *0,1*.

Quanto maior a quantidade de linhas de código de todo o bloco condicional, mais chance de o bloco ser uma oportunidade de refatoração. Mas um número abaixo de *30* linhas de código pode apenas indicar um falso positivo. Esse valor de parâmetro pode variar de *20* até *100*, onde caso o número seja muito baixo, talvez não seja interessante aplicar a refatoração. A simples quantidade de linhas de código não contribui tanto para a descoberta de oportunidades de refatoração, assim, recebeu o peso *0,25*.

A média de linhas de código por condição é um indício importante para se aferir as oportunidades de refatoração. Dessa forma, foi utilizado o valor *0,9* para o peso. Para o valor do parâmetro foi utilizado o valor *quatro*, conforme apresenta a Tabela 4.12, onde um valor menor que esse provavelmente indique um falso positivo. Um intervalo de valores para esse parâmetro pode ser de *três* a *50*.

A Tabela 4.12 apresenta o *valor* cinco para o parâmetro do indício *quantidadeExpressoesSimples*. O valor do parâmetro desse indício e do indício *quantidadeCondicionais* estão ligados, pois é interessante que a quantidade de condições seja a mesma quantidade de expressões simples. Mas mesmo assim, quanto maior a quantidade de expressões simples condicionais presentes no bloco condicional, maior a chance de ser uma oportunidade. O intervalo segue a mesma recomendação do primeiro indício, sendo de *três* a *15*. Apenas a quantidade de expressões simples não é tão relevante, mas mesmo assim se faz presente, dessa forma, o peso recebeu o valor *0,5*.

Enfim, a média de condições simples se mostrou relevante para o cálculo das oportunidades, recebendo assim o peso *um*. O valor do parâmetro também recebeu esse valor, pois o valor mais próximo de *um* significa que a maioria ou todas as sentenças condicionais são simples. O intervalo para os valores de parâmetro são inferiores a *um*, sendo de *0,1* até *um*.

As próximas Seções descrevem os resultados da execução da mecânica para Substituir Envio Condicional por *Command* com os valores da Função 4.4. Para o projeto Apache-Ant, os valores dos pesos são diferentes, conforme os resultados obtidos durante as execuções.

### 4.3.1 ADempiere

A Tabela 4.13 apresenta os três primeiros resultados da execução da mecânica de busca por oportunidades de refatoração para Substituir Envio Condicional por *Command*, de um total de *236* oportunidades candidatas. Todas as três oportunidades foram avaliadas e consideradas de "muito alta" aplicabilidade.

Tabela 4.13 – Três primeiras candidatas a oportunidades de refatoração para Substituir Envio Condicional por *Command* do projeto ADempiere.

| # | Classe | Resultado | Aplicabilidade |
|---|---|---|---|
| 1 | Doc_InOut | 12,739999 | Muito alta |
| 2 | GridTab | 10,910000 | Muito alta |
| 3 | ExportHelper | 9,620000 | Muito alta |

A Listagem 4.5 apresenta o bloco condicional da primeira oportunidade candidata *Doc_*

*InOut*. O bloco possui 237 linhas de código, com cinco condicionais, resultando em uma média de 47,4 linhas por condição. Entretanto, as expressões condicionais não são consideradas simples, pois envolvem o uso de uma comparação de *Strings* e um valor booleano. Sendo assim, os dois indícios que fizeram esse bloco ficar na primeira posição é o número de linhas e a média de linhas por condição.

Listagem 4.5 – Oportunidade candidata 1 antes da refatoração.

```
1  public ArrayList<Fact> createFacts(MAcctSchema as) {
2    ...
3    // *** Sales - Shipment
4    if (getDocumentType().equals(DOCTYPE_MatShipment) && isSOTrx()) {
5      ...
6    // *** Sales - Return
7    } else if (getDocumentType().equals(DOCTYPE_MatReceipt) && isSOTrx()) {
8      ...
9    // *** Purchasing - Receipt
10   } else if (getDocumentType().equals(DOCTYPE_MatReceipt) && !isSOTrx()) {
11     ...
12   // *** Purchasing - Return
13   } else if (getDocumentType().equals(DOCTYPE_MatShipment) && !isSOTrx()) {
14     ...
15   // *** None of above
16   } else {
17     ...
18   }
19
20   facts.add(fact);
21   return facts;
22 }
```

Esse bloco monta as linhas de um documento de acordo com o tipo do documento (*getDocumentType*) e se o mesmo é uma transação de venda (*isSOTrx*). Caso não for nenhuma das quatro condições, é então enviado uma mensagem de exceção. Logo, cada condição representa uma ação diferente a ser realizada pela classe *Doc_InOut*. Sendo assim, essas características a tornam uma oportunidade de refatoração com uma aplicabilidade "muito alta".

A Figura 4.13 apresenta a estrutura das classes após a aplicação da refatoração. Foi criada uma classe de comando concreto para cada condição, e uma classe de comando abstrata *Handler* para definir a interface de comunicação entre o cliente *Doc_InOut* e as classes de comando. O construtor da classe *Handler* recebe a instância da classe *Doc_InOut* para que seja possível acessar os métodos e valores contidos nela. Além disso, o método *execute* recebe como parâmetro todas as variáveis necessárias para a construção do documento.

Para que fosse possível os *Handlers* concretos acessassem os valores das variáveis privadas, foi necessário publicizá-las através dos métodos *getters*. Foram criados também dois métodos, um para criar os comandos na classe *Doc_InOut* chamado *createHandlers*, e outro

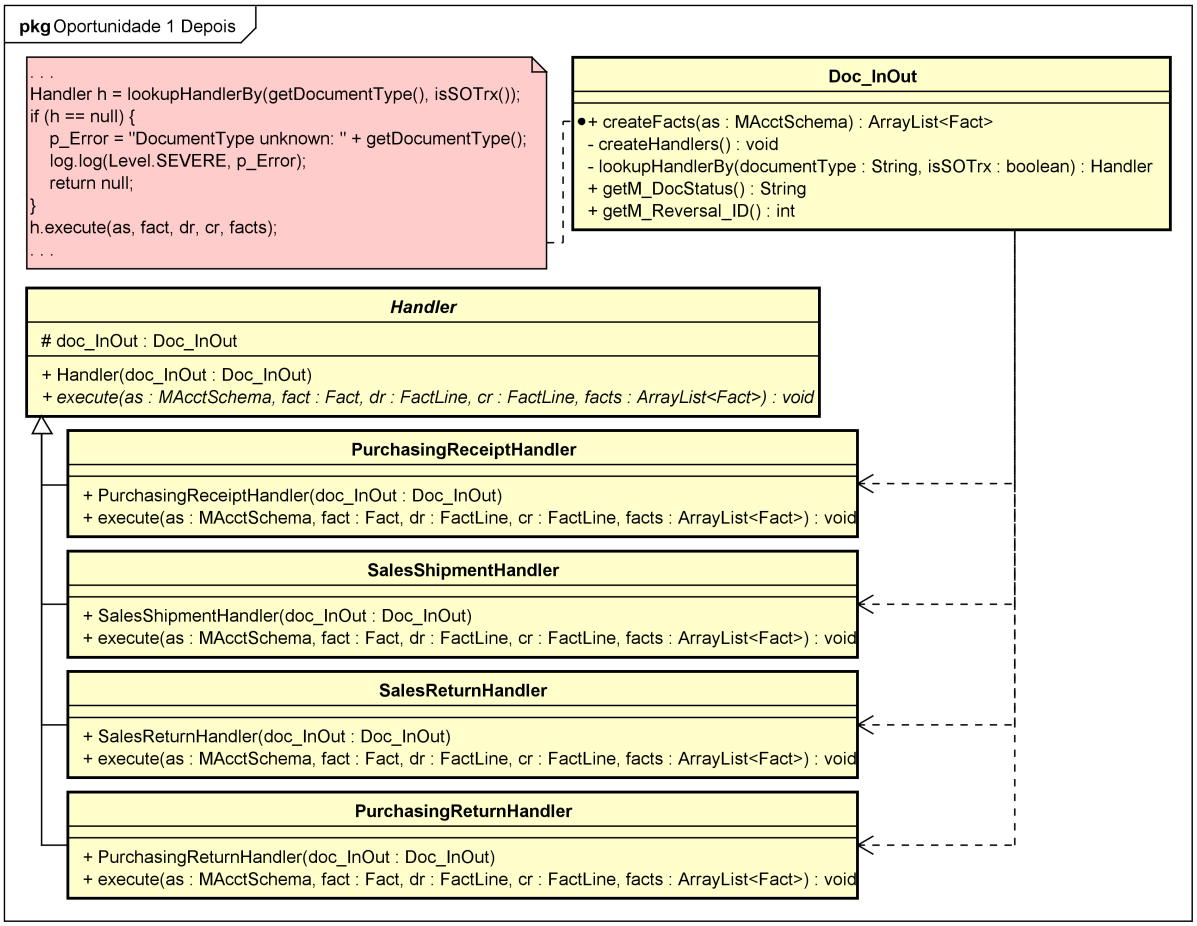


Figura 4.13 – Oportunidade candidata 1 depois da refatoração.

para buscá-los de acordo com o tipo do documento e a transação chamado *lookupHandlerBy*.

Após, o bloco condicional foi substituído por uma busca do *handler* adequado de acordo com o tipo do documento e se é uma transação, como pode ser observado na anotação da Figura 4.13. Caso não seja encontrado o comando concreto, será disparada uma exceção, assim como previsto no `else` do bloco condicional. Senão, o comando será executado, montando o documento de acordo com o seu tipo e sua operação. Assim, as linhas de código do bloco condicional foram substituídas de *237* a apenas *sete* dentro do método *createFacts*. Na classe *Doc_InOut* houve o acréscimo de *19* linhas. Já no projeto, se aumentou em *quatro* classes e *uma* classe abstrata.

A segunda oportunidade candidata a refatoração é o bloco condicional da classe *GridTab*, que contém 125 linhas de código espalhadas em três sentenças condicionais, com uma média de 41,66 linhas por condição. Das três condições, apenas uma foi considerada simples. O bloco condicional verifica o nome da tabela e retorna as informações de transação, onde para cada tabela realiza ações diferentes. Com essas características e como o bloco condicional

representa um roteamento de ações esta oportunidade tem uma aplicabilidade “muito alta”.

Para a aplicação da refatoração Substituir Envio Condicional por *Command* foi criada uma classe abstrata *Handler* para a ser a interface de comunicação do cliente *GridTab* com as classes concretas. Cada condição do bloco foi refatorada para uma subclasse de *Handler*. Em seguida, foi criada a instância de cada subclasse de *Handler* e armazenadas em uma mapa, sendo a chave a *String* específica de cada expressão condicional. Uma das expressões condicionais, além da *String*, é utilizado um valor inteiro. Assim, para esse caso, é verificado o valor da *String* e do número inteiro. Por fim, para uma variável privada da classe *GridTab* foi criado um método *get* para que fosse possível o seu acesso. Ao final, das *125* linhas do bloco condicional dentro do método *getTrxInfo* restaram apenas *duas*, já fora do método houve o aumento de *16* linhas, *três* classes concretas e *uma* classe abstrata.

A terceira oportunidade candidata para essa refatoração é o bloco condicional presente no método *generateExportFormat* da classe *ExportHelper*. É composto por 156 linhas de código em cinco sentenças condicionais, resultando numa média de 31,2 linhas por condição. Das cinco condições, quatro foram consideradas simples e uma é uma sentença `else`. Esse bloco faz o roteamento de solicitações de exportação para XML, que são realizadas ações diferentes dependendo do tipo do formato da linha. Dessa forma, essa oportunidade recebeu uma avaliação de aplicabilidade “muito alta”.

Depois de aplicada a refatoração, foram criadas classes para cada condição do bloco condicional, onde todas estendem da superclasse abstrata *Handler*. Um mapa com uma instância das subclasses de *Handler* foi adicionado na classe cliente *ExportHelper*. Os atributos privados da classe cliente que eram necessários serem visíveis para as classes de comando foram publicizados por métodos *getters* e *setters*. O bloco condicional então foi substituído por uma busca pelas subclasses de *Handler* no mapa. Caso nulo, ou seja, não havendo comando concreto específico para aquele tipo de formato de linha, é lançada uma exceção, comportamento que a sentença `else` do bloco original realizava. Por fim, é executado o método *execute* com os valores de parâmetros necessários para a realização da ação específica. Esta refatoração diminuiu o método *generateExportFormat* de *156* linhas de código para apenas *cinco*, enquanto fora do método se criou *16* linhas. Além disso, foram criadas *três* classes concretas e *uma* classe abstrata.

### 4.3.2 Apache-Ant

No projeto Apache-Ant os valores dos indícios das oportunidades encontradas são homogêneos. A Tabela 4.14 apresenta os maiores e menores valores obtidos na busca por oportunidades nesse projeto.

Tabela 4.14 – Maiores e menores valores obtidos de cada indício do projeto Apache-Ant.

| **Indício** | **Menor valor** | **Maior valor** |
|---|---|---|
| Número de condicionais | 3,0 | 4,0 |
| Número de linhas de código | 9,0 | 31,0 |
| Média de LOC por condição | 3,0 | 10,33 |
| Número de expressões simples | 0,0 | 3,0 |
| Média de condições simples | 0,0 | 1,0 |

Assim, foi necessário reajustar os valores dos pesos de cada indício para que fosse possível identificar as melhores oportunidades candidatas. Os valores de pesos para os indícios que trouxeram os melhores resultados foram:

- Peso do indício *quantidadeCondicionais*: *1,8* (*0,1* para os outros projetos);
- Peso do indício *quantidadeLoC*: *0,9* (*0,25* para os outros projetos);
- Peso do indício *mediaLoCPorCondicao*: *0,4* (*0,9* para os outros projetos);
- Peso do indício *quantidadeExpressoesSimples*: *0,3* (*0,5* para os outros projetos);
- Peso do indício *mediaCondicoesSimples*: *0,5* (*1,0* para os outros projetos).

Os pesos que sofreram maior mudança em relação aos outros projetos foram dos indícios *quantidadeCondicionais* e *quantidadeLoC*. De todas as oportunidades candidatas, apenas três possuem quatro condicionais, o restante das oportunidades possui somente três condições presente no bloco. A partir dessa caraterística, o peso da quantidade de condicionais em relação aos demais indícios se torna muito relevante.

Quanto maior o número de linhas, mais interessante pode ser ao desenvolvedor aplicar a refatoração para esse padrão de projeto. Dada a homogeneidade dos valores, o peso do indício *quantidadeLoC* recebeu o valor *0,9*.

A execução com os valores de pesos definidos anteriormente resultou em *23* oportunidades candidatas, onde a primeira foi considerada com uma aplicabilidade “muito alta”, a segunda como “média” e a terceira como “baixa” apresentadas na Tabela 4.15.

Tabela 4.15 – Três primeiras candidatas a oportunidades de refatoração para Substituir Envio Condicional por *Command* do projeto Apache-Ant.

| # | Classe | Resultado | Aplicabilidade |
|---|---|---|---|
| 1 | GenericDeploymentTool | 3,0900000 | Muito alta |
| 2 | Jar | 3,0433333 | Média |
| 3 | CvsVersion | 2,9850000 | Baixa |

A Listagem 4.6 apresenta o método *getJarBaseName*, onde está localizado o bloco condicional referente a primeira oportunidade candidata. O bloco possui quatro sentenças condicionais, onde nenhuma foi considerada simples pela mecânica. Ao todo o bloco possui 30 linhas de código, com uma média de 7,5 linhas por condição. O objetivo desse bloco condicional é obter o *baseName* de acordo com o tipo do *NamingScheme*, assim, realizando ações diferentes para cada tipo. Dessa maneira, essa candidata foi avaliada e classificada com uma aplicabilidade "muito alta".

Listagem 4.6 – Oportunidade candidata 1 antes da refatoração.

```
1  protected String getJarBaseName(String descriptorFileName) {
2    String baseName = "";
3    if (EjbJar.NamingScheme.BASEJARNAME.equals(
         config.namingScheme.getValue())) {
4      ...
5    } else if (EjbJar.NamingScheme.DESCRIPTOR.equals(
         config.namingScheme.getValue())) {
6      ...
7    } else if (EjbJar.NamingScheme.DIRECTORY.equals(
         config.namingScheme.getValue())) {
8      ...
9    } else if (EjbJar.NamingScheme.EJB_NAME.equals(
         config.namingScheme.getValue())) {
10     ...
11   }
12   return baseName;
13 }
```

Após a aplicação da refatoração foram criadas as classes de comando para cada `if` do bloco condicional. A Figura 4.14 apresenta essa estrutura de classes, onde todas as subclasses de *Handler* implementam o método *execute* recebendo a *String descriptorFileName* como parâmetro. A instância de *GenericDeploymentTool* é armazenada em cada subclasse para que seus valores de atributos possam ser acessados para obter o *baseName*. Além disso, o atributo *handler* teve de ser publicizado com o método *getHandler* para que fosse possível o seu acesso pelas subclasses de *Handler*.

Após, foi criado o método *createHandlers* para realizar a instanciação de cada subclasse de *Handler* associando com sua respectiva chave (`BASEJARNAME`, por exemplo). Por fim, o

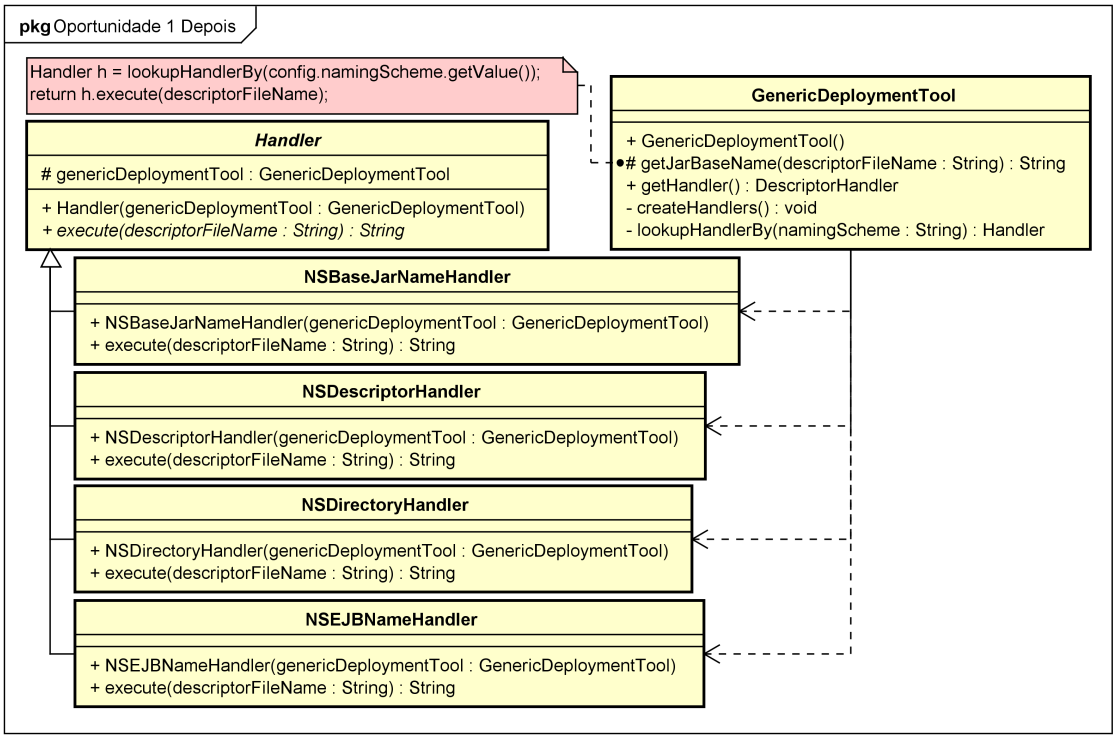


Figura 4.14 – Oportunidade candidata 1 depois da refatoração.

bloco condicional foi removido e inseridas as *duas* linhas para obter o comando concreto e executar o método *execute*. Fora do método foram adicionadas *12* linhas de código, *quatro* classes concretas e *uma* classe abstrata.

A segunda oportunidade candidata é o bloco condicional do método *filesetManifest* presente na classe *Jar*. O bloco possui um total de 31 linhas de código, com uma média de 10,3 linhas por condição, contendo três sentenças condicionais. Entretanto, as sentenças condicionais não possuem nenhuma expressão simples. O objetivo do bloco é verificar e obter o *Manifest*, onde a primeira condição compara o atributo *manifestFile* com o *file* enviado por parâmetro e a segunda condição verifica se é uma configuração para não ignorar (*skip*). A terceira condição é apenas um `else` sem nenhuma linha de código, com alguns comentários que caso entre nessa condição deve ser ignorada (*skipped*) a obtenção do *Manifest*.

As três condições podem ser consideradas comandos distintos, até mesmo o ato de ignorar a construção pode ser considerado como um comando. Entretanto, como a verificação é realizada sobre variáveis diferentes, não é possível utilizar um mapa de *handlers* para a obtenção do comando concreto, necessitando assim a presença das sentenças condicionais. Dessa forma, essa oportunidade candidata foi considerada com uma aplicabilidade “média”.

Foi aplicada parte da refatoração, isolando o conteúdo de cada condição em um comando

concreto distinto. Foi criada a classe abstrata *Handler* que se tornou a superclasse dos comandos concretos. Alguns atributos e métodos foram publicizados dentro do pacote para que pudessem ser utilizados pelas classes de comando. Então, o conteúdo de cada condição foi substituído pela chamada à classe comando concreta respectiva, mantendo-se assim as sentenças condicionais do bloco. Dessa forma, o bloco foi reduzido de *31* para *seis* linhas de código dentro do método. Fora do método não houve adição de código, mas apenas a adição das *três* classes concretas e *uma* classe abstrata.

A última candidata é o bloco condicional da classe *CvsVersion* que está no método *execute*. Ele possui uma média de 4,5 linhas de código em quatro sentenças condicionais, totalizando 18 linhas de código. Das quatro condições, três foram consideradas expressões simples. Entretanto, duas condições são formadas por duas expressões simples e as outras duas são formadas por cinco expressões. Isso impossibilita o uso de uma busca por chave/valor utilizando um mapa. Assim, a candidata a oportunidade de refatoração recebeu uma avaliação "baixa" de aplicabilidade.

Assim como a oportunidade anterior, a refatoração foi aplicada em parte. Foi criada a superclasse *Handler* e as linhas de código dentro de cada condição foram inseridas em seu respectivo comando concreto. O método *execute* recebeu por parâmetro todas as variáveis necessárias para a execução de cada comando concreto. Os métodos para atribuir valores aos atributos *serverVersion* e *clientVersion* foram criados para que as subclasses de *Handler* pudessem acessá-los. Por fim, as linhas internas do bloco condicional foram substituídas pelas chamadas às classes concretas, executando o método abstrato *execute*. Com isso, a quantidade total de linhas de código do bloco candidato foi reduzida para *oito* linhas. Dentro da classe mas fora do método não foram realizadas modificações. Por fim, foram criadas *quatro* classes concretas e *uma* classe abstrata.

### 4.3.3 EclipseLink

Um total de *177* oportunidades candidatas para a refatoração Substituir Envio Condicional por *Command* foram identificadas após a execução da mecânica sobre o projeto EclipseLink. A Tabela 4.16 apresenta as três oportunidades com maior valor para o resultado. A primeira e terceira oportunidades foram consideradas de aplicabilidade "alta", enquanto a segunda foi considerada com "média" aplicabilidade.

A candidata a oportunidade de refatoração *CriteriaQueryImpl* possui uma média de 28

Tabela 4.16 – Três primeiras candidatas a oportunidades de refatoração para Substituir Envio Condicional por *Command* do projeto EclipseLink.

| # | Classe | Resultado | Aplicabilidade |
|---|---|---|---|
| 1 | CriteriaQueryImpl | 7,0600000 | Alta |
| 2 | JAXBMarshaller | 5,8083333 | Média |
| 3 | SDODataHelper | 5,5266666 | Alta |

linhas de código por condição, formada por 84 linhas de código no total distribuída em três expressões condicionais, sendo um `else`. As duas condições são comparações do valor de uma variável com Enumerações de *ResultType*, não sendo assim considerados expressões simples. A Listagem 4.7 apresenta o bloco condicional antes da aplicação da refatoração.

Listagem 4.7 – Oportunidade candidata 1 antes da refatoração.

```
1  protected ObjectLevelReadQuery createSimpleQuery() {
2    ...
3    if (this.queryResult == ResultType.UNKNOWN) {
4      ...
5    } else if (this.queryResult.equals(ResultType.ENTITY)) {
6      ...
7    } else {
8      ...
9    }
10   ...
11 }
```

De acordo com o valor da variável *queryResult* é executada uma sequência de passos diferentes para a criação de uma instrução de consulta no banco de dados. Cada sequência de passos pode ser considerada uma ação distinta que será realizada pelo método *createSimpleQuery*. Entretanto, a quantidade de condições no bloco é baixa. Assim, essa oportunidade candidata foi considerada de aplicabilidade “alta”.

Para a aplicação da refatoração Substituir Envio Condicional por *Command* foram criadas as classes *UnknownResultHandler*, *EntityResultHandler* e *ElseHandler*, sendo todas subclasses da classe abstrata *Handler*. Cada condição do bloco foi movida para dentro do método *execute* da respectiva classe concreta. Para que as classes concretas acessassem os atributos e métodos da classe *CriteriaQueryImpl*, a instância da mesma é enviada como parâmetro no método construtor de *Handler*. A Figura 4.15 apresenta a estrutura de classes e seus relacionamentos depois da refatoração ser aplicada.

O objetivo do bloco condicional é montar uma consulta de acordo com o valor do atributo *queryResult*. Essa consulta é montada no objeto *query* da classe *ObjectLevelReadQuery*. Portanto, o objeto *query* deve ser enviado por parâmetro para que os comandos concretos reali-

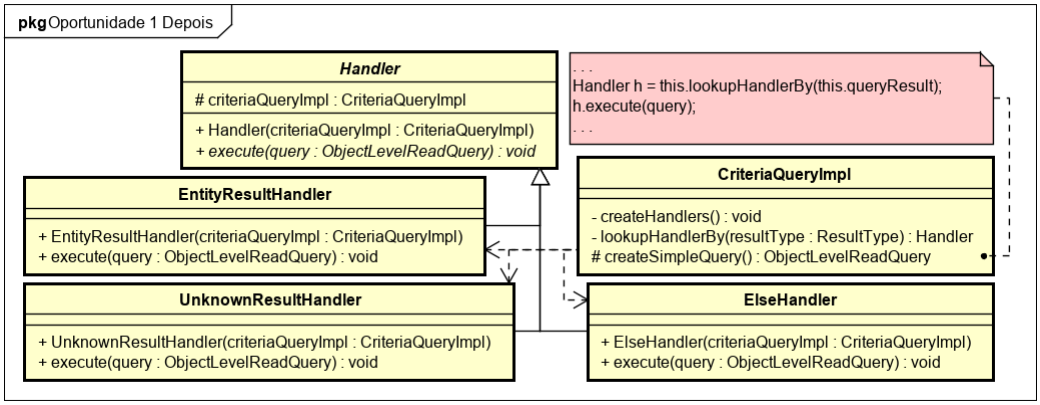


Figura 4.15 – Oportunidade candidata 1 depois da refatoração.

zem a montagem da consulta. Após a execução de um dos comandos concretos, o objeto *query* é retornado para o cliente que invocou o método *createSimpleQuery*.

Foram criados na classe *CriteriaQueryImpl* os métodos *createHandlers* e *lookupHandlerBy* que instancia os comandos concretos e busca pelo comando, respectivamente. Em seguida, o bloco condicional é substituído pela busca e execução do comando específico, como a anotação da Figura 4.15 apresenta. O resultado foi a redução de *84* linhas de código para apenas *duas* no método *createSimpleQuery*. Na classe *CriteriaQueryImpl* houve o acréscimo de *13* linhas, e fora da classe se criou *três* classes de comando concretas e *uma* classe abstrata.

A segunda candidata a oportunidade de refatoração é o bloco condicional pertencente a *JAXBMarshaller*, composto por 110 linhas de código distribuído em 30 instruções condicionais, resultando em 3,6 linhas por condicional em média. O bloco verifica e executa uma ação de acordo com o valor de uma chave que é passada por parâmetro. Por essa característica, 86% (26 das 30) condições foram consideradas simples. Dessa maneira, a oportunidade candidata foi considerada “média”.

A refatoração foi aplicada ao bloco condicional, substituindo-se as condições por instâncias de classes criadas para cada respectivo `if`. Tais classes herdam da superclasse abstrata *Handler*, que recebe a instância de *JAXBMarshaller* em seu construtor. Foram criados *setters* para quatro atributos da classe *JAXBMarshaller*, para que fosse possível o acesso pelas subclasses de *Handler*. Em seguida, a visibilidade da classe interna privada foi modificada para ser acessível pelas outras classes. Foi criado um *Map* para armazenar as chaves (condição de cada `if`) e os valores (instância de cada subclasse concreta referente ao respectivo trecho dentro do `if`). Ao final, o bloco condicional foi reduzido de *110* para *nove* linhas de código. Já na classe *JAXBMarshaller* foram adicionadas *42* linhas, e fora dela mais *28* classes de comando concretas

e *uma* classe abstrata.

O bloco condicional do método *convertToStringValue* da classe *SDODataHelper* ficou como terceiro candidato a oportunidade de refatoração. Composto de 50 linhas de código, o bloco condicional possui três sentenças condicionais, todas consideradas simples, resultando em 16,6 linhas por condição. Entretanto, a verificação de uma sentença condicional difere das outras. As duas primeiras sentenças verificam o tipo da classe da variável *value*. A última faz a verificação pelo nome da classe da variável *value*. Dada essas características, esta oportunidade candidata foi considerada de aplicabilidade "alta".

Para a aplicação da refatoração, foi desenvolvida a hierarquia de classes partindo da classe abstrata *Handler*. Suas subclasses *CalendarHandler*, *DateHandler* e *JavaxDateHandler* implementam o trecho de código respectivo de cada sentença condicional do bloco. O método *execute* recebe como parâmetros as mesmas variáveis do método *convertToStringValue*, necessárias para a realização do comportamento das classes. Além disso, o método *getXMLConversionManager* foi publicizado a nível do pacote, para que a classe *JavaxDateHandler* pudesse utilizar para a conversão do objeto em uma *String*.

O restante da refatoração foi aplicada na classe *SDODataHelper*. Se desenvolveu o método *createHandlers* que cria um *Map* para o armazenamento das instâncias dos comandos concretos, ou seja, das subclasses de *Handler*. Tal método foi utilizado nos dois construtores da classe *SDODataHelper*. Já o método *lookupHandlerBy* verifica o tipo da classe de *value* e retorna o comando concreto correspondente. Por fim, o bloco candidato a oportunidade de refatoração é substituído pela chamada ao método *lookupHandlerBy*, e chamado o comando *execute*. Como resultado, há a redução de *50* linhas de código para apenas *seis* linhas no método *convertToStringValue*. Fora deste método há a adição de 13 linhas, e *quatro* classes fora da desta classe, onde *uma* delas é a classe abstrata *Handler*.

### 4.3.4 *Pattern Refactoring*

A Tabela 4.17 apresenta os três primeiros resultados das *cinco* oportunidades candidatas para a aplicar a refatoração Substituir Envio Condicional por *Command*. As oportunidades foram consideradas com as aplicabilidades "alta", "média" e "muito baixa" respectivamente.

A Listagem 4.8 apresenta a primeira candidata a oportunidade de refatoração: o bloco condicional pertencente ao método *doubleClick* da classe *DoubleClickListener*. Este bloco possui 17 linhas de código e três sentenças condicionais simples, resultando em 5,6 linhas por

Tabela 4.17 – Três primeiras candidatas a oportunidades de refatoração para Substituir Envio Condicional por *Command* do projeto *Pattern Refactoring*.

| # | Classe | Resultado | Aplicabilidade |
|---|---|---|---|
| 1 | DoubleClickListener | 2,7766666 | Alta |
| 2 | EncapsulateClassWithFactory | 2,6100000 | Média |
| 3 | ReplaceConditionalDispatcherWithCommand | 2,2279166 | Muito baixa |

condição em média. O método *doubleClick* trata eventos de clique duplo sobre os itens das listas exibidas na visão do AOPJungle e *Pattern Refactoring*. Caso o item inicie em uma linha dentro de uma classe, será exibida a linha correspondente da oportunidade de refatoração. Se o item for uma classe, o cursor será colocado na linha de criação da classe. Com essas características, a oportunidade foi considerada com uma aplicabilidade "alta".

Listagem 4.8 – Oportunidade candidata 1 antes da refatoração.

```
1  public void doubleClick(DoubleClickEvent event) {
2    ...
3    if (selectedNode instanceof StartLinePosition) {
4      ...
5    } else if (selectedNode instanceof TreeElement) {
6      ...
7    } else if (selectedNode instanceof Type)
8      clazz = ((Type) selectedNode).getAojTypeDeclaration();
9    ...
10 }
```

A verificação do tipo do objeto *selectedNode* é feita com o operador `instanceof`. Se *selectedNode* em algum momento ser uma instância da classe *TreeAttribute*, subclasse de *StartLinePosition*, a primeira condição será verdadeira. Com a aplicação da refatoração atual, será criado um *HashMap* que contém a instância de cada comando presente dentro de cada `if`. A forma de busca que a classe *HashMap* realiza em seu método *get* utiliza o método *equals*, que pertence à classe *Object*. Em outras palavras, será realizada a comparação do nome da classe a qual o objeto *selectedNode* é instância com o nome da classe sendo comparada (no exemplo *StartLinePosition*). Assim, se *selectedNode* em algum momento ser uma instância da classe *TreeAttribute*, utilizando a refatoração Substituir Envio Condicional por *Command*, a primeira condição será falsa, não mais possuindo o mesmo comportamento externo observável.

Uma maneira de resolver esse impasse é tornar a classe *StartLinePosition* `final`, ou seja, não podem ser derivadas subclasses. Assim, é garantida que as instancias *selectedNode* serão sempre da classe *StartLinePosition*. Essa solução não é aplicável ao projeto *Pattern Refactoring*, pois as subclasses derivadas de *StartLinePosition* possuem sentido em sua existência.

Outra forma de resolução é, no momento de definição de cada *Handler*, associar a classe

*StartLinePosition* e cada uma de suas subclasses ao objeto de comando concreto. Esta solução foi adotada para o projeto. Entretanto, a cada nova subclasse criada precisa ser adicionada no mapa de *Handlers*.

A Figura 4.16 apresenta a estrutura de classes e relacionamentos após a aplicação da refatoração. Foi criada a classe abstrata *Handler* com o método *execute*, e derivada dela foram criadas as subclasses *StartLinePositionHandler*, *TreeElementHandler* e *TypeHandler*. O conteúdo de cada sentença condicional foi movido para o método *execute* da respectiva classe.

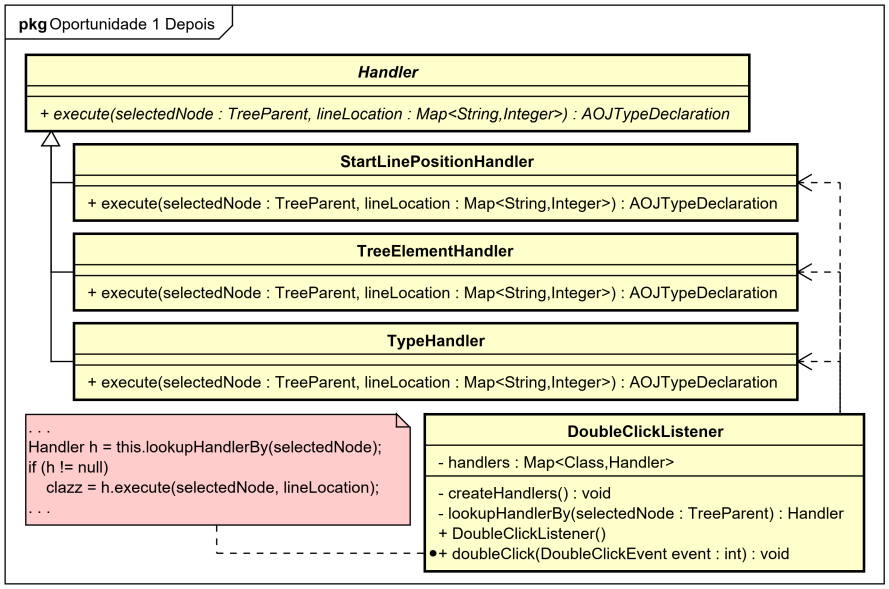


Figura 4.16 – Oportunidade candidata 1 depois da refatoração.

Na classe *DoubleClickListener* foram criados os métodos *createHandlers*, *lookupHandlerBy* e o método construtor da classe, que chama o método *createHandlers* inicializando o mapa de handlers. Por fim, o bloco condicional foi substituído pela chamada ao método *lookupHandlerBy* passando a variável *selectedNode* como parâmetro para que seja verificado e retornado qual a respectiva instância de *Handler* a ser executada. Como resultado se tem a redução de *17* para *três* linhas de código dentro do método *doubleClick*. Dentro desta classe foram adicionadas *18* linhas, e fora dela foram criadas *três* classes concretas e *uma* classe abstrata.

O desenvolvedor decidiu não manter a refatoração dessa oportunidade no projeto *Pattern Refactoring*. A hierarquia de classes de *StartLinePosition*, *TreeElement* e *Type* fazem parte da hierarquia relatada nas três oportunidades de refatoração Encapsular Classes com *Factory* apresentados na Seção 4.1.4. Logo, é necessária uma modificação na hierarquia de classes, seja em sua visibilidade, ou instanciação, para diminuir o acoplamento do projeto. Em seguida, realizar uma nova execução desta mecânica e verificar se ainda haverá a presença de uma oportunidade

de refatoração para Substituir Envio Condicional por *Command*.

A segunda candidata a oportunidade de refatoração é o bloco condicional que encontra-se dentro do método *searchInStatements* da classe *EncapsulateClassWithFactory*. O método tenta encontrar os tipos das classes através da palavra-chave "`new`". Neste contexto, o bloco condicional tem o objetivo de verificar se a expressão é uma invocação de um método, uma instanciação de uma classe ou uma atribuição. Dependendo de cada expressão, é realizada uma forma diferente de se obter o tipo da classe.

Esta oportunidade é composta por 15 linhas de código em três condições, todas elas simples, resultando em cinco linhas de código em média. Assim como a primeira oportunidade, as sentenças condicionais utilizam o operador `instanceof`. A solução utilizada foi a mesma para essa oportunidade. Como o tamanho do bloco foi considerado pequeno, a aplicação da refatoração não contribuirá muito para o projeto. Dada essas características, essa oportunidade recebeu uma aplicabilidade "média".

A refatoração foi aplicada nesta oportunidade, criando-se a classe abstrata *Handler* como interface de comunicação da classe *EncapsulateClassWithFactory* com as classes concretas *AOJMethodInvocationHandler*, *AOJAssignmentHandler* e *AOJClassInstanceCreationHandler*. Em seguida, cada condição do bloco condicional foi refatorada para sua respectiva subclasse de *Handler*. Na classe *EncapsulateClassWithFactory* foi criado um mapa contendo as instâncias das classes concretas, e também codificado um método para a obter a instância de comando de acordo com o nome da classe. Para que funcionasse corretamente, foi criado um método *getter* para o atributo *requiredClazzes* e alterado o modificador de publicidade do método *addIfNotContains*, ambos da classe *EncapsulateClassWithFactory*. Por fim, foi adicionado o código para a busca do *Handler* no lugar do bloco condicional desta oportunidade de refatoração. Como resultado, houve a redução de *15* linhas para *três* linhas de código no método *searchInStatements*. Fora deste método houve a adição 14 linhas de código, e implementadas *três* classes concretas e *uma* classe abstrata fora da classe *EncapsulateClassWithFactory*.

O desenvolvedor também preferiu não aplicar a refatoração na segunda oportunidade. Os benefícios que a aplicação da refatoração trariam ainda não justificam a alteração. Caso aumente a quantidade de condições neste bloco condicional e incremente o número de linhas, então será aplicada a refatoração e gozado de seus benefícios.

Por fim, a terceira oportunidade de refatoração é o bloco condicional do método *identifyType* da classe *ReplaceConditionalDispatcherWithCommand*. Ele possui quatro condições,

sendo a última uma condição `else`. As três verificações realizadas por este bloco condicional candidato utilizam o operador `instanceof`. A forma de aplicar a refatoração nesta oportunidade foi a mesma que a primeira e segunda oportunidades. Ao total o bloco possui 17 linhas, uma média de 4,25 por sentença condicional.

O objetivo do método é identificar o tipo de uma variável presente em um método. O bloco condicional retorna o tipo da variável caso ela seja criada em um *EnhancedForStatement* ou *ForStatement*. A forma de realizar a busca é de dentro para fora do método, "subindo" os nós da AST até identificar o tipo da variável ou até chegar na declaração do método. Nesse último caso, é realizada a verificação do tipo da variável fora do método, como em um atributo por exemplo.

Há dentro de cada condição do bloco um `return` com o tipo da variável detectado ou `null` quando for uma declaração de um método. Em certos casos, uma condição `if` é satisfeita, mas o tipo da variável não é detectado (é um comportamento esperado). Nesses casos, a execução do *Pattern Refactoring* sai do bloco condicional e continua executando o método *identifyType*. Com a aplicação da refatoração, é necessário se estabelecer uma condição para cada retorno de cada comando concreto, para que a execução retorne `null` quando uma declaração de método, retorne o tipo da variável quando detectado, e continue a execução do método *identifyType* quando não encontrado. Assim, o bloco condicional é substituído por outro bloco condicional desta vez com três condições, sendo portanto uma oportunidade de refatoração com aplicabilidade "baixa".

A forma de aplicar a refatoração para esta oportunidade candidata foi retornar uma *String* com um valor específico quando não detectar o tipo da variável e não for um retorno nulo. Dessa forma, é garantido o mesmo comportamento externo observável do resultado do método. Após a aplicação, o bloco foi reduzido de *17* para *14* linhas de código. Além disso, foram adicionadas *11* linhas na classe *ReplaceConditionalDispatcherWithCommand*, mais *três* classes concretas e *uma* classe abstrata. Dada essas características pós-aplicação, o desenvolvedor decidiu não aplicar em permanente a refatoração no projeto *Pattern Refactoring*.

### 4.3.5 Avaliação dos resultados para a mecânica Substituir Envio Condicional por *Command*

A Figura 4.17 apresenta a quantidade total de oportunidades de refatoração identificadas em cada projeto. Em primeiro lugar vem o projeto ADempiere com *236* oportunidades, acompanhado pelo EclipseLink com *177*. O projeto Apache-Ant, com valores de pesos diferentes dos

outros projetos, teve *23* oportunidades encontradas. Enfim, o projeto *Pattern Refactoring* obteve *cinco* oportunidades candidatas após a execução da mecânica. Os resultados representam que, em média, a cada *2744* linhas de código uma oportunidade de refatoração para Substituir Envio Condicional por *Command* é encontrada. O projeto com menor quantidade de oportunidades por linha é o Apache-Ant, onde é necessário *3635* linhas de código em média para ser identificada uma oportunidade de refatoração. Já com *2282*, o EclipseLink é o projeto com maior quantidade de oportunidade per LoC.

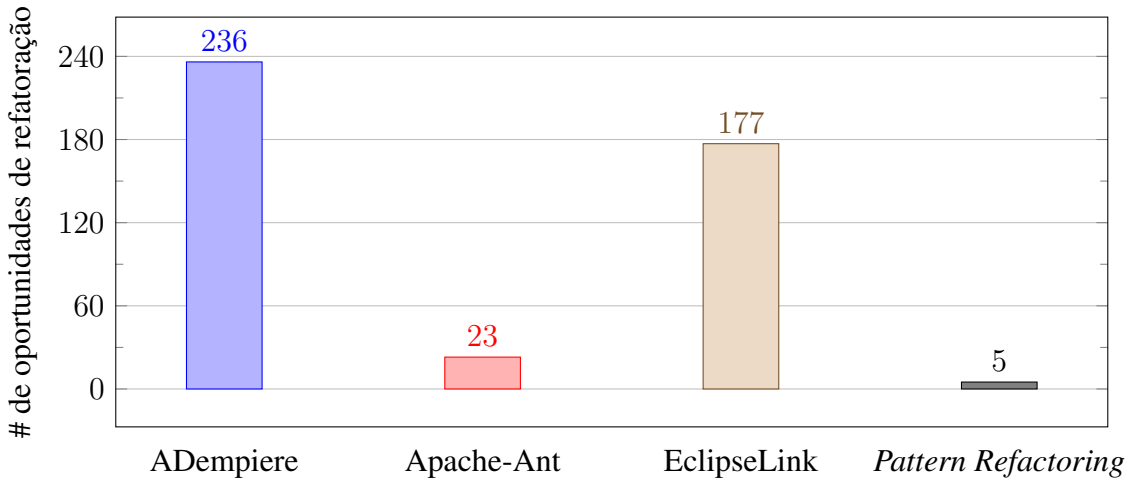


Figura 4.17 – Total de candidatas a oportunidades de refatoração encontradas para Substituir Envio Condicional por *Command*.

No projeto Apache-Ant foi necessário realizar uma readequação dos valores dos pesos para os indícios da Função 4.4 para que houvesse uma melhor ordenação das oportunidades candidatas. Isso ocorreu devido a homogeneidade das candidatas encontradas no projeto. Uma parcela considerável delas são oportunidades com aplicabilidade "média" a "muito baixa". Assim, a mudança dos valores dos pesos em busca de privilegiar as oportunidades reais teve de ser grande.

Não são todos os casos que é possível aplicar esta refatoração na presença do operador `instanceof`, apesar dele ser considerado um tipo simples de verificação. Isso ocorre pois uma verificação de tipo de instância não é equivalente a uma verificação de igualdade. É possível no entanto realizá-la, mas se faz necessária verificar se cada classe utilizada não pode ser derivada (com a palavra-chave `final`), ou se todas as suas subclasses fazem parte das verificações do bloco condicional candidato.

Uma forma de mitigar as oportunidades candidatas com a presença do operador `instanceof` é adicionar um indício à Função Heurística 3.5 representando a presença do opera-

dor, e atribuir um valor negativo à ele. Assim, se privilegiará as oportunidades sem a presença desse operador. Esse tipo de cálculo negativo já acontece com Função Heurística 3.1 da mecânica de busca para Encapsular Classes com *Factory*, com o indício $visitor$, por exemplo.

A Tabela 4.18 apresenta as dez primeiras candidatas a oportunidades utilizando a Função 4.4 para os projetos ADempiere, EclipseLink e *Pattern Refactoring*, e a mesma função, mas com valores de pesos diferentes para o projeto Apache-Ant, conforme a Seção 4.3.2 apresenta. *Nove* oportunidades pertencem ao projeto ADempiere e apenas a oportunidade da sexta colocação pertence ao projeto EclipseLink.

Tabela 4.18 – Dez primeiras candidatas a oportunidades de refatoração para Substituir Envio Condicional por *Command* dos quatro projetos.

| # | # no projeto | Classe | Resultado | Projeto |
|---|---|---|---|---|
| 1 | 1 | Doc_InOut | 12,739999 | ADempiere |
| 2 | 2 | GridTab | 10,910000 | ADempiere |
| 3 | 3 | ExportHelper | 9,620000 | ADempiere |
| 4 | 4 | VHRActionNotice | 8,605000 | ADempiere |
| 5 | 5 | MMeasure | 7,830000 | ADempiere |
| 6 | 1 | CriteriaQueryImpl | 7,060000 | EclipseLink |
| 7 | 6 | SmjPdfReport | 6,868333 | ADempiere |
| 8 | 7 | OrderReceiptIssue | 6,576666 | ADempiere |
| 9 | 8 | Export | 6,296666 | ADempiere |
| 10 | 9 | WPOSActionMenu | 5,915833 | ADempiere |

A primeira oportunidade do projeto Apache-Ant está localizada na *76ª* posição. Enquanto a primeira oportunidade do projeto *Pattern Refactoring* ficou na *105ª* posição. A maioria dos resultados melhores colocados pertencem ao projeto ADempiere.

Esta mecânica de busca para Substituir Envio Condicional por *Command* foi avaliada em projetos diferentes, e foi possível verificar a presença de oportunidades de refatoração com aplicabilidade “alta” e “muito alta”. Assim, a mecânica auxilia a identificar oportunidades de refatoração aplicáveis, possibilitando a melhoria na qualidade do código após a aplicação pelo desenvolvedor.

Mas há melhorias que podem ser realizadas nesta mecânica. Uma melhoria é verificar se em cada sentença condicional do bloco candidato há a presença da mesma variável sendo testada. Isso significa que há um roteamento de informações baseado no valor dessa variável. Nesse tipo de roteamento pode haver uma oportunidade de refatoração para *Command*. Outro indício é a presença do método *equals*, pois a forma de busca dos objetos de comandos concretos no mapa é feita de maneira semelhante, assim, não é necessário nenhuma alteração mais

abrangente no código-fonte.

Outra melhoria seria a sugestão da refatoração de algumas das sentenças condicionais do bloco, e não necessariamente o bloco condicional como um todo. Para isso, cada sentença deveria ser avaliada, calculada e ordenada. Assim, o bloco com o somatório dos resultados das sentenças condicionais maiores podem representar melhores oportunidades de refatoração.

Em alguns casos, o desenvolvedor do projeto já realizou o primeiro passo da desta refatoração, que é extrair o trecho de código de cada condição e transformá-los em métodos separados. Quando um bloco condicional já recebeu essa extração, a média de linhas de código por condição é reduzida, ficando em alguns casos abaixo de três linhas. Assim, a mecânica criada para essa refatoração não busca por oportunidades nessas condições.

Por fim, uma característica intrínseca da aplicação da refatoração Substituir Envio Condicional por *Command* é a presença de uma hierarquia de classes, que há a dependência de um cliente à elas. Indícios esses que são buscados na primeira refatoração deste trabalho, Encapsular Classes com *Factory*. Ou seja, após a aplicação da refatoração Substituir Envio Condicional por *Command* de cada oportunidade, poderia ser aplicada a refatoração Encapsular Classes com *Factory* na hierarquia de classes resultante de *Handler*, protegendo os comandos concretos, diminuindo o acoplamento entre as classes e publicizando apenas a interface de *Handler* que é comum a todos e utilizado pela classe cliente.

## 4.4 Ameaças à Validade

Uma ameaça externa do trabalho realizado é o limitado número de projetos testados, assim, as conclusões extraídas da avaliação podem não ser generalizadas para um maior grupo de aplicações. Uma forma de mitigar essa ameaça é aumentar conjunto de projetos, contemplando softwares de outras áreas de aplicação e número de linhas de código.

Em relação a validade interna as oportunidades de refatoração identificadas pelas mecânicas foram avaliadas manualmente pelos autores. Além disso, os desenvolvedores reais dos projetos ADempiere, Apache-Ant e EclipseLink não estão envolvidos com esse trabalho, sendo assim, não é possível saber se os mesmos utilizariam tais oportunidades encontradas.

Além disso, as buscas das mecânicas só são realizadas nas classes que pertencem aos projetos. Assim, classes externas, como de uma biblioteca referenciada, não podem ser verificadas em conjunto com as classes do projeto.

Quando as oportunidades resultantes das mecânicas possuem uma grande homogenei-

dade nos valores de seus indícios, é necessário refinar e/ou readequar os valores dos parâmetros e pesos. O desenvolvedor pode então realizar essas modificações e reexecutar as mecânicas.

## 4.5 Implementação

A implementação das mecânicas propostas neste trabalho foi realizada com o uso do *framework* AOPJungle (DE FAVERI, 2013). O *framework* AOPJungle foi desenvolvido para fornecer metainformações sobre código-fonte orientado a objetos e orientado a aspectos. Após o armazenamento das metainformações de programas escritos em Java e AspectJ é possível receber e executar consultas. Sua implementação foi realizada como um *plug-in* da plataforma Eclipse, que possui uma visão para o carregamento e visualização das metainformações dos projetos abertos na IDE Eclipse.

Para realizar a busca no código-fonte é necessário realizar a extração das informações disponíveis nas *Abstract Syntax Trees* (ASTs) das unidades de compilação (UC) do projeto. Esse processo é realizado através da implementação do padrão *Visitor*, onde cada método *visit* acessa um nó específico de uma AST. Cada método *visit* armazena as informações do nó visitado em um metamodelo orientado a objetos, que posteriormente é utilizado para a consulta das informações armazenadas.

Para o desenvolvimento das mecânicas propostas neste trabalho foi necessário o incremento do metamodelo criado por De Faveri (DE FAVERI, 2013) e estendido por Teixeira Júnior (TEIXEIRA JÚNIOR, 2014) para atender as novas informações que serão necessárias. A Figura 4.18 apresenta um recorte do metamodelo do AOPJungle. As classes em amarelo são originais do trabalho de De Faveri (DE FAVERI, 2013), enquanto as classes em azul claro foram adicionadas ou modificadas no trabalho de Teixeira Júnior (TEIXEIRA JÚNIOR, 2014). As classes adicionadas ou modificadas no decorrer deste trabalho se encontram na cor verde. Através desse metamodelo as metainformações do código-fonte dos projetos são extraídas e carregadas para a utilização do *plug-in*.

O *plug-in* AOPJungle foi estendido, adicionando uma nova visão chamada *Pattern Refactoring*. Para a execução dessa nova visão, primeiro se executa o AOPJungle, onde se selecionam os projetos desejados para carregar. Após o processo de carregamento do metamodelo, se executa o *plug-in Pattern Refactoring*, que busca no metamodelo pelas informações necessárias para a execução das mecânicas de busca por oportunidades de refatoração desenvolvidas neste trabalho. Cada mecânica de busca possui uma aba de com os resultados da sua execução.

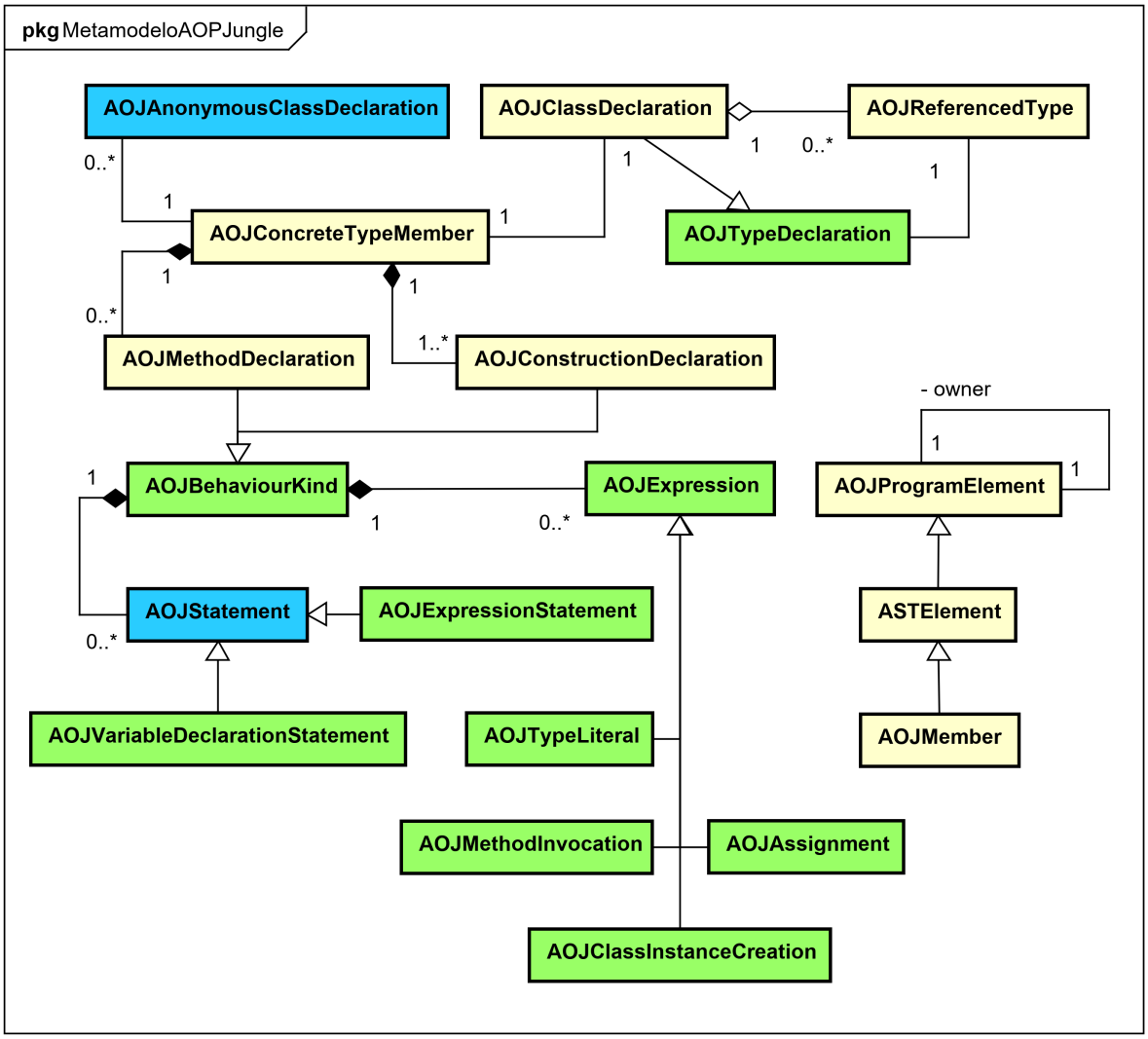


Figura 4.18 – Metamodelo do AOPJungle (DE FAVERI, 2013; TEIXEIRA JÚNIOR, 2014)

Os candidatos a oportunidades de refatoração são exibidos na visão do *Pattern Refactoring* em ordem decrescente do valor de seu resultado, conforme a Figura 4.19.

Vale ressaltar que é possível realizar o agrupamento das oportunidades pelo projeto, pacote ou classe a qual pertencem. Além disso, o duplo clique na oportunidade abre a classe no editor de código do Eclipse, na respectiva linha da oportunidade candidata.

## 4.6 Fechamento do Capítulo

Este capítulo apresentou um estudo de caso da execução das mecânicas de busca de oportunidades para refatoração utilizando o *plug-in Pattern Refactoring*. Foram escolhidos quatro projetos de repositórios abertos para a aplicação das mecânicas, com propósitos e quantidade de linhas e classes variados. Para mostrar a qualidade dos resultados das Funções Heurísticas, com

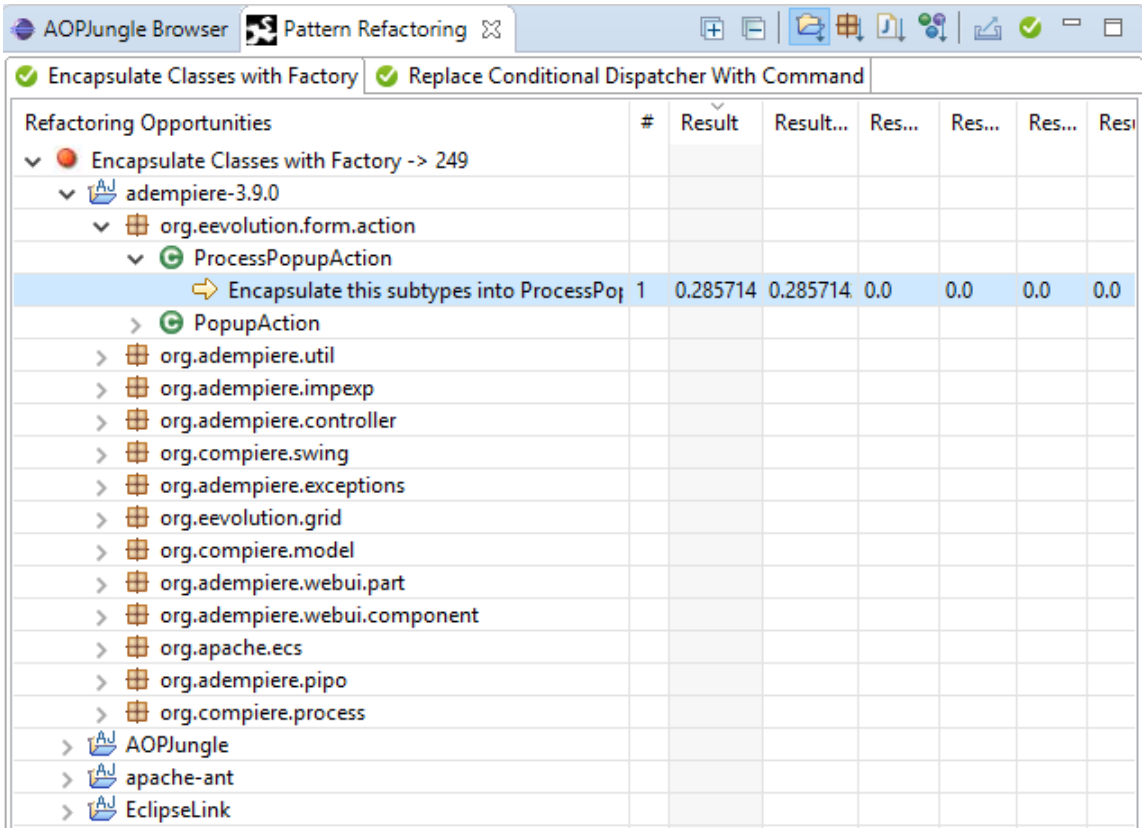


Figura 4.19 – Visão do *plug-in Pattern Refactoring*, com a exibição dos candidatos a oportunidades de refatoração.

seus respectivos valores de parâmetros e pesos, foram selecionadas as três primeiras candidatas dos projetos de cada mecânica de busca. Essas oportunidades foram aplicadas e classificadas de acordo com sua aplicabilidade, variando entre "muito alta" a "muito baixa". São discutidas as características da aplicação das refatorações, destacando suas particularidades. Ao final de cada mecânica há a consolidação e avaliação dos resultados. Em seguida, são discutidas as ameaças à validade deste trabalho. Por fim, é apresentada a implementação do *plug-in Pattern Refactoring*, que modifica e estende o metamodelo para oferecer suporte aos elementos do código investigado pelas mecânicas. O próximo capítulo apresenta as considerações finais e referências a trabalhos futuros.

# 5 CONCLUSÃO

Sistemas de software tendem a se degradar ao longo do tempo. Uma maneira de evitar tal degradação é através da refatoração, sendo que as refatorações para padrões auxiliam na estabilidade do código-fonte. A pesquisa na área de refatoração vem aumentando com o passar dos anos, com um enfoque considerável atualmente. Dentre os trabalhos na área de refatoração, três trabalhos foram considerados com maior relevância em relação a esta proposta.

Neste trabalho são apresentadas três mecânicas de busca por oportunidades de refatoração para padrões, cada uma utilizando uma função heurística parametrizável para o cálculo das oportunidades candidatas. Para cada mecânica são definidos alguns indícios que caracterizam uma oportunidade de refatoração, com uma sugestão dos seus devidos valores de parâmetros e pesos. São definidas duas funções para a normalização dos valores extraídos do código-fonte alvo, para que seja possível uma melhor definição dos pesos de cada indício nas funções heurísticas.

As mecânicas foram implementadas através do *plug-in Pattern Refactoring* para a IDE Eclipse. O *Pattern Refactoring* foi avaliado em um conjunto de projetos orientado a objetos e, baseado nos resultados, é possível afirmar que as mecânicas criadas auxiliam na identificação de oportunidades de refatoração relevantes de maneira semiautomática. Assim, a ferramenta torna mais prática a manutenção de sistemas não-triviais através de refatoração.

## 5.1 Trabalhos Futuros

A seguir são apresentadas alguma sugestões para trabalhos futuros.

- **Definir um limiar de aplicabilidade das oportunidades**. Alguns pontos importantes a serem pesquisados é a criação, verificação e validação de um limiar de aplicabilidade das oportunidades candidatas. Assim, apenas os resultados mais relevantes seriam exibidos para o desenvolvedor, que poderia focar na aplicação das refatorações de maior impacto positivo.

- **Refinar os valores dos parâmetros e pesos**. Esse limiar de aplicabilidade entretanto, está diretamente ligado aos valores de pesos e parâmetros. O refino e ajuste desses valores, tornando-os mais genéricos ou específicos de acordo com as características do projeto é um importante tema a ser investigado. Uma forma de realizar este refino é através de

métodos de decisão multicritério, como o Método AHP.

- **Ampliar a quantidade de projetos a serem testadas as mecânicas**. Para ser possível ajustar os valores de pesos e parâmetros e também verificar a aplicabilidade das oportunidades identificadas pelas mecânicas é necessário ampliar a quantidade de projetos para o estudo de caso. Uma forma é a obtenção em larga escala do código-fonte dos projetos de repositórios abertos, como o GitHub[7]. Podem existir dois focos: um para projetos com características semelhantes, e outro para projetos heterogêneos. No primeiro caso, será possível afirmar se as mecânicas são úteis ou não para determinados tipos de projetos e também qual os valores de parâmetros e pesos que trouxeram melhores resultados. No segundo caso, será possível verificar quais características os projetos possuem em comum que trazem oportunidades de refatoração mais interessantes.

- **Considerar o histórico de versões como validação dos indícios de busca**. Uma forma de validar as oportunidades identificadas é procurar no código-fonte do projeto o padrão já aplicado, e em seguida executar a mecânica de busca para o padrão em versões anteriores, quando o padrão ainda não existia. Dessa maneira, os indícios utilizados na busca poderiam ser ajustados, descartados ou adicionados novos e mais relevantes.

- **Melhorar aspectos não-funcionais do *Pattern Refactoring***. Há algumas limitações na ferramenta desenvolvida para as buscas das oportunidades. Uma delas é o desempenho, onde o tempo de extração das informações do código pode ser considerado alto. Além disso, é importante implementar uma reexecução das mecânicas, que possibilitará o acompanhamento dos valores dos indícios durante a aplicação da refatoração. Ademais, seria relevante um maior isolamento do código pertence às mecânicas, ao metamodelo e a extração do código, para que o *Pattern Refactoring* possa ser estendido para outras IDEs, preservando seu comportamento e ampliando a quantidade de locais que o mesmo pode ser utilizado.

- **Ampliar a quantidade de mecânicas de busca para padrões**. Novos indícios e mecânicas de busca podem ser adicionados ao *Pattern Refactoring*, tornando esta ferramenta mais completa. Algumas refatorações para padrões sugeridas são: Encapsular *Composite* com *Builder*, Extrair *Composite* e Substituir Árvore Implícita por *Interpreter*.

[7] https://github.com/

# REFERÊNCIAS

ABEBE, M.; YOO, C.-J. Trends, Opportunities and Challenges of Software Refactoring: a systematic literature review. **International Journal of Software Engineering & Its Applications**, [S.l.], v.8, n.6, 2014.

AL DALLAL, J. Identifying Refactoring Opportunities in Object-Oriented Code: a systematic literature review. **Information and Software Technology**, [S.l.], v.58, n.0, p.231 – 249, 2015.

AMPATZOGLOU, A. et al. The Effect of GoF Design Patterns on Stability: a case study. **IEEE Transactions on Software Engineering**, [S.l.], v.41, n.8, 2015.

DE FAVERI, C. **Uma Linguagem Específica de Domínio para Consulta em Código Orientado a Aspectos**. 2013. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Santa Maria (UFSM). Centro de Tecnologia. Programa de Pós-Graduação em Informática.

DU BOIS, B. **A study of quality improvements by refactoring**. [S.l.]: Citeseer, 2006. v.68, n.02.

DU BOIS, B.; MENS, T. Describing the impact of refactoring on internal program quality. In: INTERNATIONAL WORKSHOP ON EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE APPLICATIONS. **Anais...** [S.l.: s.n.], 2003. p.37–48.

FOWLER, M. **Refactoring Catalog**. Acessado em Julho/2019, http://refactoring.com/catalog/.

FOWLER, M. et al. **Refactoring**: improving the design of existing code. [S.l.]: Addison-Wesley Professional, 1999.

GAMMA, E. et al. **Padrões de Projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

KATAOKA, Y. et al. Automated Support for Program Refactoring Using Invariants. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'01). **Proceedings...** [S.l.: s.n.], 2001. p.736.

KERIEVSKY, J. **Refatoração para Padrões**. Porto Alegre: Bookman, 2008.

KIM, J.; BATORY, D.; DIG, D. Scripting Parametric Refactorings in Java to Retrofit Design Patterns. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME). **Anais...** [S.l.: s.n.], 2015.

MENS, T.; TAENTZER, G.; RUNGE, O. Detecting structural refactoring conflicts using critical pair analysis. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.127, n.3, p.113–128, 2005.

MENS, T.; TOURWÉ, T. A Survey of Software Refactoring. **Software Engineering, IEEE Transactions on**, [S.l.], v.30, n.2, p.126–139, 2004.

PAULI, G. d. B. **Busca por Oportunidades de Refatoração para Aplicação de Padrões de Projeto**. 2014. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Santa Maria (UFSM). Centro de Tecnologia. Programa de Pós-Graduação em Informática.

PAULI, G. d. B.; PIVETA, E. K. Searching for Refactoring Opportunities to Apply the Strategy Pattern. **X Simpósio Brasileiro de Sistemas de Informação**, [S.l.], p.357–368, 2014.

PIVETA, E. K. **Improving the Search for Refactoring Opportunities on Object-Oriented and Aspect-Oriented Software**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul (UFRGS). Instituto de Informática. Programa de Pós-Graduação em Computação.

SAATY, T. L. The Analytic Hierarchy Process: a new approach to deal with fuzziness in architecture. **Architectural Science Review**, [S.l.], v.25, n.3, p.64–69, 1982.

SAATY, T. L. What is the Analytic Hierarchy Process? In: MATHEMATICAL MODELS FOR DECISION SUPPORT, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 1988. p.109–121.

TEIXEIRA JÚNIOR, J. E. **Um Catálogo de Refatorações Envolvendo Expressões Lambda em Java**. 2014. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Santa Maria (UFSM). Centro de Tecnologia. Programa de Pós-Graduação em Informática.

VEDURADA, J.; NANDIVADA, V. K. Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State. **Proceedings of the ACM on Programming Languages**, New York, NY, USA, v.2, n.OOPSLA, p.138:1–138:28, Oct. 2018.

# ANEXOS

# ANEXO A – Framework AOPJungle

O Framework AOPJungle (DE FAVERI, 2013) foi desenvolvido para fornecer metainformações sobre código-fonte orientado a objetos e aspectos. Após o armazenamento das metainformações de programas escritos em Java e AspectJ é possível receber e executar consultas.

O AOPJungle possui quatro módulos em sua arquitetura: o *Extrator* que analisa o código-fonte e instancia o metamodelo, o *Analisador de Dependências* responsável pelas construções das associações e dependências entre os elementos do modelo, o *Processador de Complementos* que gerencia a execução dos módulos complementares, e, por fim, o *Processador de Consultas* que recebe e executa as consultas das informações do metamodelo. A Figura A.1 apresenta a arquitetura do AOPJungle.

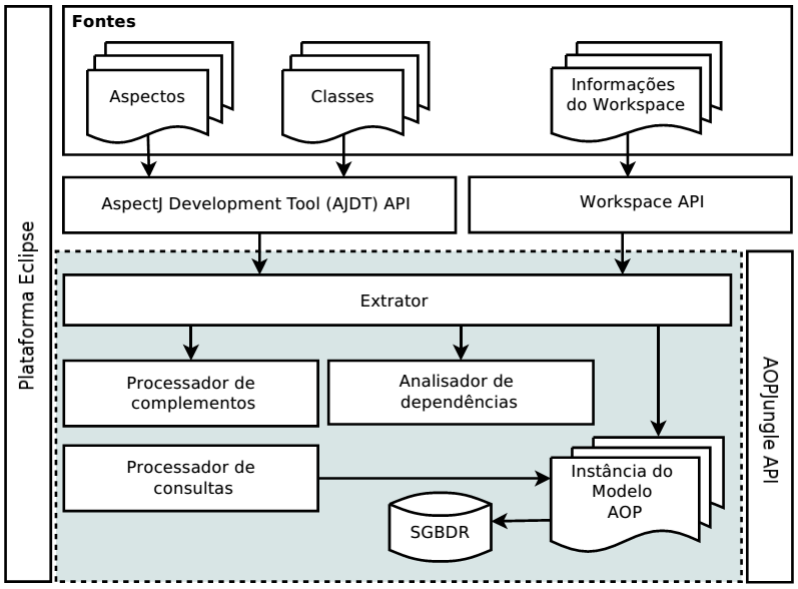


Figura A.1 – Arquitetura do AOPJungle (DE FAVERI, 2013)

Para realizar a análise do código-fonte, o módulo *Extrator* busca as informações do *workspace* do Eclipse e da AST (*Abstract Syntax Tree*) de cada unidade de compilação[8] (UC). Primeiro são identificados os projetos em AspectJ ou Java disponíveis no IDE (*Integrated Development Enviroment*) Eclipse através da API (*Application Programming Interface*) do *workspace*. Essa API fornece uma lista de recursos disponíveis como os projetos, pastas e arquivos do *workspace*. A partir da lista de projetos adquirida, o *Extrator* utiliza a API do AJDT (*AspectJ Development Tools*) para extrair as unidades de compilação e os pacotes. Por fim, o *Extrator* percorre a AST de cada unidade de compilação disponível em busca de informações necessárias

[8] Unidade de Compilação (Compilation Unit): representa o código-fonte de um arquivo Java ou AspectJ.

para a instanciação do metamodelo.

A implementação do acesso às ASTs é realizada através do padrão *Visitor* (GAMMA et al., 2000). Dessa maneira, é possível definir novas operações sem alterar os elementos de uma estrutura de dados. Assim, o módulo *Extrator* implementa uma pilha dos objetos que estão sendo visitados.

Após a instanciação do metamodelo, o módulo *Analisador de Dependências* é executado para realizar as ligações entre os elementos. Logo após, o *Processador de Complementos* é chamado para realizar análise de dados estatísticos e cálculo de métricas sobre o metamodelo. Novos complementos podem ser adicionados ao *Processador de Complementos* através do registro no configurador de complementos.

Enfim, o módulo *Processador de Consultas* pode ser executado para a realização de consultas. Para isso é necessário que o cliente informe o manipulador (*Handler*) e o código-fonte da consulta.

# ANEXO B – Listagem de Contagem de Linhas de Código (LoC)

A Listagem B.1 apresenta o código-fonte para a contagem de linhas de código (LoC) de sentenças internas à métodos.

Listagem B.1 – Método de contagem de linhas de código (LoC)

```
public static int countStatements(Statement statement) throws Exception {
  if (statement == null)
    return 0;

  if (statement instanceof EmptyStatement)
    return 0;

  if (statement instanceof AssertStatement
      || statement instanceof BreakStatement
      || statement instanceof ConstructorInvocation
      || statement instanceof ContinueStatement
      || statement instanceof ExpressionStatement
      || statement instanceof ReturnStatement
      || statement instanceof SuperConstructorInvocation
      || statement instanceof SwitchCase
      || statement instanceof VariableDeclarationStatement
      || statement instanceof SynchronizedStatement
      || statement instanceof ThrowStatement)
    return 1;

  if (statement instanceof Block) {
    int counter = 0;
    for (Object s : ((Block) statement).statements())
      counter += countStatements((Statement) s);
    return counter;
  }

  if (statement instanceof DoStatement)
    // ' 2 + ' se refere ao 'do {' e ao 'while (Expression);'
    return 2 + countStatements(((DoStatement) statement).getBody());

  if (statement instanceof EnhancedForStatement) {
    Statement bodyStatement = ((EnhancedForStatement) statement).getBody();
    // '1 + ' se refere ao for
    int count = 1 + countStatements(bodyStatement);
    if (bodyStatement instanceof Block)
      if (((Block) bodyStatement).statements().size() > 1)
        // '1 + ' caso tenha mais de uma linha, precisa do fecha chave
        return 1 + count;
    return count;
  }

  if (statement instanceof ForStatement) {
    Statement bodyStatement2 = ((ForStatement) statement).getBody();
    // '1 + ' se refere ao for
    int count2 = 1 + countStatements(bodyStatement2);
    if (bodyStatement2 instanceof Block)
      if (((Block) bodyStatement2).statements().size() > 1)
        // '1 + ' caso tenha mais de uma linha, precisa do fecha chave
```

```
50          return 1 + count2;
51        return count2;
52      }
53
54      if (statement instanceof IfStatement) {
55        // se refere a linha do if
56        int linhaDoIf = 1;
57        int linhaDoElse = 0;
58        int countThen = countStatements(((IfStatement)
              statement).getThenStatement());
59        int countElse = countStatements(((IfStatement)
              statement).getElseStatement());
60
61        boolean isBlock = ((IfStatement) statement).getElseStatement()
              instanceof Block;
62        boolean isIf = ((IfStatement) statement).getElseStatement() instanceof
              IfStatement;
63
64        if (countElse >= 1 && !isIf)
65          linhaDoElse = 1;
66
67        int fechaChaveAdicional = 0;
68        if (countThen <= 1 && linhaDoElse == 0)
69          fechaChaveAdicional = 0;
70        else if (countThen <= 1 && isBlock && countElse > 1)
71          fechaChaveAdicional = 1;
72        else if (countThen <= 1 && isBlock && countElse <= 1)
73          fechaChaveAdicional = 0;
74        else if (countThen > 1 && countElse == 0)
75          fechaChaveAdicional = 1;
76
77        int retorno = linhaDoIf + countThen + linhaDoElse + countElse +
              fechaChaveAdicional;
78        return retorno;
79      }
80
81      if (statement instanceof LabeledStatement) {
82        int count3 = countStatements(((LabeledStatement) statement).getBody());
83        if (count3 > 1)
84          // '2 + ' se refere ao inicio e ao fecha chave
85          return 2 + count3;
86        // '1 + ' se refere somente ao inicio do LabeledStatement
87        return 1 + count3;
88      }
89
90      if (statement instanceof SwitchStatement) {
91        int counter = 0;
92        for (Object s : ((SwitchStatement) statement).statements())
93          // '2 + ' se refere ao inicio e o fecha chave
94          counter = 2 + countStatements((Statement) s);
95        return counter;
96      }
97
98      if (statement instanceof TryStatement) {
99        int linhaDoTry = 1;
100       int countBody = countStatements(((TryStatement) statement).getBody());
101
102       int countCatches = 0;
```

```
103     for (Object obj : ((TryStatement) statement).catchClauses())
104       // '1 + ' se refere a linha do catch
105       countCatches += 1 + countStatements(((CatchClause) obj).getBody());
106
107     int linhaDoFinally = 0;
108     int countFinally = countStatements(((TryStatement)
            statement).getFinally());
109     if (countFinally > 0)
110       linhaDoFinally = 1;
111
112     int fechaChaveFinal = 1;
113
114     return linhaDoTry + countBody + countCatches + linhaDoFinally +
            countFinally + fechaChaveFinal;
115   }
116
117   if (statement instanceof WhileStatement) {
118     int countWhile = countStatements(((WhileStatement)
            statement).getBody());
119     if (countWhile > 1)
120       // '2 + ' se refere ao while e o fecha chave
121       return 2 + countWhile;
122     // '1 + ' se refere ao while
123     return 1 + countWhile;
124   }
125
126   throw new Exception("Statement desconhecido!");
127 }
```