THALYS DE AGUIAR GOMES

ANÁLISE COMPARATIVA DA APLICAÇÃO DE ABORDAGENS DE AUTOMATIZAÇÃO DE REFATORAÇÃO PARA O PADRÃO DE PROJETO STRATEGY

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de Software.

SÃO PAULO
2013

THALYS DE AGUIAR GOMES

ANÁLISE COMPARATIVA DA APLICAÇÃO DE ABORDAGENS DE AUTOMATIZAÇÃO DE REFATORAÇÃO PARA O PADRÃO DE PROJETO STRATEGY

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dr. Fábio Levy Siqueira

SÃO PAULO
2013

FICHA CATALOGRÁFICA

GOMES, THALYS DE AGUIAR

Análise comparativa entre abordagens de automatização de refatoração para o padrão de projeto *Strategy* / T.A. Gomes. -- São Paulo, 2013.

47 p.

Monografia (MBA em Tecnologia de Software) – Escola Politécnica da Universidade de São Paulo. Programa de Educação Continuada em Engenharia.

1. Desenvolvimento de software 2.Refatoração I. Universidade de São Paulo. Escola Politécnica. Programa de Educação Continuada em Engenharia II.t.

DEDICATÓRIA

Dedico este trabalho a minha esposa e toda minha família que têm me dado apoio e ânimo para continuar e sempre melhorar.

AGRADECIMENTOS

Agradeço a Deus por me guiar e ser o Senhor da minha vida.

À minha esposa por estar sempre ao meu lado, me ouvindo, me auxiliando e me incentivando a seguir sempre em frente.

Aos meus pais que sempre acreditaram e investiram em mim, por me transmitirem segurança e força para chegar até aqui.

Agradeço ao professor orientador Prof. Dr. Fábio Levy Siqueira, pela confiança, pelo apoio e paciência nas dificuldades encontradas possibilitando a conclusão deste trabalho.

À todos que de alguma maneira contribuíram direta ou indiretamente para que a realização deste trabalho fosse possível.

RESUMO

É improvável que se desenvolvam sistemas de software que não sofrerão mudanças, especialmente de requisitos ou de *design*. No entanto, se as manutenções no sistema não ocorrerem de maneira controlada elas podem levar a problemas de qualidade ou até mesmo ao caos. Entre as técnicas que auxiliam na manutenção e na melhoria da qualidade de software está a Refatoração. Para aprimorar a técnica de refatoração e torná-la mais eficiente e confiável, foi criado o conceito de automatização de refatoração. Este trabalho tem como objetivo realizar uma análise comparativa entre duas abordagens de automatização de refatoração para o padrão de projeto *Strategy*. Nesta análise, serão apresentadas as vantagens e desvantagens da aplicação destas abordagens para a manutenção de um grande sistema de seguros. Por fim, estas duas abordagens serão também avaliadas em relação às melhorias obtidas sobre o código fonte do sistema.

ABSTRACT

It’s unlikely to develop software systems that don’t suffer any change, especially requirements or design changes. Nevertheless, if such changes don’t occur in a controlled way, they can lead to quality problems or even to chaos. Among the techniques that help to realize maintenance and that improve the quality of software is Refactoring. In order to improve the refactoring technique and to make its use more efficient and reliable, the concept of automated refactoring was created. This work realizes a comparative analysis between two approaches of automated refactoring of the *Strategy* design pattern. This analysis presents the advantages and disadvantages of the application of these approaches in the maintenance of a large insurance system. Finally, these two approaches will also be assessed in regard of the improvements obtained in the source code of the system.

LISTA DE FIGURAS

Pág.

FIGURA 1: REPRESENTAÇÃO DO PADRÃO *STRATEGY,* EXTRAÍDO DE (GAMMA ET AL., 1995). 20
FIGURA 2: REPRESENTAÇÃO DO MÉTODO DE CINNÉIDE (2001) PARA REFATORAÇÃO DE PADRÕES DE PROJETO. 26
FIGURA 3: DEFINIÇÃO APRESENTADA POR CINNÉIDE (2001) PARA APLICAÇÃO DO PADRÃO *STRATEGY*. 28
FIGURA 4: TRECHO DO CÓDIGO ONDE SERÁ APLICADO O PADRÃO *STRATEGY*. 32
FIGURA 5: TRECHO DE CÓDIGO APÓS A APLICAÇÃO DA REFATORAÇÃO. 35
FIGURA 6: DEFINIÇÃO DA NOVA MINITRANSFORMAÇÃO DELEGATIONBLOCK PARA UMA COMPLETA APLICAÇÃO DO PADRÃO *STRATEGY*. 37
FIGURA 7: BLOCO NA CLASSE AGENTLIQUIDATIONDAO QUE CONFIGURA E INVOCA A ESTRATÉGIA. 38

LISTA DE TABELAS

Pág.

TABELA 1: RESULTADO DA APLICAÇÃO DAS MÉTRICAS...38

SUMÁRIO

1. INTRODUÇÃO 10

1.1. Motivação 11

1.2. Objetivo 11

1.3. Justificativa 11

1.4. Estrutura do trabalho 12

2. REVISÃO BIBLIOGRÁFICA 14

2.1. Refatoração 14

2.2. Automatização de Refatoração 16

2.3. Automatização de refatorações para padrões de projetos 18

2.3.1. Padrões de projeto 18

2.3.2. Padrão de projeto *Strategy*. 19

2.3.3. Abordagens de Automação de refatoração para padrões de projeto 21

3. Abordagens de Automação de Refatoração para o Padrão de Projeto *Strategy* 23

3.1. Ferramenta *JDeodorant* 24

3.2. Método de Cinnéide (2001) 25

3.2.1. Especificação de pré e pós-condições 27

3.2.2. Refatoração para o padrão *Strategy* 28

4. Análise Comparativa 30

4.1. Contexto 30

4.2. Critérios de análise 32

4.3. Aplicação da ferramenta 35

4.4. Aplicação do método 35

4.5. Análise dos resultados 38

5. CONCLUSÃO 42

5.1. Resumo 42

5.2. Trabalhos futuros 43

5.3. Contribuições 43

REFERÊNCIAS 44

1. INTRODUÇÃO

Dificilmente se desenvolve sistemas de software que não sofrerão mudanças, especialmente de requisitos ou de design. Assim, uma vez concluído, qualquer sistema está sujeito a mudanças. No entanto se as manutenções no sistema não ocorrerem de maneira controlada, podem levar a problemas de qualidade do sistema ou até mesmo ao caos.

Estas adversidades demonstram que a manutenção de um software não é uma tarefa fácil, consumindo muito tempo. Projetar um software para ser facilmente mantido e ampliado é ainda mais difícil, pois força os desenvolvedores a não apenas pensarem em detalhes de uma solução específica, mas também em detalhes do escopo do problema atual e em possíveis evoluções.

Entre os processos que auxiliam na manutenção e na melhoria da qualidade de software está a Refatoração. A refatoração é uma técnica de reestruturar um código existente, alterando apenas sua estrutura interna e não o seu comportamento externo (FOWLER, 1999). Segundo Fowler (1999), uma refatoração, individualmente, pode parecer não trazer tantos benefícios ao código fonte. Seu acúmulo, contudo, pode levar a significativas melhorias no sistema.

Alguns processos de desenvolvimento ágil sugerem a utilização de refatoração. De fato, Hill, Parnin e Black (2011) afirmam que atualmente, o uso de refatoração tem se disseminado entre desenvolvedores. Por outro lado, algumas refatorações podem ser tediosas e propensas a erros (VAKILIAN et al., 2012), e até mesmo levar muito tempo. Para superar estes fatores contrários e para tornar o uso de refatoração mais eficiente e confiável, foi criado o conceito de automatização de refatoração.

1.1. Motivação

A melhoria contínua do sistema e a atualização do código fonte são essenciais em sistemas comerciais. Isto se torna mais crítico para sistemas que possuam extensas bases de código e grandes equipes de desenvolvimento. Por estes motivos, são necessárias abordagens automatizadas que melhorem o código fonte, aumentem a qualidade do sistema e preservem seu comportamento. Adicionalmente, é importante que uma abordagem automatizada leve menos tempo que uma refatoração manual.

A utilização de padrões de projetos (GAMMA et al., 1995) traz soluções reutilizáveis e torna os sistemas mais flexíveis para atender problemas comuns do desenvolvimento de software. Estes ganhos podem ser maximizados com a automatização de refatoração. Esta tarefa, contudo, pode ser complexa. Por isto, alguns trabalhos têm proposto a utilização de ferramentas e métodos para realizar automatização de refatoração para padrões de projetos.

1.2. Objetivo

Este trabalho tem como objetivo realizar uma análise comparativa entre duas abordagens de automatização de refatoração para o padrão de projeto *Strategy*. Nesta análise, serão apresentadas as vantagens e desvantagens da aplicação da abordagem para a manutenção de um grande sistema de seguros. Por fim, estas duas abordagens serão também avaliadas em relação às melhorias obtidas sobre o código fonte do sistema.

1.3. Justificativa

Existem alguns trabalhos que tratam o assunto de automatização de refatoração para padrões de projetos. Dentre eles há o trabalho de Christopoulou et al., (2012), que utiliza a ferramenta *JDeodorant* (2013) para identificação da oportunidade e aplicação da refatoração para o padrão de

projeto *Strategy* (GAMMA et al., 1995). O foco principal desse trabalho é a identificação de oportunidade de refatoração para o padrão *Strategy*.

Outra proposta é a de Cinnéide (2001), que apresenta um método para criação de automatização de refatoração para padrões de projetos. Seu foco principal é apresentar um método de criação de transformações para automatização de refatoração de sete padrões de projetos propostos por Gamma et al. (1995), possibilitando a reutilização do método ao aplicar a diferentes padrões e na preservação de comportamento, fator fundamental na utilização da refatoração.

Em nenhum dos dois casos há uma análise mais detalhada do ganho de melhoria na aplicação da automatização de refatoração ao sistema existente.

1.4. Estrutura do trabalho

O Capítulo 1 (Introdução) apresenta as motivações, o objetivo, as justificativas e a estrutura do trabalho.

O Capítulo 2 (Revisão bibliográfica) apresenta o fundamento teórico deste trabalho. Primeiramente é apresentado a refatoração e suas definições e atividades. Posteriormente são apresentadas as abordagens para automatização de refatoração e a automatização de refatoração para padrões de projeto. Também há uma breve apresentação de padrões de projeto e trabalhos com propostas de automatização de refatoração para padrões de projeto.

O Capítulo 3 (Abordagens de Automatização de Refatoração para o Padrão de Projeto *Strategy*) apresenta em detalhe as duas abordagens que propõem a automatização de refatoração aplicáveis ao padrão de projeto *Strategy*.

O Capítulo 4 (Análise Comparativa) apresenta inicialmente o contexto e os critérios a serem analisados. Posteriormente efetua a aplicação da refatoração

com a ferramenta e o método. Finalmente, analisa os resultados obtidos da aplicação.

O Capítulo 5 (Conclusão) apresenta a conclusão do trabalho, apontando as contribuições e trabalhos futuros.

2. REVISÃO BIBLIOGRÁFICA

Para analisar as abordagens de automatização de refatoração para o padrão de projeto *Strategy*, neste capítulo são apresentados os conceitos de refatoração, automatização de refatoração, padrões de projetos e algumas automatizações de refatoração para padrões de projetos.

2.1. Refatoração

Inicialmente conhecida como reestruturação (OPDYKE, 1992) no caso específico de desenvolvimento orientado a objetos, a refatoração foi originalmente introduzida por Opdyke em sua tese de doutorado (OPDYKE, 1992). Segundo Opdyke, refatoração é o processo de mudança de um sistema de software de tal forma que ele não altera o comportamento externo do código e ainda melhora a sua estrutura interna. É uma forma disciplinada de melhorar o código e minimizar as chances de introduzir erros, buscando manter as pré-condições e pós-condições. As pré-condições e pós-condições são utilizadas para manter preservação de comportamento do sistema à qual faz parte da definição da refatoração. Em essência, quando se refatora, o design do código é melhorado depois de ter sido escrito (OPDYKE, 1992). A refatoração também pode ser utilizada no contexto de evolução de software e reengenharia, melhorando atributos de qualidade do software (por exemplo, extensibilidade, modularidade, capacidade de reutilização, complexidade de manutenção, eficácia) (OPDYKE, 1992).

Mens e Tourwé (2004) discutem várias pesquisas formais e informais sobre a preservação de comportamento. Em Cinnéide (2001) foi utilizada uma linguagem própria para definição de preservação de comportamento, apresentada na seção 3.2.1.

Segundo Fowler (1999), cada etapa da refatoração é simples. Move-se um atributo de uma classe para outra, retira-se um código de um método para fazer outro método, muda-se um código para cima ou para baixo de uma

hierarquia etc. No entanto, os efeitos cumulativos dessas pequenas mudanças podem melhorar radicalmente o design do sistema.

Mens e Tourwé (2004) realizaram uma pesquisa detalhada sobre refatoração de software. Como parte da sua pesquisa, os autores propuseram uma abordagem geral com 6 atividades para se aplicar refatoração (MENS; TOURWÉ, 2004):

a) *Identificar qual parte do software deve ser refatorada*: a partir do nível de abstração da refatoração que se deseja aplicar, identificar em quais lugares é necessário aplicar refatoração.
b) *Determinar quais refatorações devem ser aplicadas aos locais identificados*: Fowler (1999) define maus cheiros como uma estrutura de código que sugere, ou que identifica-se um bom candidato a ser refatorado. Adicionalmente, descreve várias técnicas que podem ser usadas para refatoração, separadas em categorias como maus cheiros (*bad smells*) no código, composição de métodos, movendo recursos entre objetos, organização de dados, simplificando operações condicionais, chamadas a métodos mais simples e lidando com generalização e, até mesmo no caso da linguagem Java, para aplicação da convenção de código Java.
c) *Garantir que a refatoração aplicada preserva o comportamento*: Opdyke (1992) inicialmente propôs que seja determinado um conjunto de pré-condições e pós-condições para avaliar o comportamento do código. Em geral, vários trabalhos propõem que seja utilizado um conjunto de testes para que se garanta a preservação do comportamento do componente refatorado.
d) *Aplicar a refatoração*: aplicar as refatorações apresentadas no passo b, desenvolvendo e implementando as devidas melhorias.
e) *Avaliar o efeito da refatoração com relação a características de qualidade*: fazendo uma distinção entre atributos de qualidade de software, é possível analisar os que são perceptíveis em tempo de execução como, desempenho, segurança, disponibilidade, funcionalidade e usabilidade, os não perceptíveis em tempo de

execução como, modificabilidade, portabilidade, reusabilidade, integrabilidade e testabilidade, e aqueles relacionados a qualidades intrínsecas da arquitetura, como, integrabilidade conceitual, correção, completude e facilidade de construção (ABRAN et al., 2005). A idéia central é que a cada refatoração executada seja feita também uma relação dos atributos de qualidade atingidos. Também é possível que se classifique o quanto estas refatorações atingiram, em melhoria de código no sistema.

f) *Manter a coerência entre o código refatorado*: por fim, é necessário manter a coerência entre o código refatorado e suas respectivas suítes de testes.

2.2. Automatização de Refatoração

A refatoração pode ser feita manualmente, mas algumas tarefas repetitivas como identificações e refatoração de maus cheiros podem ser muito demoradas. O quanto uma ferramenta de automatização de refatoração contribui com o desenvolvedor depende de quais das atividades de refatoração da Seção 2.1 são suportadas pela automatização, podendo economizar tempo de desenvolvimento.

As IDEs como Eclipse (2013), NetBeans (2013) e IntelliJ (2013) suportam refatorações semi-automáticas. A refatoração é semi-automática quando permanece ao desenvolvedor a tarefa de identificar qual parte do software precisa ser refatorada e selecionar a refatoração mais adequada a se aplicar. A aplicação efetiva da refatoração é automática. Herbold, Grabowski e Neukirchen (2009) propõem um plug-in para o IDE Eclipse, que utiliza para análise do código fonte na detecção de erros e outros problemas ao nível do código fonte. Esse plug-in detecta problemas relacionados com a estrutura interna de um sistema e sugere ao desenvolvedor a utilização de uma determinada refatoração integrada ao Eclipse (2013). Isso aperfeiçoa ainda mais as IDEs que nativamente suportam apenas refatorações semi-automáticas.

De uma forma geral, o uso, desuso e uso indevido de automatização de refatoração é analisado por Vakilian et al. (2012). Através da coleta de dados qualitativos e quantitativos em interações com as refatorações automatizadas do Eclipse, os autores estudaram 26 desenvolvedores trabalhando em seus ambientes ao longo de 3 meses. Mesmo em IDEs como Eclipse, NetBeans e IntelliJ, os desenvolvedores ainda tem barreiras quanto ao uso de refatorações automatizadas (VAKILIAN et al., 2012). Sobre a utilização de ferramenta na automatização de refatoração, identificou-se que os desenvolvedores preferem métodos leves de invocação, como atalhos. Em relação ao desuso de uma refatoração automatizada, ele ocorre quando um desenvolvedor executa uma refatoração manualmente mesmo que essa seja suportada por alguma IDE. Hill, Parnin e Black (2011) analisaram o histórico de controle de versão do Eclipse (2013) e identificaram que 90% das refatorações são executadas manualmente, sem utilizar a ferramenta de refatoração. Vakilian et al. (2012) também identificaram que em média os desenvolvedores não tinham conhecimento de mais de 9 das refatorações automatizadas do IDE Eclipse. O estudo de Vakilian et al. (2012) também fornece evidências de que refatorações automatizadas cujos nomes são difíceis de entender, muito técnicos, confusos, ou que não tenham seus benefícios claros ficam mais propensos ao desuso da automatização devido à sobrecarga do usuário na avaliação e utilização da automatização. O uso indevido ou mau uso da refatoração automatizada é, por exemplo, quando a ferramenta relata um problema (uma mensagem de erro ou aviso) ao aplicar a refatoração, não havendo mais garantia de que seu comportamento será preservado (VAKILIAN et al., 2012). No estudo (VAKILIAN et al., 2012), isso aconteceu em mais de 50% das refatorações aplicadas. A sugestão de Vakilian et al., (2012) é que os projetistas escolham nomes mais intuitivos, consistentes e melhorem as opções de pré-visualização para ajudar os desenvolvedores a prever os resultados das refatorações automatizadas. Por fim, Vakilian et al., (2012) concluíam que, apesar das dificuldades por parte da confiabilidade, design e usabilidade das ferramentas, os desenvolvedores ainda apóiam e estão dispostos a utilizar ferramentas para automatizar a refatoração, principalmente de refatorações que estão certos de o quanto a ferramenta irá alterar do código.

Outra abordagem para automatização da refatoração é apresentada por Moghadam e Cinnéide (2012). Os autores utilizam a diferenciação de *design* para automatização de refatoração, apresentando um processo de refatoração em que são utilizadas algumas ferramentas, como JDEvAn - (*Java Design Evolution and Analysis*) (2013) e Code-Imp (KEEFFE; CINNÉIDE, 2008). É extraído um modelo UML do sistema e criado um modelo UML desejado. Fazendo a comparação dos dois modelos são verificadas as diferenças e, usando a ferramenta Code-Imp (KEEFFE; CINNÉIDE, 2008), essas diferenças são corrigidas através de refatorações que são aplicadas automaticamente. Esta ferramenta já conta com 16 tipos de refatorações podendo aplica-las automaticamente.

Outra refatoração automatizada é proposta por Khatchadourian, Sawin e Rountev (2007). Os autores propõem um algoritmo para analisar o código existente em busca de *tipagem não segura*, substituindo conjuntos fixos de constantes (*static final*) de uma lista de valores pré-definidas, por tipos *enum,* recurso criado a partir de Java 5. Em seus testes em 17 aplicações de código aberto Java, o algoritmo apresentou-se capaz de identificar e refatorar 87% dos campos que eventualmente poderiam ser utilizados com padrão *enum*.

2.3. Automatização de Refatorações para Padrões de Projetos

Nas seções a seguir são apresentadas uma visão geral de padrões de projeto, alguns detalhes mais específicos do padrão *Strategy* e outras abordagens de automatização para padrões de projetos.

2.3.1. Padrões de projeto

Padrão de projeto (GAMMA et al., 1995) é um conjunto de soluções reutilizáveis para atender problemas comuns do desenvolvimento de sistemas orientado a objetos, tornando os sistemas mais flexíveis, reutilizáveis e

melhorando a manutenção, além de fornecer uma linguagem comum aos desenvolvedores.

Em Gamma et al. (1995) foi criado um catálogo de padrões com um formato consistente em que os elementos essenciais de um padrão são: nome, problema a ser resolvido, descrição da solução e as consequências ou resultados da utilização do padrão. Esse catálogo contém 23 padrões que variam de granularidade e abstração. Para facilitar o aprendizado deles, eles foram categorizados por seu propósito:

- *Padrões de criação*: dizem respeito ao processo de criação de objetos.
- *Padrões estruturais*: lidam com a composição de classes ou objetos.
- *Padrões comportamentais*: caracterizam as maneiras pelas quais as classes ou objetos se comunicam e delegam responsabilidades.

Ao desenvolver um sistema orientado a objetos, um dos desafios é saber qual o nível de abstração do mundo real deve-se adotar. Atender a mudanças no sistema com facilidade de manutenção e flexibilidade para alterar a solução não é uma tarefa fácil. As abstrações que surgem durante o desenvolvimento do projeto são a chave para se criar um projeto flexível e os padrões de Gamma et al. (1995) auxiliam a chegar ao maior nível de flexibilidade no projeto.

2.3.2. Padrão de projeto *Strategy*

Um dos padrões de projetos apresentados por Gamma et al. (1995) é o padrão *Strategy*. O padrão *Strategy* é um padrão comportamental que tem como objetivo definir uma família de algoritmos, encapsular cada um e torná-los intercambiáveis, independentemente dos clientes que o utilizam. Conhecido também como *Policy*, o padrão *Strategy* é voltado para encapsulamento de algoritmos que podem variar para fornecer um comportamento adequado de acordo com uma necessidade de negócio.

O uso do padrão *Strategy* (GAMMA et al., 1995) é indicado quando uma classe define muitos comportamentos e estes aparecem em múltiplas declarações condicionais em suas operações. A representação do mecanismo da estratégia pode ser visualizada na Figura 1.

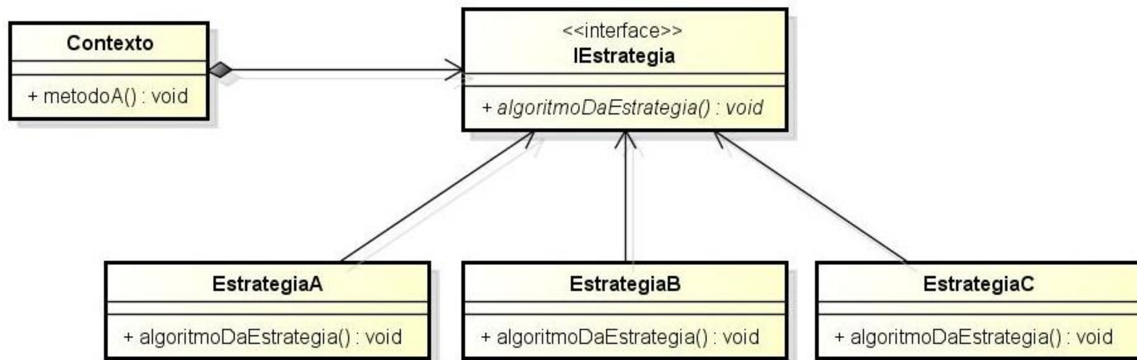

Figura 1: Representação do padrão *Strategy* (GAMMA et al., 1995).

O padrão *Strategy* define 3 papéis como participantes da estratégia (GAMMA et al., 1995):

- *IEstrategia*: Declara uma interface comum a todos os algoritmos suportados. A classe *Contexto* utiliza essa interface para chamar o algoritmo definido por uma classe concreta da estratégia, como, *EstrategiaA*.
- *EstrategiaConcreta*: Classes que implementam os algoritmos usando a interface *IEstrategia*.
- *Contexto*: Representa a classe de contexto, também conhecida como cliente. É configurada para manter referência a um objeto *EstrategiaConcreta* e utiliza o algoritmo para alguma lógica de negócio.

A hierarquia da interface *IEstrategia* da Figura 1 define uma família de algoritmos ou comportamentos de um mesmo contexto que podem ser reutilizadas. Portanto o padrão *Strategy* (GAMMA et al., 1995) apenas deverá ser utilizado quando a variação de comportamento é relevante para os contextos (que podem ser vistos como os clientes). Por fim, um ponto de atenção é que o padrão *Strategy* aumenta o número de objetos no sistema,

principalmente de acordo com a quantidade de variações de algoritmos (GAMMA et al., 1995).

2.3.3. Abordagens de Automação de Refatoração para Padrões de Projeto

Existem alguns trabalhos na área de automatização de refatoração que apresentam abordagens e ferramentas para apoiar a automação para padrões de projeto (CHRISTOPOULOU et al., 2012; CINNÉIDE, 2001; JEBELEAN; CHIRILA; CRETU, 2010; JEON; LEE; BAE, 2002).

Em Industrial Logic (2013) Eric Gamma, afirma que o alvo de refatorações são padrões de projetos, Kerievsky (2004) orienta que para a refatoração ser valiosa, ela deve possuir um foco, e não ser somente um exercício intelectual abstrato. O autor fornece instruções como um guia sobre como melhorar o seu código através da introdução de padrões adequados a determinadas situações e apresenta princípios que fundamentam a implementação dos padrões de projeto. É utilizada uma estrutura chamada de Mecanismo, como o passo a passo de cada refatoração para aplicar determinado padrão de projeto.

Uma abordagem para padrão de projeto *Composite* (GAMMA et al., 1995) é apresentado por Jebelean, Chirila e Cretu (2010). Essa abordagem utiliza um plug-in para Eclipse (2013) que possibilita consultar o código-fonte Java através de árvore de sintaxe abstrata (*Abstract Syntax Tree - AST*) representada internamente por uma base de conhecimento Prolog. Cada nó *AST* é representado por fatos em Prolog que dizem algo sobre um pequeno aspecto do projeto Java em análise. Por fim, define uma regra de predicado para detectar em um determinado cenário de anti-padrão *Composite* onde um desenvolvedor teria que analisar se verdadeiramente há necessidade de utilização do padrão *Composite* ou não. Usando esta abordagem tornou-se possível a detecção de problemas em cada um dos seguintes cinco padrões: *Composite, Abstract Factory, Strategy, State e Visitor* de Gamma et al. (1995).

Anteriormente, Jeon; Lee; Bae (2002) utilizaram o mesmo conceito de representar o código Java de um projeto para base de conhecimento Prolog. E assim, através de fatos Prolog, fazer consultas e análises para aplicação de um padrão de projeto. No trabalho, foi realizada uma refatoração para o padrão de projeto *AbstractFactory* em que a identificação de oportunidades de refatoração e as transformações no código fonte Java são realizadas através de passos obtidos pelas consultas via Prolog. A maior diferença em relação ao trabalho de Jebelean, Chirila e Cretu (2010) é que toda a extração de código Java para representação como predicados Prolog era realizada de maneira manual.

Em Cinnéide (2001) é proposto uma modelo para criação de transformações para padrões de projetos. Na sua tese foi criado um mecanismo de *minitransformações* e *minipadrões* que compõem a transformação para um determinado padrão de projeto e o conceito de reutilização de *minitransformações* para as transformações aplicadas a diferentes padrões propostos por Gamma et al. (1995). Tratando também do padrão de projeto *Strategy*, em Christopoulou et al. (2012) é apresentado um algoritmo para identificação de oportunidades de refatoração e a aplicação da refatoração de maneira automatizada. Esses dois trabalhos serão apresentados em detalhes na seção 3.

3. Abordagens de Automação de Refatoração para o Padrão de Projeto *Strategy*

No padrão de projeto *State* (GAMMA et al., 1995), um objeto delega requisições para outro objeto de estado que representa o seu estado atual. No padrão *Strategy*, um objeto delega um pedido específico a outro objeto que representa uma estratégia para a realização do pedido. Um objeto terá apenas um estado, mas podem existir muitas estratégias para diferentes solicitações. Em ambos os casos este mecanismo é conhecido como delegação. O objetivo de ambos os padrões é mudar o comportamento de um objeto alterando os objetos a quem é delegada a solicitação.

A similaridade entre esses dois padrões em termos de estrutura e adequação para lidar com expressões condicionais muitas vezes faz com que sejam tratados da mesma forma. Fowler (1999) propõe a refatoração de expressões condicionais "Substituir Tipo de Código com *State/Strategy*" para lidar com código de tipo que afeta o comportamento de uma classe, mas não pode usar subclasses. Esta refatoração é similar à refatoração "Substituir Tipo de Código por Subclasses", mas com a diferença de que uma estratégia pode ser alterada por outra em tempo de execução. Esta é uma das características do padrão *Strategy*. Já em Kerievsky (2004) as refatorações para os padrões *Strategy* e *State* são tratadas de maneira completamente distinta. Para o padrão *Strategy* a refatoração é chamada de "Substituir lógica condicional por *Strategy*". Para o padrão *State,* a refatoração é chamada de "Substituir condicionais que alteram estado por *State*".

Existem alguns trabalhos que propõem automatizações para o padrão *Strategy*. O trabalho de Christopoulou et al. (2012) é focado na identificação e refatoração para o padrão *State/Strategy* através da ferramenta *JDeodorant* que também fornece a identificação e refatoração de maus cheiros. Outro trabalho que propõe a automatização de refatoração para o padrão de projeto *Strategy* é o de Cinnéide (2001). Nesse trabalho a transformação é tratada especificamente para o padrão *Strategy*. Adicionalmente, o método proposto

também permite a aplicação de outros padrões. A seguir são apresentados esses dois trabalhos com maiores detalhes.

3.1. Ferramenta *JDeodorant*

JDeodorant (CHRISTOPOULOU et al., 2012) é um plug-in desenvolvido para a IDE Eclipse (2013) para automatização de refatoração. Ele é utilizado na identificação de problemas conhecidos como “maus cheiros” e os resolve através da aplicação de refatorações apropriadas. Atualmente a ferramenta identifica quatro tipos de maus cheiros:

- *Feature Envy*: problemas que são resolvidos através da aplicação apropriada da refatoração *Mover Método* (FOWLER, 1999).
- *Type Checking*: problemas que são resolvidos através da aplicação das refatorações *Substituição de Código Condicional por Polimorfismo* e *Substituição de Código de Tipo por State/Strategy* (FOWLER, 1999).
- *Long Method*: problemas que são resolvidos através da aplicação apropriada da refatoração *Extrair Método* (FOWLER, 1999).
- *God Class*: problemas que são resolvidos através da aplicação apropriada da refatoração *Extrair Classe* (FOWLER, 1999).

O uso da ferramenta para a identificação de oportunidade de refatoração para padrão *Strategy* e a aplicação da refatoração através de uma transformação no código fonte é apresentada por Christopoulou et al. (2012). O algoritmo de identificação proposto trabalha da seguinte maneira: dado um código fonte selecionado para análise, são procurados blocos de declarações como: *switch, if/else, if/else if/.../else* nos métodos de todas as classes deste projeto. São considerados para análise os blocos *s* em que:

- *s* deve ter pelo menos duas condições;
- *s* não pode ser parte de um método estático ou bloco de código estático;

- *s* não pode ser parte de um método redefinido de uma classe que não é do sistema, como, por exemplo, métodos da biblioteca da linguagem Java *equals, toString, hashCode*;
- *s* deve possuir pelo menos duas linhas de execução, exceto para os casos que contêm linhas de execução com invocações de métodos da própria classe *Context*.

Após a identificação, a ferramenta possibilita que o usuário aplique a refatoração a um código existente de maneira completamente automática. A aplicação da refatoração consiste em:

a) Identificada a refatoração para padrão *Strategy*, a mesma estará disponível na *view Type Checking* do plug-in no Eclipse.

b) Ao seleciona-la, o bloco de código identificado é apresentado e se torna disponível a opção de *Apply Refactoring*.

c) O plug-in disponibiliza uma tela para que, se desejado, o usuário personalize os nomes das classes que serão geradas e visualize passo a passo as transformações que serão efetuadas.

De acordo com Christopoulou et al. (2012) a ferramenta foi extensamente testada por dois experientes engenheiros de software em termos do algoritmo de identificação, classificação das sugestões de refatoração e tempos de execução e apresentou eficácia tal que mais da metade das refatorações efetuadas foram realizadas com sucesso.

3.2. Método de Cinnéide (2001)

Cinnéide (2001) propõe um método para automatização de refatoração dos padrões de projeto propostos por Gamma et al. (1995). Nesse método inicialmente deve-se identificar um precursor, que expressa à intenção de utilizar um padrão de projeto. Esta transformação para um determinado padrão

é então decomposto em uma sequência de *minipadrões*, como apresentado na Figura 2.

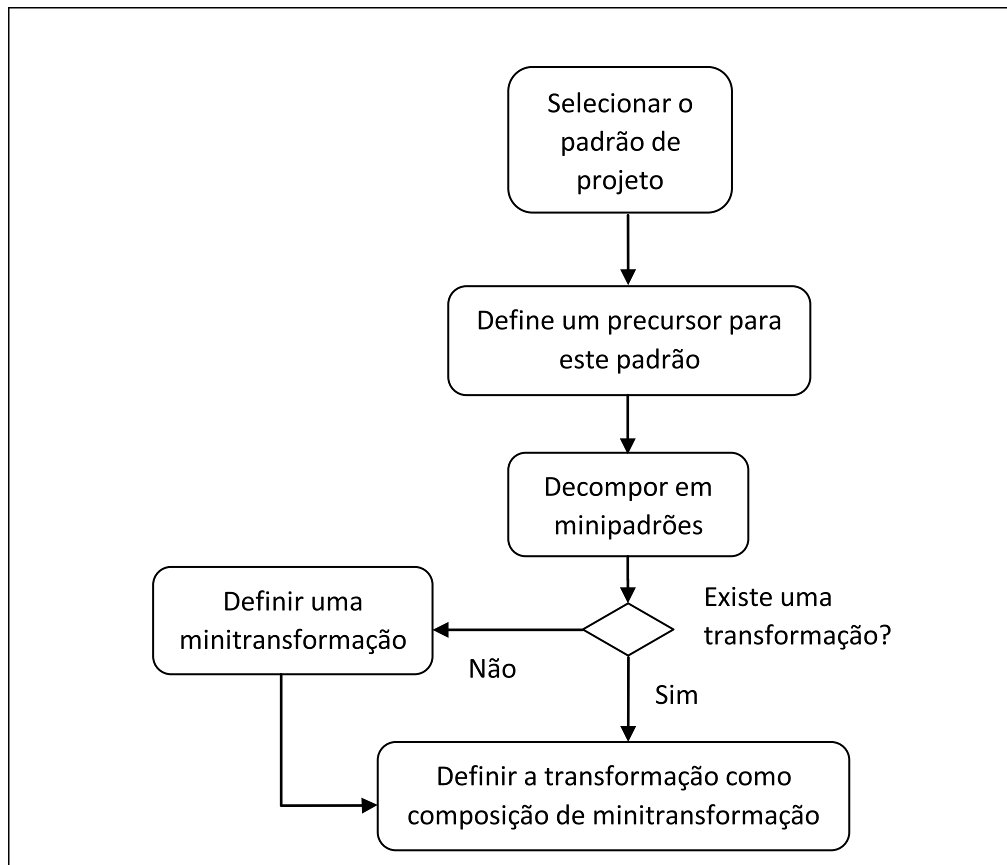

Figura 2: Representação do método de Cinnéide (2001) para refatoração de padrões de projeto.

O *Minipadrão* é um motivo do padrão de projeto que ocorre com frequência. Os minipadrões podem ser combinados de várias maneiras, produzindo diferentes padrões de projetos. Dessa forma, eles são semelhantes a um padrão de projeto, mas em uma construção de baixo nível. Para cada *minipadrão* descobre-se uma *minitransformação* correspondente que pode aplicar este *minipadrão*.

Uma *minitransformação* inclui um conjunto de condições, uma descrição do algoritmo da transformação, e um conjunto de pré e pós-condições. Ela é a unidade de reutilização, portanto, para qualquer *minipadrão* identificado. Deve-se verificar se uma *minitransformação* já foi construída como parte do desenvolvimento na aplicação de padrões em alguma refatoração anterior. Se

assim for, esta *minitransformação* pode ser reutilizada, caso contrário, uma nova *minitransformação* deve ser desenvolvida.

Ao concentrar-se no desenvolvimento de transformações para *minipadrões*, Cinnéide (2001) desenvolveu uma biblioteca de transformações úteis que podem ser reutilizadas sempre que o mesmo *minipadrão* é identificado novamente em um desenvolvimento futuro. A transformação que corresponde a um *minipadrão* é naturalmente chamada de *minitransformação*. Ou seja, na prática o *minipadrão* é o nome e a descrição de cada *minitransformação*.

3.2.1. Especificação de pré e pós-condições

Para tratar de preservação de comportamento são utilizadas pré e pós-condições. As pré e pós-condições e as transformações são representadas e realizadas através de refatorações primitivas e funções de análise e auxiliares:

- *Refatorações primitivas*: vão de refatorações simples, como *addClass*, a até refatorações complexas como *replaceObjCreationWithMethInvocation*, que substitui uma expressão que cria um objeto pela invocação de uma método utilizando uma lista de argumentos. Caso necessário pode-se definir uma nova refatoração primitiva, descrevendo:
    - Nome;
    - Tipo de retorno;
    - Tipos de argumentos;
    - Descrição informal;
    - Pré-condições; e
    - Pós-condições.
- *Funções de análise*: são utilizadas para extrair informações e validar afirmações sobre o programa, como, por exemplo, se uma classe existe ou não. Para a representação de um conjunto de entidades do tipo *X* como interface, classes e argumentos, utiliza-se *SetOfX*.

- *Funções auxiliares*: são utilizadas para fazer extrações mais complexas de um programa a partir das funções de análise, como, por exemplo, *abstractClass* que cria uma interface de uma classe com base em seus métodos públicos.

Em seu trabalho, Cinnéide (2001) apresenta uma lista completa de *minitransformações* utilizadas para aplicação do método em sete padrões de projeto de Gamma et al. (1995) utilizando as refatorações primitivas, funções de análise e funções auxiliares para especificação e preservação de comportamento.

3.2.2. Refatoração para o padrão *Strategy*

Para a aplicação do padrão *Strategy*, Cinnéide (2001) utiliza as minitransformações *Delegation*, *Abstraction* e *AbstractAccess* como apresentado na Figura 3. Primeiro, na minitransformação *Delegation* cria-se uma nova classe com nome *strategyName*, criando um atributo na classe Contexto do mesmo tipo e se move o método da estratégia. Depois se cria uma interface e se faz o relacionamento de realização da interface. Finalmente se substitui qualquer referência na classe Contexto para a classe da estratégia pela interface da estratégia.

```
applyStrategy(Class context, SetOfMethod strategyMethods,
                  String strategyName){
      Delegation(context, strategyMethods, strategyName)
      Abstraction(strategyName, strategyInterface);
      AbstractAccess(context, strategyName, strategyInterface);
}
```

Figura 3: Definição apresentada por Cinnéide (2001) para aplicação do padrão *Strategy*.

No método proposto por Cinnéide (2001) a fase de identificação da oportunidade de refatoração é tratada na fase de identificação do precursor como apresentado na fig. 2. Este é um trabalho manual, que deve ser feito pelo desenvolvedor, sendo inclusive uma das motivações no trabalho (Cinnéide,

2001) – que sempre a identificação e decisão de execução estejam a critério do desenvolvedor.

4. Análise Comparativa

De acordo com Cinnéide (2001) a refatoração para o padrão *Strategy* se apresentou um grande desafio de automatização. Por um lado, trata-se de um padrão comportamental em que se altera a estrutura interna de um algoritmo. Por outro, considera que clientes não necessitam ter conhecimento sobre a implementação de cada estratégia. Estas duas características tornam difícil sua automatização.

Nesta seção, o padrão de projeto *Strategy* é analisado tanto com a aplicação do método de Cinnéide (2001) quanto pela utilização da ferramenta JDeodorant (CHRISTOPOULOU et al., 2012). Estas duas abordagens são comparadas e suas vantagens e desvantagens são apresentadas. Também é avaliado o quanto cada uma delas auxilia a melhoria da qualidade do sistema.

4.1. Contexto

A análise usa como objeto de estudo um sistema de seguros baseado em componentes reutilizáveis e com um modelo de dados dinâmico, configurado por metadados. Este sistema vem sendo utilizado por diversas empresas em vários países da América Latina e da Europa. A interface e o comportamento do sistema podem ser implementados para atender às necessidades de qualquer companhia de seguros. Os produtos de seguros, em geral, sofrem grandes variações de configuração e de comportamento de acordo com parâmetros comerciais e com legislações e regras impostas por órgãos fiscalizadores de seguros de cada país. Esta maleabilidade é atendida por uma grande plataforma de configuração.

O intermediário entre o segurado e a seguradora é conhecido como corretor ou, no caso de seguradoras de banco, agente de contas. É ele quem vende a apólice e por tal transação é acordado um percentual ou valor a ser pago ao corretor. De modo geral este pagamento deve ser feito a diferentes partes: ao

parceiro da seguradora (parceria de negócio entre empresa que vende o produto e a seguradora) e ao corretor (papel obrigatório em qualquer apólice de seguros no Brasil). Esses papéis ou tipos de corretores são tratados muitas vezes como agentes. O processo utilizado para efetuar o pagamento se chama liquidação de agentes.

Dado o tamanho do sistema, para realizar a análise comparativa é necessário definir um escopo. A escolha desse escopo deve considerar as diferenças de abrangência dos métodos: o método de Cinnéide (2001) não trata da identificação de oportunidades de refatoração, ficando ao critério do desenvolvedor a identificação do *precursor*. Por outro lado, a ferramenta de Christopoulou et al. (2012) apresenta uma proposta de identificação da oportunidade de refatoração. Por esse motivo foi utilizada a ferramenta proposta em Christopoulou et al. (2012) para seleção do escopo a ser refatorado.

A classe *AgentLiquidationDAO* possui um método *processLiquidationItems* que tem como parâmetros os tipos *String*, *String*, *String*, *BigInteger*, *BigInteger*, *String*, *String*, *OpenItemSubStatus*, *CommissionType*. O *CommissionType* é uma classe *enum* declarada na própria classe *AgentLiquidationDAO* para definir os tipos de comissões de agentes que serão liquidados. Este método possui também um bloco condicional *switch* apresentado na Figura 4, o qual foi identificado pela ferramenta proposta em Christopoulou et al. (2012) como candidato à refatoração. Por esse motivo e dada a sua relevância ao negócio de seguros, a refatoração do padrão *Strategy* será aplicada para esse código. A simplicidade do código na Figura 4 não diminui sua importância para o negócio, nem os ganhos que podem ser obtidos com sua refatoração para manutenções futuras.

```
switch (process) {
      case updateStatusCommissionDue:
            sql.append("update openitem set status = ?, opm_substatus=?")
            .append("where openitemid in ( ")
            .append("select oi.openitemid ");
            params.add(OpenItemStatus.APPLIED);
            params.add(subStatusUpdate);
            break;
      case updateStatusGroupCommission:
            sql.append("update openitem set status = ?, opm_substatus=?")
            .append("where openitemid in ( ")
            .append("select distinct oi.opm_masteropenitemid ");
            params.add(OpenItemStatus.APPLIED);
            params.add(subStatusUpdate);
            break;
      case createHistoryCommissionDue:
            sql.append("insert into stca_openitemhistory (")
            .append("oih_id, opm_id, oih_auditdate, ")
            .append("oih_description, oih_status, oih_substatus) ")
            .append("select SQ_STCA_OPENITEMHISTORY.nextval, ")
            .append("oi.openitemid, to_date('")
            .append(sysdate + "','yyyy-mm-dd hh24:mi:ss'), ")
            .append("'openItemHistoryStatus.blank', ")
            .append("'" + OpenItemStatus.APPLIED + "', '")
            .append(subStatusUpdate + "' ");
            break;
      case createHistoryGroupCommission:
            sql.append("insert into stca_openitemhistory (")
            .append("oih_id, opm_id, oih_auditdate, ")
            .append("oih_description, oih_status, oih_substatus) ")
            .append("select SQ_STCA_OPENITEMHISTORY.nextval, ")
            .append("openitemid, to_date('")
            .append(sysdate + "','yyyy-mm-dd hh24:mi:ss'), ")
            .append("'openItemHistoryStatus.blank', ")
            .append("'" + OpenItemStatus.APPLIED + "', '")
            .append(subStatusUpdate + "' ")
            .append("from openitem where openitemid ")
            .append("in ( select distinct oi.opm_masteropenitemid ");
            break;
}
```

Figura 4: trecho do código onde será aplicado o padrão *Strategy*.

4.2. Critérios de Análise

Na seção 2.1 foram apresentadas as 6 atividades propostas por Mens e Tourwé (2004) para aplicação da refatoração. Neste trabalho algumas dessas atividades estão ligadas diretamente à proposta de automação da refatoração. Será adotado como atividades *Aplicar a refatoração* e *Avaliar o efeito da refatoração com relação a características de qualidade* – as demais atividades estão implícitas nas abordagens da automatização da refatoração. A

identificação da refatoração utilizou a ferramenta de Christopoulou et al. (2012), como citado anteriormente. Para a aplicação da refatoração foram utilizadas as abordagens Christopoulou et al. (2012) e Cinnéide (2001).

Em ambos os casos, a análise dos resultados será feita com base nos seguintes critérios:

- *Complexidade ciclomática de McCabe (1976)*: que calcula a quantidade de caminhos independentes a partir de um código fonte, medindo assim sua complexidade.
- *SLOC (Sources line of code)*: para medir a quantidade de linhas no projeto analisado (PARK, 1992).
- *MLOC* (*Method lines of code*): para medir a quantidade de linhas de código por método.
- *Métricas CK (CHIDAMBER; KEMERER, 1994)*: fornece um conjunto de métricas de software para projetos orientados a objetos para auxiliar no processo de desenvolvimento de software.

O fato de o padrão de projeto encapsular algoritmos e torná-los intercambiáveis de acordo com a estratégia desejada por seus clientes deveria reduzir a complexidade ciclomática que é afetada diretamente pela quantidade de fluxos alternativos no código fonte (MCCABE, 1976). Também foram selecionadas as métricas *SLOC* e *MLOC*. Como detalhado no padrão *Strategy* (GAMMA et al., 1995), a *SLOC* pode aumentar em razão da criação das classes concretas para cada estratégia. Por sua vez, o *MLOC* pode diminuir na medida em que a lógica de cada algoritmo é transferida para as classes concretas de cada estratégia. Essas três métricas também foram utilizadas no trabalho de Christopoulou et al. (2012) para avaliar os resultados da aplicação da refatoração para o padrão *Strategy* em alguns projetos de código aberto.

Das métricas CK (CHIDAMBER; KEMERER, 1994) são utilizadas as seguintes métricas:

- *Métodos Ponderados por Classe (WMC)*: valor obtido através do somatório das complexidades dos métodos de uma determinada classe.

- *Profundidade da Árvore de Herança (DIT)*: valor obtido através da profundidade de herança de uma classe. Nos casos que envolvem a herança múltipla, a DIT é o comprimento máximo da classe até a raiz da árvore.
- *Número de Filhos (NOC)*: obtido através da quantidade de subclasses imediatas de uma determinada classe.

Essas métricas foram selecionadas porque auxiliam a medição qualitativa e quantitativa de quanto o código fonte está aderente aos princípios de orientação a objetos. Elas possibilitam também a comparação dos resultados da aplicação de ambas refatorações através da aplicação do padrão *Strategy*, permitindo analisar a superioridade de uma das abordagens sobre a outra. Para uma análise mais sucinta da aplicação do padrão *Strategy*, foi selecionada apenas a classe *AgentLiquidationDAO* como objeto de análise geral, e somente as métricas *SLOC* e *MLOC* serão analisadas no projeto de forma geral. Isto possibilitará a análise do impacto na aplicação da refatoração em termos de quantidades de linhas sobre o projeto.

Em relação às demais métricas CK (CHIDAMBER; KEMERER, 1994), o *Acoplamento entre Classes de Objetos (CBO)* e *Resposta de Classe (RFC)* não serão utilizadas, pois elas consideram diversas classes de um projeto – e a análise em questão será focada apenas na classe *AgentLiquidationDAO*. A métrica *Falta de Coesão em Métodos* (LCOM), não será discutida, pois a classe analisada possui apenas um método e o resultado dessa métrica considera quantidade de atributos de uma classe que são modificados por métodos da mesma classe de maneira independente.

Por fim, como ferramenta para medir as métricas, através dos resultados da aplicação da refatoração foi utilizada o plug-in Metrics (2013) do Eclipse.

4.3. Aplicação da ferramenta

Ao término da execução da refatoração pela ferramenta, obtiveram-se diversas mudanças. Foram criadas 5 classes, uma abstrata que contém a interface da estratégia e 4 que representam cada condicional do *switch*. A lógica condicional foi totalmente substituída, eliminando as instruções condicionais e as trocando por um método de invocação polimórfica em uma referência da classe abstrata *PreparedProcess*. Essa classe abstrata foi gerada pela refatoração e representa a interface da estratégia (GAMMA et al., 1995) que contém o método abstrato *processLiquidationItems*, cujos parâmetros são as variáveis antes localizadas no bloco *switch* fig. 4. Ao ser executado através de uma instância de *CreateHistoryGroupCommission,* por exemplo, o método *processLiquidationItems* adiciona à pesquisa a respectiva instrução para geração da comissão correta. Com isso, o código da Figura 4, com a declaração do bloco *switch*, foi substituído pelo código representado na Figura 5.

```
getPreparedProcessObject(process).processLiquidationItems(
                            subStatusUpdate, sql, params, sysdate);
```

Figura 5: Trecho de código após a aplicação da refatoração.

Adicionalmente, um novo método foi criado na classe *AgentLiquidationDAO*. Este método, chamado de *getPreparedProcessObject*, recebe como parâmetro a instância do *enum CommissionType* e retorna uma instância da estratégia como, por exemplo, *CreateHistoryGroupCommission*.

4.4. Aplicação do método

Com a aplicação dos passos propostos pelo método proposto por Cinnéide (2001) (apresentados na Seção 3.2), percebe-se que não existiria uma completa aplicação do padrão *Strategy* como sugerido por Gamma et al. (1995), em que são solicitados três participantes: *Contexto*, *IEstrategia* e as

implementações da estratégia que seriam as *EstrategiaConcreta*. Da forma apresentada pelo método Cinnéide (2001), apenas utiliza-se um *moveMethod* para mover *por inteiro* o método candidato para a nova classe que faria o papel de *EstrategiaConcreta*, cria-se uma interface para esta classe e altera-se as referências da nova classe para referenciar a interface que estiver sendo utilizada na classe *Contexto*. Apesar de se ter todos os participantes, isso não contempla todas as características de cada participante e vai contra algumas características do padrão *Strategy* (GAMMA et al., 1995) como:

- A utilização do padrão *Strategy* é indicada apenas quando a variação de comportamento é relevante para os clientes. Da forma apresentada pelo método, apenas se estaria delegando o método que contém os algoritmos para a classe da estratégia. Apesar de o trabalho referir-se a *SetOfMethod* como um conjunto de métodos, como apresentado em funções de análise na seção 3.2.1, mesmo que se considerado como um conjunto de algoritmos para cada estratégia, todos iram ser movidos para mesma classe.
- A classe *Contexto* deve ser configurada e manter referência a apenas uma estratégia, usando a interface.

De acordo com os problemas identificados anteriormente, foi necessário criar uma nova minitransformação para uma completa aplicação do padrão *Strategy* e que atenda a todas as características sugeridas por Gamma et al. (1995). Com isso foi feita a definição de uma nova minitransformação, chamada de *DelegationBlock* e apresentada na Figura 6, para ser chamada após a minitransformação *Delegation* do método proposto por Cinnéide (2001). Ao extrair o método da estratégia, esta minitransformação define um método abstrato na classe *Delegation*. Em seguida, ela extrai cada bloco de código de acordo com cada constante que define a estratégia. Então é criada uma classe para cada constante e se move o algoritmo da estratégia para um método desta classe. Este método é criado pela refatoração primitiva *addClass* que recebe o nome da classe a ser criada e qual superclasse (CINNÉIDE, 2001). Caso a classe tenha método abstrato, esse método é implementado na classe concreta em criação.

Nome: DelegationBlock

Argumentos:

String delegationName: nome da classe criada para fazer o papel da estratégia.

SetOfMethod moveMethods: método movido para classe recém criada de onde será lido para extrair as classes concretas.

SetOfVariable constants: família de tipos de onde são retirados quais blocos são os algoritmos específicos para cada classe.

Descrição: Implementar o algoritmo na classe concreta para cada estratégia segundo a interface.

Algoritmo:

```
Method m = abstractMethod(moveMethods);
renameMethod(Abstract, m);
addMethod(delegationName, m);
ForAll block:Constants, typeOf(block) = Constants {
     addClass(Constants, delegationName);
     moveBlock(delegationName, Constants, m, block);
}
```

Figura 6: Definição da nova minitransformação *DelegationBlock* para uma completa aplicação do padrão *Strategy*.

Para aplicar o método foi usado como precursor a classe *AgentLiquidationDAO* que tem várias implementações para lidar com diferentes tipos de liquidação de agentes. A classe de contexto, *AgentLiquidationDAO*, não necessita ter conhecimento de como é efetuada a liquidação de cada tipo de agente.

Após a criação da nova minitransformação apresentada na Figura 6, foi aplicada a refatoração para o padrão de projeto *Strategy* de acordo com a seqüência de minitransformações propostas em Cinnéide (2001), estendida pela proposta deste trabalho. Ao término da execução do método, observaram-se diversas transformações. Inicialmente uma nova classe que foi chamada de *ProcessCommissionAbstract*, onde ficou o método para configuração da estratégia e a declaração do método abstrato para estratégia. Adicionalmente foram criadas classes de estratégia representada cada uma das constantes que separavam as lógicas no *switch* anteriormente declarados pela classe *AgentLiquidationDAO*. No lugar de toda a lógica condicional na classe cliente *AgentLiquidationDAO*, obtém-se a estrutura apresentada na Figura 7:

```
delegation =
   ProcessCommissionAbstract.createProcessCommission(process);
delegation.processLiquidationItems(subStatusUpdate, sql, params,
   sysdate);
```

Figura 7: Bloco na classe *AgentLiquidationDAO* que configura e invoca a estratégia.

4.5. Análise dos resultados

Na

Tabela 1 são apresentados os valores das métricas analisadas para o método e para a ferramenta analisada.

Tabela 1: Resultado da aplicação das métricas.

Métrica	Antes da refatoração	Refatoração com JDeodorant	Refatoração com o método
Complexidade Ciclomática (*McCabe*)	16	12	12
Profundidade da Árvore de Herança (*DIT*)	2	2	2
Métodos Ponderados por Classe (*WMC*)	17	18	13
Número de Filhos (*NOC*)	0	0	0
Linhas de Código (*SLOC*)	164	203	220
Linhas de Código por Método (*MLOC*)	87	96	96

A métrica Complexidade Ciclomática (MCCABE, 1976), que mede a complexidade por método, obteve o valor 12 para ambas as aplicações. Esses valores foram calculados considerando o método na classe *AgentLiquidationDAO* que tinha a lógica condicional e que foi substituído pela invocação da estratégia. Já com a métrica Métodos Ponderados por Classe (WMC) que se trata da soma da complexidade ciclomática (MCCABE, 1976) de

toda a classe, ela sofreu um aumento na aplicação da refatoração com da ferramenta. Além do método com a invocação da estratégia (com complexidade 12), a ferramenta criou um novo método para configuração da estratégia com complexidade 5 na própria classe de Contexto, além do construtor com complexidade 1, somando 18 ou invés do valor anterior de 17. Em contraposição, a refatoração através do método obteve o valor 13 incluindo o valor do construtor, separando o método de configuração da estratégia na classe abstrata *ProcessCommissionAbstract*. Ou seja, tratando-se da métrica Métodos Ponderados por Classe (WMC), o método demonstrou-se mais eficiente que a ferramenta, pois, diminuiu 23,5% da complexidade da classe da classe em análise.

No caso da métrica Profundidade da Árvore de Herança (DIT), ela permaneceu em todos os casos com valor 2. Isso aconteceu porque não houve alteração da posição da classe *AgentLiquidationDAO* na árvore de herança. Igualmente a métrica Número de Filhos (NOC) permaneceu com valor 0 em todos os casos, pois em todos nenhuma classe herda a classe *AgentLiquidationDAO* em análise.

Como dito nos critérios de análise, na seção 4.2, das métricas CK apenas os critérios de número de Linhas de Código (SLOC) e número de Linhas de Código por Método (MLOC) serão analisados, por sua abrangência em todo o projeto. Ambas as aplicações causaram um aumento no número de linhas, o que já era esperado a qualquer aplicação do padrão *Strategy* (GAMMA, 1995).

Por um lado, na refatoração com a ferramenta (CHRISTOPOULOU et al., 2012) o método de configuração da estratégia ficou na própria classe *AgentLiquidationDAO*, enquanto que a classe abstrata *PreparedProcess*, gerada pela ferramenta, continha apenas a interface do método abstrato *processLiquidationItems*. Esse método foi implementado por cada estratégia concreta. Por outro lado, na refatoração com o método (CINNÉIDE, 2011) criou uma nova interface *IProcessCommission* onde ficou o método *processLiquidationItems* para ser implementado por sua imediata classe concreta. Além do método de configuração da estratégia, também foi redefinido

o método abstrato *processLiquidationItems* na classe abstrata *ProcessCommissionAbstract*. A aplicação do método foi a que causou maior aumento do número de linhas no código fonte. Por fim, o fato de o método de configuração da estratégia pela aplicação do método ficar na classe abstrata da estratégia e não na classe *AgentLiquidationDAO* de contexto, como foi feito pela ferramenta, fez com que diminuísse bastante a complexidade da classe *AgentLiquidationDAO*.

Observou-se também a facilidade de utilização da ferramenta. Com apenas um clique do botão "*Apply Refactoring*", os nomes da classe abstrata e das implementações da estratégia já são sugeridos para a refatoração selecionada. Com mais um clique já se tem a refatoração aplicada. Em contraposição, no método proposto por Cinnéide (2001) essas atividades são totalmente manuais. De fato, de acordo com Christopoulou et al. (2012), esta facilidade da automatização da refatoração para padrões de projeto permite contribuições significativas para qualidade do projeto, mesmo para desenvolvedores com pouco ou nenhuma experiência prática com o uso de padrões de projeto. Esta vantagem se mantém, ainda que no mesmo trabalho seja realizada uma validação quanto a "falso negativo", em que a ferramenta apresentada rejeita oportunidades de refatoração que, segundo a avaliação de engenheiros de software, é considerada uma oportunidade de utilização do padrão *Strategy*.

Além das limitações das abordagens apresentadas nos seus respectivos trabalhos, outras limitações foram também notadas durante os testes realizados. O método de Cinnéide (2001) tem como limitações: não possibilita refatoração para códigos que fazem uso de reflexão, assumindo que todos os objetos são criados apenas usando operador *new* e que classes privadas não são consideradas pelo trabalho. O trabalho de Christopoulou et al. (2012), também não possibilita a refatoração em códigos que fazem uso de reflexão. Adicionalmente, o código a ser refatorado não pode ser parte de um método redefinido de uma classe que não é do sistema, como, por exemplo, métodos de outros projetos da empresa ou de códigos de terceiros. Isto faz com que o método Cinnéide (2001) seja mais geral.

Como conclusão sobre a automatização de refatoração para padrões de projetos, embora a ferramenta de Christopoulou et al. (2012) efetue a refatoração de maneira completamente automatizada para o padrão *Strategy*, o método de Cinnéide (2001) se demonstrou superior especificamente na aplicação do *Strategy* em redução da complexidade. Indiretamente, isto também causa uma melhor legibilidade e facilidade de manutenção do código fonte. Outra vantagem é que o método permite tanto criar novas minitransformações quanto reutilizar as existentes, quando necessário. Por sua vez, observou-se que a aplicação da ferramenta de Christopoulou et. al. (2012) altera o código fonte de maneira intrinsecamente particular a um padrão. Esta característica demanda que todo o algoritmo seja refeito caso se necessite fazer a refatoração para outro padrão. Esta limitação não é encontrada no método de Cinnéide (2001), que, ao contrário, se mostra flexível.

5. CONCLUSÃO

O objetivo deste trabalho foi realizar uma análise comparativa entre as abordagens de Christopoulou et al. (2012) e de Cinnéide (2001) de automatização de refatoração para o padrão de projeto *Strategy.* Nesta análise, foi possível verificar quais das abordagens apresentaram melhores resultados em termos de melhorias no código fonte e melhor qualidade em termos de propostas de automatização de refatoração para padrões de projetos. Para isso, usou-se como base um sistema de gestão de seguros em um módulo de pagamento de comissões.

5.1. Resumo

O trabalho de Cinnéide (2001) foi considerado superior, por apresentar uma maior flexibilidade em termos de automatização de refatoração para padrão de projeto. Além de propor um método de criação de transformações para atingir um padrão de projeto, o trabalho também obteve um melhor resultado conforme as métricas selecionadas, diminuindo a complexidade do código fonte.

Constatou-se, adicionalmente, que novos métodos e ferramentas de automatização têm surgido. Porém, verificou-se também que este campo de pesquisa ainda necessita evoluir na identificação confiável de oportunidades de refatoração, em sua aplicação automatizada e na verificação efetiva da melhoria de qualidade de um sistema.

Adicionalmente, para mensurar quão abrangente é uma abordagem ou uma ferramenta, seria importante que elas deixassem mais claro quais atividades (MENS; TOURWÉ, 2004) serão executadas para a refatoração.

5.2. Contribuições

Como principais contribuições deste trabalho, pode-se destacar a extensão que foi criada para o método de Cinnéide (2001) para refatoração do padrão de projeto *Strategy*. Destaca-se, também, a análise das abordagens de Cinnéide (2001) e de Christopoulou et al. (2012) e os impactos positivos e negativos da aplicação de ambas refatorações com o padrão de projeto *Strategy*.

Observou-se, também, uma necessidade de evolução da automatização de refatoração para padrões de projetos de modo geral.

Este trabalho mostrou alguns pontos importantes a serem considerados ao utilizar automatização de refatoração para padrões de projetos. Entre eles estão analisar quais atividades (MENS; TOURWÉ, 2004) de refatoração serão atendidas, qual nível de automatização será utilizado na refatoração e verificar qual a real contribuição de uma refatoração ao sistema. Isto é importante, pois, como visto neste trabalho, uma pequena refatoração a mais ou de forma diferente pode trazer relevantes ganhos ao sistema.

5.3. Trabalhos futuros

Como trabalho futuro é possível considerar a criação de uma ferramenta de automatização de refatoração que aplique o método proposto por Cinnéide (2001).

Outra área que tem, aparentemente, muitas possibilidades de evolução é a análise de estudos que proponham a identificação de oportunidades de refatoração para padrões de projeto.

Outra área a ser considerada é a análise da automatização da refatoração para outros padrões de projetos ou, até mesmo, para padrões arquiteturais.

REFERÊNCIAS

ABRAN, A. MOORE, J. W. BOURQUE, P. DUPUIS, R. **Guide to the Software Engineering Body of Knowledge, SWEBOK**: 2004 Version. 200 pages. IEEE Computer Society Order Number C2330, ISBN 0-7695-2330-7. 2005.

CHIDAMBER, S.R. KEMERER, C.F. **A Metrics Suite for Object Oriented Design.** IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June, 1994.

CHRISTOPOULOU, A. GIAKOUMAKIS, E. A. ZAFEIRIS, V. SOUKARA, V. **Automated refactoring to the Strategy design pattern**. Information & Software Technology 54 (11): 1202-1214, 2012.

CINNÉIDE, M Ó. **Automated Application of Design Patterns: A Refactoring Approach.** 2001.

THE ECLIPSE FAUNDATION, Disponível em: <http://www.eclipse.org/>. Ambiente Integrado de Desenvolvimento **Eclipse.** Acessado em 22 jun. 2013.

FOWLER, M. **Refactoring: Improving the Design of Existing Code.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

GAMMA, E. HELM, R. JOHNSON, R. VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

HERBOLD, S. GRABOWSKI, J. NEUKIRCHEN, H. **Automated Refactoring Suggestions Using the Results of Code Analysis Tools**. First International Conference on Advances in System Testing and Validation Lifecycle, IEEE, 2009.

HILL, M. E. PARNIN, C. BLACK, A. P. **How We Refactor, and How We Know It.** IEEE Trans. Software Eng. Vol. 38 No. 1 Pg. 5-18. 2012.

JET BRAINS, Disponível em: <http://www.jetbrains.com/idea/>. Ambiente Integrado de Desenvolvimento **IntelliJ IDE.** Acessado em 22 jun. 2013.

JDEODORANT. Disponível em <http://java.uom.gr/~jdeodorant/>. Plug-in **JDeodorant** para Eclipse IDE. Acessado em 10 mar. 2013.

JDEVAN, Disponível em: <http://webdocs.cs.ualberta.ca/~stroulia/Zhenchang_Xing_Old_Home/jdevan.html>. Acessado em 22 jun. 2013.

JEBELEAN, C. CHIRILA, C. B. CRETU, V. **A logic based approach to locate composite refactoring opportunities in object-oriented code.** International Conference on Automation, Quality and Testing, Robotics, pp. 1-6, IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2010.

JEON, S. U. LEE, J. S. BAE, D. H. **An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs.** Asia-Pacific Software Engineering Conference, p. 337, 9th Asia-Pacific Software Engineering Conference (APSEC'02), 2002.

JTRANSFORMER, Disponível em: <https://sewiki.iai.uni-bonn.de/research/jtransformer/start>. Plug-in **JTransformer** para Eclipse IDE. Acessando em 23 jun. 2013.

KEEFFE, M. O. CINNÉIDE, M. Ó. **Search-based refactoring for software maintenance**, Journal of Systems and Software, vol. 81, no. 4, pp. 502–516, 2008.

KERIEVSKY, J. **Refactoring to Patterns.** Addison-Wesley Professional; 2004.

KHATCHADOURIAN, R. SAWIN, J. ROUNTEV, A. **Automated Refactoring of Legacy Java Software to Enumerated Types.** ICSM Pg. 224-233. 2007.

MCCABE, T. J. **A complexity measure.** IEEE Trans. Software Eng. Vol. 2 No. 4 Pg. 308-320. 1976.

MENS, T. TOURWÉ, T. **A Survey of Software Refactoring.** IEEE Transactions on Software Engineering, vol. 30, no. 2, Pg. 126-139, Feb. 2004.

METRICS. Disponível em: <http://metrics.sourceforge.net/>. Plug-in **Metrics** para Eclipse IDE. Acessado em: 29 jun. 2013.

MOGHADAM, I. H. CINNÉIDE, M. Ó. **Automated Refactoring Using Design Differencing.** CSMR Pg. 43-52. 2012.

ORACLE CORPORATION, Disponível em: <http://netbeans.org/>. Ambiente Integrado de Desenvolvimento **NetBeans.** Acessado em 22 jun. 2013.

INDUSTRIAL LOGIC, Refactoring to Patterns, Disponível em <http://industriallogic.com/xp/refactoring/>. Acessado em 17/09/2013.

OPDYKE, W.F. **Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.** PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

ORACLE CORPORATION, **Code Conventions for The Java Programming Language**, Disponível em: <http://www.oracle.com/technetwork/java/codeconv-138413.html>. Acessando em 26 jun. 2013.

PARK, R. **Software Size Measurement: A Framework for Counting Source Statements.** (CMU/SEI-92-TR-020). Software Engineering Institute, Carnegie Mellon University, 1992.

VAKILIAN, M. CHEN, N. NEGARA, S. RAJKUMAR, B. A. BAILEY, B. P. JOHNSON, R. E. **Use, disuse, and misuse of Automated Refactorings.** ICSE Pg. 233-243. 2012.