UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO

DISSERTAÇÃO DE MESTRADO

Guinther de Bitencourt Pauli

Santa Maria, RS, Brasil

2014

BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO

Guinther de Bitencourt Pauli

Dissertação apresentada ao Curso de Mestrado do Programa de
Pós-Graduação em Informática 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

2014

Pauli, Guinther de Bitencourt
Busca por oportunidades de refatoração para aplicação de padrões de projeto / Guinther de Bitencourt Pauli.- 2014.
97 p.; 30cm

Orientador: Eduardo Kessler Piveta
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2014

1. Programação Orientada a Objetos 2. Padrões de Projeto 3. Refatoração I. Piveta, Eduardo Kessler II. Título.

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

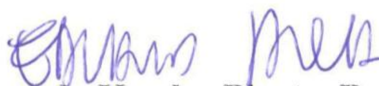
A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO

elaborada por
Guinther de Bitencourt Pauli

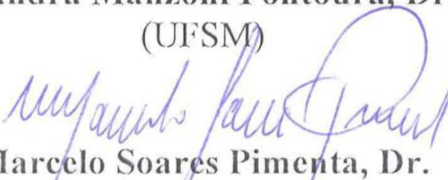
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:



Eduardo Kessler Piveta, Dr.
(Presidente/Orientador)

Lisandra Manzoni Fontoura, Dr^a^.
(UFSM)



Marcelo Soares Pimenta, Dr.
(UFRGS)

Santa Maria, 27 de agosto de 2014.

AGRADECIMENTOS

Meus agradecimentos a todos que contribuíram direta ou indiretamente para o desenvolvimento deste trabalho.

Ao meu orientador, professor Eduardo Kessler Piveta, por todo o apoio prestado durante esta jornada, fornecendo ideias, sugestões, comentários e orientações para a conclusão deste projeto. Sua capacidade de motivar as pessoas, sua simplicidade em resolver problemas e a segurança sempre passada foram fundamentais para que os objetivos fossem alcançados.

Aos meus amigos do Grupo de Pesquisa em Linguagens de Programação e Bancos de Dados da UFSM, Cristiano e Jânio, o meu muito obrigado pelas longas horas de discussão e ideias trocadas sobre padrões.

Aos meus pais Ivo e Rosa, professores por formação e projeto de vida, agradeço a distinta educação recebida, baseada em princípios e valores que cultivam o dom que há de mais importante em um ser humano: o conhecimento.

À minha esposa Fernanda (Nina), pela sua compreensão, apoio e carinho durante esta longa jornada.

“*Any fool can write code that a computer can understand.*

Good programmers write code that humans can understand.”

(MARTIN FOWLER)

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO
AUTOR: GUINTHER DE BITENCOURT PAULI
ORIENTADOR: EDUARDO KESSLER PIVETA
Data e Local da Defesa: Santa Maria, 27 de agosto de 2014.

Manter e adaptar código mal escrito, com problemas estruturais, é uma tarefa difícil. As técnicas de refatoração são usadas para melhorar o código e a estrutura de aplicações, aumentando sua qualidade e tornando-as mais fáceis de serem modificadas. Padrões de projeto são soluções reutilizáveis usadas para resolver problemas comumente encontrados em sistemas orientados a objetos. A aplicação de padrões de projeto no contexto de refatoração usando uma abordagem corretiva torna-se uma atividade interessante no ciclo de vida de um sistema de software. Contudo, para projetos de média e larga-escala, examinar manualmente os artefatos de software na busca por problemas e oportunidades para aplicar um determinado padrão de projeto é uma tarefa árdua e difícil. Nesse contexto, apresentamos um conjunto de funções heurísticas baseadas em métricas que permitem detectar onde um padrão de projeto pode ser aplicado nos artefatos de código de um determinado projeto, mais especificamente, os padrões de projeto *Strategy*, *Factory Method*, *Template Method*, *Creation Method* e *Chain Constructors*. Para avaliar as funções propostas, desenvolvemos uma ferramenta que avalia código-fonte e exibe possíveis oportunidades para aplicar uma refatoração rumo a tais padrões. Essa ferramenta foi desenvolvida usando-se ASTs (*Abstract Syntax Trees*), procurando por oportunidades de refatoração, indicando os locais no código fonte onde o padrão é sugerido e mostrando algumas evidências usadas para a detecção.

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

ABSTRACT

Master Course Dissertation
Professional Graduation Program in Informatics
Universidade Federal de Santa Maria

SEARCHING FOR REFACTORING OPPORTUNITIES TO APPLY DESIGN PATTERNS
AUTHOR: GUINTHER DE BITENCOURT PAULI
ADVISER: EDUARDO KESSLER PIVETA
Defense Place and Date: Santa Maria, August 27th, 2014.

It is difficult to maintain and to adapt poorly written code presenting shortcomings in its structure. Refactoring techniques are used to improve the source code and the structure of applications, making them better and easier to modify. Design patterns are reusable solutions used in similar problems in object-oriented systems, so there is no need to recreate the solutions. Applying design patterns in the context of refactoring in a corrective way becomes a desired activity in the life cycle of a specific software system. However, for medium and large-scale projects, the manual examination of artifacts to find problems and opportunities to apply a design pattern is a hard task. In this context, we present a set of metric-based heuristic functions to detect where a design pattern can be applied in a given project, more specifically, the Strategy, Factory Method, Template Method, Creation Method and Chain Constructors patterns. To evaluate the heuristic functions and its results we have also built a tool to show the results. This tool can examine source code traversing ASTs (Abstract Syntax Trees), searching for opportunities to apply the patterns, indicating the exact location in the source code where the pattern is suggested, also showing some evidences used in the detection.

Keywords: Design Patterns. Refactoring. Refactoring to Patterns.

LISTA DE FIGURAS

Figura 1 – Estrutura do Padrão de Projeto Strategy (GAMMA et al., 1999) 22
Figura 2 – Estrutura do Padrão de Projeto Factory Method (GAMMA et al., 1999)............... 24
Figura 3 – Estrutura do Padrão de Projeto Template Method (GAMMA et al., 1999)........... 25
Figura 4 – Estrutura do Padrão de Projeto Creation Method (KERIEVSKY, 2008)............... 27
Figura 5 – Estrutura do Padrão de Projeto Chain Constructors (KERIEVSKY, 2008) 27
Figura 6 – Substituir Lógica Condicional por Strategy, exemplo antes da refatoração 30
Figura 7 –Substituir Lógica Condicional por Strategy, exemplo depois da refatoração 30
Figura 8 –Introduzir Criação Polimórfica com Factory Method, exemplo antes da refatoração 31
Figura 9 – Introduzir Criação Polimórfica com Factory Method, exemplo depois da refatoração 32
Figura 10 –Formar Template Method, exemplo antes da refatoração.................................... 33
Figura 11 – Formar Template Method, exemplo depois da refatoração 34
Figura 12 – Substituir Construtores por Métodos de Criação, exemplo antes da refatroração 35
Figura 13 – Substituir Construtores por Métodos de Criação, exemplo depois da refatoração36
Figura 14 –Encadear Construtores, exemplo antes da refatoração.. 37
Figura 15 – Encadear Construtores, exemplo depois da refatoração 37
Figura 16 – Atividades envolvidas no processo de buscas por oportunidades de refatoração. 42
Figura 17 – Análise da complexidade condicional com base no tamanho do bloco, número de testes e uso de “*type codes*”.. 44
Figura 18 – Análise do bloco switch usando “type codes” para criar subclasses 45
Figura 19 - Comparação de pilhas de chamadas de métodos para detectar uma oportunidade para aplicar o Template Method (Exemplo 1)... 46
Figura 20 - Comparação de pilhas de chamadas de métodos para detectar uma oportunidade para aplicar o Template Method (Exemplo 2)... 47
Figura 21 – Análise dos construtores considerando número de sobrecargas e parâmetros...... 48
Figura 22 – Análise de construtores com código de inicialização duplicado.......................... 49
Figura 23 – AROS: uma ferramenta para buscas por oportunidades de refatoração para aplicação de padrões de projeto.. 52
Figura 24 – Framework com classes para buscar oportunidades de refatoração...................... 53
Figura 25 – Função para detectar oportunidades para aplicar o padrão Strategy..................... 54
Figura 26 - Função para detectar oportunidades para aplicar o padrão Factory Method 55
Figura 27 - Função para detectar oportunidades para aplicar o padrão Template Method 56
Figura 28 - Função para detectar oportunidades para aplicar o padrão Creation Method........ 57
Figura 29 - Função para detectar oportunidades para aplicar o padrão Chain Constructors.... 58
Figura 30 – Sentença Switch no NUnit, uma oportunidade para aplicar um padrão Strategy . 64
Figura 31 – Código após a refatoração Substituir Lógica Condicional por Strategy 65
Figura 32 – Diagrama de classes UML após aplicação da refatoração para o padrão Strategy 65
Figura 33 – Sentença Switch no NUnit, porém não é uma oportunidade vantajosa para aplicar o padrão Strategy 67
Figura 34 –Switch no NAnt criando subclasses, oportunidade para aplicar o padrão Factory Method.. 68
Figura 35 – Código refatorado após a aplicação do padrão Factory Method.......................... 69
Figura 36 – Diagrama de classes após refatoração para o padrão Factory Method 70

Figura 37 – Código duplicado no NAnt, uma oportunidade para aplicar o padrão Template Method.....72
Figura 38 – Diagrama de classes para o estudo de caso no NAnt, com código duplicado em subclasses 73
Figura 39 – Código refatorado após a aplicação do padrão Template Method.....74
Figura 40 – Diagrama de classes para do exemplo no NAnt, com aplicação do padrão Template Method, reduzindo a duplicação de código.....75
Figura 41 – Sobrecargas de construtores no NAnt, uma oportunidade para aplicar o padrão Creation Method.....77
Figura 42 – Código refatorado após a aplicação do padrão Creation Method: métodos expressando melhor a intenção.....77
Figura 43 – Construtores com código duplicado no NHibernate, uma oportunidade para aplicar o padrão Chain Constructors 80
Figura 44 – Código refatorado: construtores encadeados repassando parâmetros na assinatura sem duplicar código na implementação.....80

LISTA DE TABELAS

Tabela 1 – Projetos utilizados na avaliação 60
Tabela 2 – Resultados da avaliação das buscas pelo padrão Strategy 61
Tabela 3 – Resultados da avaliação das buscas pelo padrão Factory Method 67
Tabela 4 – Resultados da avaliação das buscas pelo padrão Template Method 70
Tabela 5 – Resultados da avaliação das buscas pelo padrão Creation Method 76
Tabela 6 – Resultados da avaliação das buscas pelo padrão Chain Constructors 78

LISTA DE ANEXOS

Anexo A – Catálogo de Bad Smells – Limitações - em código-fonte 86
Anexo B – Catálogo de Padrões de Projeto (GoF) 90
Anexo C – Catálogo de Refatorações para Padrões de Projeto 93

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AROS	*Automatic Refactoring Opportunity Search*
AST	*Abstract Syntax Tree*
CC	*Complexidade Ciclomática*
GoF	*Gang of Four*
IDE	*Integrated Development Environment*
IM	*Indice de Manutenção*
LOCC	*Linhas de Código de Classe*
OO	*Object-Orientation* (Orientação a Objetos) / *Object-Oriented* (Orientado(a) a Objetos)
POO	Programação Orientada a Objetos
RAD	*Rapid Application Development*
UI	*User Interface*
UML	*Unified Modeling Language*
XP	*eXtreme Programming*
XML	*eXtensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO ... 15
2. REVISÃO DE LITERATURA ... 19
2.1. Padrões de Projeto ... 19
2.1.1. Strategy ... 20
2.1.2. Factory Method ... 22
2.1.3. Template Method ... 24
2.1.4. Creation Method ... 26
2.1.5. Chain Constructors ... 27
2.2. Refatoração ... 28
2.2.1. Substituir Lógica Condicional por Strategy ... 29
2.2.2. Introduzir Criação Polimórfica com Factory Method ... 30
2.2.3. Formar Template Method ... 32
2.2.4. Substituir Construtores por Métodos de Criação ... 34
2.2.5. Encadear Construtores ... 36
2.3. Oportunidades de Refatoração ... 37
2.4. Trabalhos Relacionados ... 38
3. BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO ... 41
3.1. Definição das atividades para a criação de heurísticas de busca ... 41
3.2. Buscando oportunidades para aplicar o padrão Strategy ... 43
3.3. Buscando oportunidades para aplicar o padrão Factory Method ... 44
3.4. Buscando oportunidades para aplicar o padrão Template Method ... 45
3.5. Buscando oportunidades para aplicar o padrão Creation Method ... 47
3.6. Buscando oportunidades para aplicar o padrão Chain Constructors ... 48
4. IMPLEMENTAÇÃO ... 50
4.1. A ferramenta AROS ... 50
4.2. Funções para detectar oportunidades para aplicar padrões de projeto ... 53
4.3. Função para detectar oportunidades para aplicar o padrão Strategy ... 54
4.4. Função para detectar oportunidades para aplicar o padrão Factory Method ... 54
4.5. Função para detectar oportunidades para aplicar o padrão Template Method ... 55
4.6. Função para detectar oportunidades para aplicar o padrão Creation Method ... 57
4.7. Função para detectar oportunidades para aplicar o padrão Chain Constructors ... 58
5. ESTUDO DE CASO ... 60
5.1. Resultados das avaliações para o padrão Strategy ... 61
5.2. Resultados das avaliações para o padrão Factory Method ... 67

5.3. Resultados das avaliações para o padrão Template Method 70
5.4. Resultados das avaliações para o padrão Creation Method... 75
5.5. Resultados das avaliações para o padrão Chain Constructors 78
6. CONCLUSÃO 81
6.1. Trabalhos futuros 82
REFERÊNCIAS 83
ANEXOS 85

1. INTRODUÇÃO

Uma premissa básica no desenvolvimento de sistemas de software é a necessidade de evolução. À medida que os sistemas são evoluídos, modificados e adaptados de acordo com novas exigências, seu código e outros artefatos envolvidos podem se tornar mais complexos e se afastarem de seu objetivo original, podendo diminuir sua qualidade. Em muitos projetos, boa parte do custo é dedicada à manutenção. Embora as metodologias de desenvolvimento de software e as ferramentas ajudem a aumentar a produtividade dos desenvolvedores em suas tarefas, esse ganho faz com que as empresas consequentemente aumentem o número de requisitos a serem implementados dentro do mesmo espaço de tempo, tornando os sistemas de software ainda mais complexos (MENS; TOURWÉ, 2004).

Para lidar com esse aumento de complexidade, que gera custos, existe a necessidade de se usar técnicas que facilitem a manutenção e a legibilidade, melhorando a qualidade interna dos sistemas de software. Uma das técnicas usadas para auxiliar nesse gerenciamento de complexidade é conhecida como **refatoração** (OPDYKE, 1992; FOWLER, 1999; MENS; TOURWÉ, 2004).

Refatoração é o processo de modificar um sistema de software sem alterar seu comportamento externamente observável, melhorando sua estrutura interna. É uma forma disciplinada de melhoria que ajuda a minimizar a introdução de erros ao reestruturar os artefatos de um sistema. Com a evolução de um sistema de software, mudanças ocorrerão, e a sua estrutura e integridade original podem diminuir. É uma técnica de desenvolvimento ágil (FOWLER, 1999).

Essa reestruturação também é definida como a transformação de uma forma de representação para outra, no mesmo nível relativo de abstração, enquanto preserva o comportamento externo do sistema (funcionalidade e semântica). Essas transformações podem melhorar a estrutura do projeto. Embora a reestruturação crie novas versões do sistema, normalmente não envolvem modificações referentes a implementação de novos requisitos. No entanto, podem conduzir a observar melhor o objetivo do sistema, já que sugerem mudanças que poderiam melhorar sua estrutura interna (MENS; TOURWÉ, 2004).

Nesse sentido, é necessário identificar locais nos artefatos de software que precisam ter sua estrutura transformada, seja para melhorar a legibilidade, melhorar o projeto, tornar mais simples as atividades de evolução e de manutenção. Tais estruturas passíveis de refatoração são também conhecidas como *bad smells* (FOWLER, 1999), ou *limitações*. Uma limitação sugere que existem deficiências nos artefatos que podem ser sanadas ou

melhoradas, porém é necessário que a aplicação continue apresentando o mesmo comportamento externo após a transformação interna.

Outro ponto a ser considerado no estudo da programação orientada a objetos é o uso de padrões de projeto (GAMMA et al., 1999), amplamente utilizados no projeto de arquiteturas de software, pois representam soluções gerais, reutilizáveis, testadas e documentadas, que auxiliam na resolução de problemas recorrentes.

Os sistemas de software evoluem e são constantemente modificados ao longo do tempo, geralmente por causa de mudanças de requisitos, correções de erros, melhorias de desempenho e migração para novas plataformas. Um sistema de software deve ser projetado para ser flexível, escalável e fácil de manter. Para lidar com esses fatores, padrões de projeto podem ser usados para melhorar a qualidade de um sistema de software para torná-lo mais fácil de modificar ao longo do tempo.

Um problema relacionado com o uso de padrões de projeto é que, muitas vezes, o projetista do sistema deve ter um conhecimento profundo para determinar qual padrão de projeto pode ser aplicado para resolver um problema específico. Com base nisso, há uma falta de ferramentas que permitam a aplicação de padrões usando uma abordagem semi-automática, onde o programador toma a decisão final sobre a aplicação ou não de um padrão.

A refatoração rumo a padrões de projeto permite melhorar os atributos de qualidade de um sistema de software por meio da aplicação de transformações que preservam o seu comportamento original. Uma alternativa para melhorar a qualidade de um sistema de software é o uso de refatoração para aplicar padrões de projeto com o objetivo de obter benefícios como: maximizar a reutilização de código, reduzir o acoplamento entre classes, incentivar a adoção de boas práticas de desenvolvimento orientado a objeto, facilitar a manutenção e a evolução do sistema de software.

Com base nesses fatores, buscar por oportunidades de refatoração no código fonte visando a aplicação de padrões de projeto é uma prática relevante, porque há um ganho na produtividade do desenvolvedor, considerando que esse processo possa ser apoiado por ferramentas semi-automáticas.

O objetivo deste trabalho é propor uma abordagem para a detecção de oportunidades de refatoração para a aplicação de padrões de projeto. Para isso, foram definidos e implementados cinco algoritmos de busca, para os seguintes padrões: *Strategy*, *Factory Method*, *Template Method* (GAMMA et al., 1999), *Creation Method* e *Chain Constructors* (KERIEVSKY, 2008) e suas respectivas refatorações: *Substituir Lógica Condicional por*

Strategy, *Introduzir Criação Polimórfica com Factory Method*, *Formar Template Method*, *Substituir Construtores por Métodos de Criação* e *Encadear Construtores* (KERIEVSKY, 2008).

Para buscar por oportunidades, a abordagem proposta utiliza um conjunto de funções heurísticas que atuam em árvores sintáticas abstratas (ASTs), que permitem analisar o código fonte a fim de localizar problemas que podem ser eliminados por meio da aplicação de padrões. De forma a avaliar a abordagem proposta, foi desenvolvida uma ferramenta chamada AROS (*Automatic Refactoring Opportunity Search*) que busca por oportunidades de refatoração para a aplicação de padrões de projeto em código C#.

Dessa forma, as principais contribuições deste trabalho são:

- um conjunto de funções heurísticas capaz de detectar se trechos de código que contenham determinadas limitações podem ser refatorados pela aplicação de padrões de projeto, de forma a melhorar um determinado conjunto de métricas;
- uma ferramenta extensível de busca de oportunidades de refatoração em código C# e
- a definição de um conjunto de atividades para definir e refinar indícios de códigos fontes que podem ser refatorados a partir da aplicação de padrões de projeto.

Com isso, podemos dizer que o "estado da arte" deste trabalho e a principal motivação está no fato de que *não existem atualmente pesquisas que foquem na busca por oportunidades de refatoração que tenham como resultado final da transformação um código em conformidade com a estrutura de um padrão de projeto, sendo esta a principal contribuição deste trabalho.* Cada busca por oportunidade define uma mecânica, e nossa implementação para a linguagem C#.

O restante deste trabalho está organizado da seguinte forma.

- o Capítulo 2, *Revisão de literatura*, apresenta a base conceitual que envolve o escopo desta dissertação. São descritas as definições, intenções, características e estruturas de cada um dos padrões de projeto utilizados: *Strategy*, *Factory Method*, *Template Method*, *Creation Method* e *Chain Constructors.* Apresenta as refatorações para estes padrões, incluindo *Substituir Lógica Condicional por Strategy*, *Introduzir Criação Polimórfica com Factory Method*, *Formar Template Method, Substituir Construtores por Métodos de Criação* e *Encadear Construtores*.

- o Capítulo 3, *Busca por oportunidades de refatoração para aplicação de padrões de projeto*, apresenta a abordagem proposta e como ocorre a busca em código fonte para encontrar oportunidades para aplicar padrões de projeto.
- o Capítulo 4, *Implementação*, apresenta a ferramenta desenvolvida e a nossa implementação das funções heurísticas propostas.
- o Capítulo 5, *Estudo de Caso,* apresenta os resultados obtidos por meio da avaliação das funções implementadas em alguns projetos de código aberto.
- o Capítulo 6, *Conclusão¸* resume as contribuições desse trabalho e discute sugestões para trabalhos futuros.

2. REVISÃO DE LITERATURA

2.1. Padrões de Projeto

Usar padrões de projeto torna mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objeto subjacente. Padrões ajudam a solucionar problemas de projeto, por exemplo, quando estamos procurando objetos apropriados em um sistema orientado a objetos. Um objeto armazena tantos os dados quanto os procedimentos que operam sobre eles. Os procedimentos são chamados de métodos ou operações. Um objeto executa uma operação quando ele recebe uma solicitação (ou mensagem) de um cliente (GAMMA et al., 1999).

Muitos fatores devem ser considerados quando se precisa decompor um sistema em objetos: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização, e assim por diante. Os padrões de projeto ajudam a identificar estas estruturas e seus relacionamentos. Por exemplo, o padrão Façade descreve como representar subsistemas inteiros como objetos. Outros padrões descrevem maneiras específicas de como decompor um objeto em objetos menores. O Abstract Factory e o Builder fornecem objetos cujas únicas responsabilidades são criar outros objetos (GAMMA et al., 1999).

Padrões de Projeto também ajudam a especificar interfaces de objetos. As interfaces são fundamentais em sistemas orientados a objetos, pois é por meio delas que um objeto pode solicitar operações a serem realizadas sobre outro objeto. A interface independe da implementação de um objeto, com isso objetos estão livres para implementar as solicitações de diferentes maneiras, ao mesmo tempo em que podem ser totalmente substituídos por outros objetos que implementam a mesma interface, realizando as mesmas operações, de forma diferente (técnica conhecida como polimorfismo). Isso reduz o acoplamento entre classes, promovendo a reutilização (GAMMA et al., 1999).

Padrões de projeto também ajudam a especificar a implementação de objetos. Uma implementação é definida por uma ou mais classes. Elas representam os dados que os objetos devem conter, bem como suas operações. Quando classes são criadas a partir da herança de outras classes, diz-se que as classes descendentes são as subclasses, e a classe base, superclasse, ou superclasses, no caso de suporte à herança múltipla. Uma classe abstrata

normalmente serve de base para outras e pode especificar uma interface comum para as suas subclasses, podendo ainda conter comportamento comum, porém não pode ser instanciada. As classes que herdam dela e são instanciadas são chamadas de classes concretas. Nesse ponto é importante diferir entre herança de classe versus herança de interface. Ao herdar de uma interface, uma classe apenas assume que irá implementar as operações por elas definida. Diferentemente, uma classe abstrata pode também conter um comportamento padrão para ser herdado por suas subclasses. Isso remete a uma das principais boas práticas da programação orientada a objetos, a programação para interfaces, e não para uma implementação. Padrões de projeto ajudam a seguir esta prática, ao promoverem o baixo acoplamento entre classes usando soluções baseadas em interfaces (GAMMA et al., 1999).

Padrões de projeto também ajudam a identificar quando utilizar herança ou composição para reaproveitamento de código. A composição é uma alternativa à herança, conhecida como herança de caixa preta, já que os detalhes do objeto composto não ficam expostos ao mundo externo, se este for privado. Já a herança é mais rígida, a classe que herda estará dependendo da hierarquia de tipos definida, tendo pouca flexibilidade, se esta herança for pública. Nesse caso, é preferível usar composição à herança. A chave para a maximização da reutilização está na antecipação de novos requisitos e mudanças nos requisitos existentes e em projetar sistemas de modo que eles possam evoluir de acordo. Padrões ajudam a projetar sistemas de software mais bem preparados para mudanças (GAMMA et al., 1999).

Nas seções a seguir são apresentados os padrões de projeto utilizados na busca por oportunidades de refatoração por meio das funções descritas neste trabalho. Os padrões considerados são: *Strategy, Template Method* (padrões comportamentais), *Creation Method, Chain Constructors* e *Factory Method* (padrões criacionais).

2.1.1. Strategy

O padrão de projeto Strategy (GAMMA et al., 1999) define uma família de algoritmos, encapsula cada uma deles e torna-os intercambiáveis. Strategy permite que o algoritmo varie independentemente das classes clientes que o utilizam. Este padrão pode ser aplicado quando muitas classes têm objetivos semelhantes e diferem apenas em seu comportamento (implementação). O Strategy pode ser utilizado quando é necessário encapsular uma solução para um problema que classes clientes não devem estar cientes (GAMMA et al., 1999).

Este padrão traz alguns benefícios, como a eliminação de instruções condicionais para selecionar o comportamento desejado (algoritmo) para resolver um problema específico. Usando herança e abstração, o Strategy encapsula esses comportamentos em subclasses e utiliza polimorfismo para substituir a condicional estática, tornando a arquitetura extensível para a inclusão de novas estratégias para resolver o mesmo problema (GAMMA et al., 1999).

Participantes:

- **Strategy** – define uma interface comum para todos os algoritmos envolvidos. *Context* usa esta interface para chamar o algoritmo definido por uma *ConcreteStrategy*;
- **ConcreteStrategy** – implementa o algoritmo usando a interface *Strategy*;
- **Context** – mantém uma referência para um objeto *Strategy*; pode definir uma interface que permite a *Strategy* acessar seus dados;

A Figura 1 mostra a estrutura do padrão Strategy. Ela define uma superclasse abstrata chamada *Strategy* com uma interface de algoritmo, geralmente um método abstrato ou virtual. Classes descendentes podem herdar esta classe base para proporcionar uma aplicação específica para a interface do algoritmo. Tais classes são conhecidas como estratégias concretas. Classes clientes utilizam então uma classe especial chamada *Context* que por meio de delegação pode invocar o método polimórfico usando a interface da estratégia (GAMMA et al., 1999).

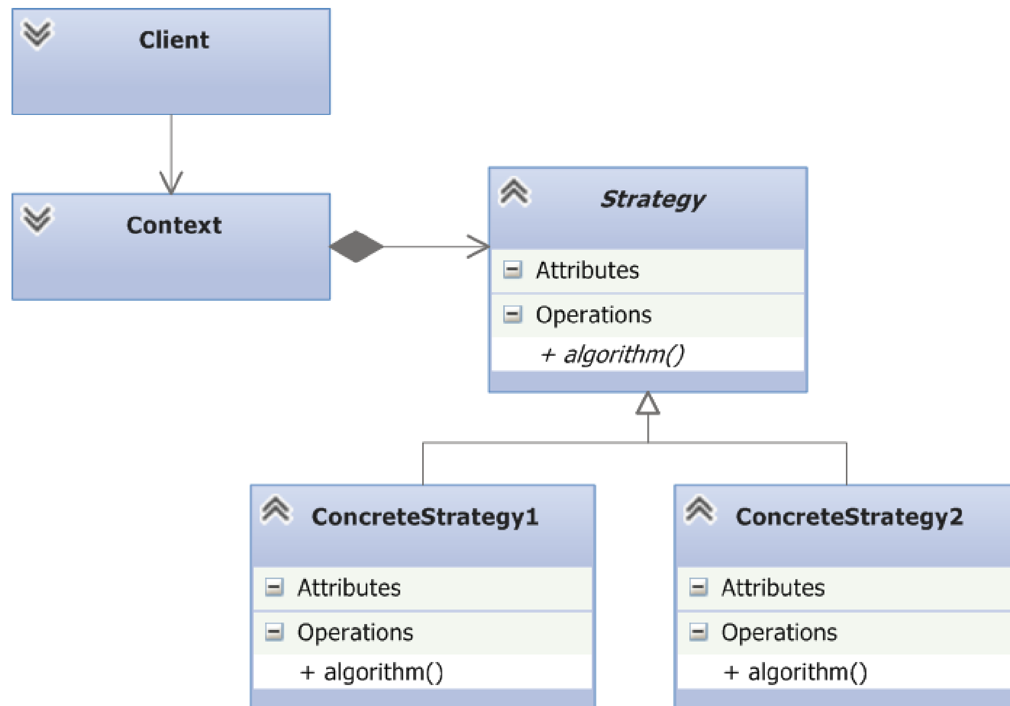


Figura 1 – Estrutura do Padrão de Projeto Strategy (GAMMA et al., 1999)

2.1.2. Factory Method

Factory Method é um padrão de projeto criacional que define uma superclasse com uma interface destinada a criar um objeto, porém deixa que suas subclasses decidam qual o tipo deste objeto. Este padrão define um método abstrato em uma determinada subclasse, também chamado de "construtor virtual", que retorna uma classe abstrata. Dessa forma, as subclasses podem substituir o método virtual para criar e retornar subclasses especializadas, que são chamadas de produtos ou classes concretas. A aplicação do padrão é indicada quando uma classe não é capaz de saber com antecedência qual objeto deve realmente ser criado, ou quando uma classe delega para suas subclasses a responsabilidade de decidir quais objetos realmente devem ser criados, usando polimorfismo (GAMMA et al., 1999).

A aplicação desse modelo traz alguns benefícios, sendo o mais importante a flexibilidade. Devido ao fato de a criação do objeto estar encapsulada por um método de fábrica, as classes clientes não precisam criar diretamente esses objetos. Além disso, como uma classe cliente utiliza o método de fábrica para obter uma instância da classe desejada, o padrão pode ser utilizado como uma forma transparente e flexível para proporcionar um objeto diferente. Isso acontece porque se pode retornar um objeto compatível com a classe retornada pela superclasse (GAMMA et al., 1999).

Outro benefício está relacionado com as sentenças condicionais, porque muitas declarações desse tipo são usadas para decidir que tipo de objeto é necessário ser criado. O padrão pode eliminar essas declarações condicionais para selecionar o comportamento desejado (algoritmo) para criar um objeto específico. Além disso, o padrão pode ser utilizado para reduzir o acoplamento de classe e de dependência, pois classes clientes não criam produtos diretamente por meio do operador *new*. Se uma classe cliente precisa de uma instância de um determinado produto, é necessário solicitar à fábrica. A criação dos objetos se mantém encapsulada (GAMMA et al., 1999).

Participantes:

- **Product** - define a interface de objetos que o método de fábrica cria;
- **ConcreteProduct** – implementa a interface *Product*;
- **Creator** – Declara o método de fábrica, o qual retorna um objeto do tipo *Product*;
- **ConcreteCreator** – redefine o método fábrica para retornar uma instância de um *ConcreteProduct*;

A Figura 2 mostra a estrutura do padrão Factory Method. Na parte superior podemos ver uma família de produtos: os dois produtos (classes concretas) chamados *ProductOne* e *ProductTwo* são subclasses de uma classe abstrata chamada *Product*. Na parte inferior podemos ver outra família de classes: *Application* é uma classe base que define um método abstrato chamado *CreateProduct* e é usado para criar objetos do tipo do *Product* (a classe base abstrata para *ProductOne* e *ProductTwo*). As subclasses *ApplicationOne* e *ApplicationTwo* podem então substituir o método virtual para retornar instâncias concretas de *ProductOne* e *ProductTwo*, que são compatíveis com o tipo retornado pelo método polimórfico na classe base. Neste exemplo, *CreateProduct* é o método de fábrica propriamente dito. Classes clientes podem usar a fábrica para solicitar por produtos, como instâncias das classes *ProductOne* e *ProductTwo*. Toda a complexidade da criação é encapsulada pelo método de fábrica e por sua família de subclasses.

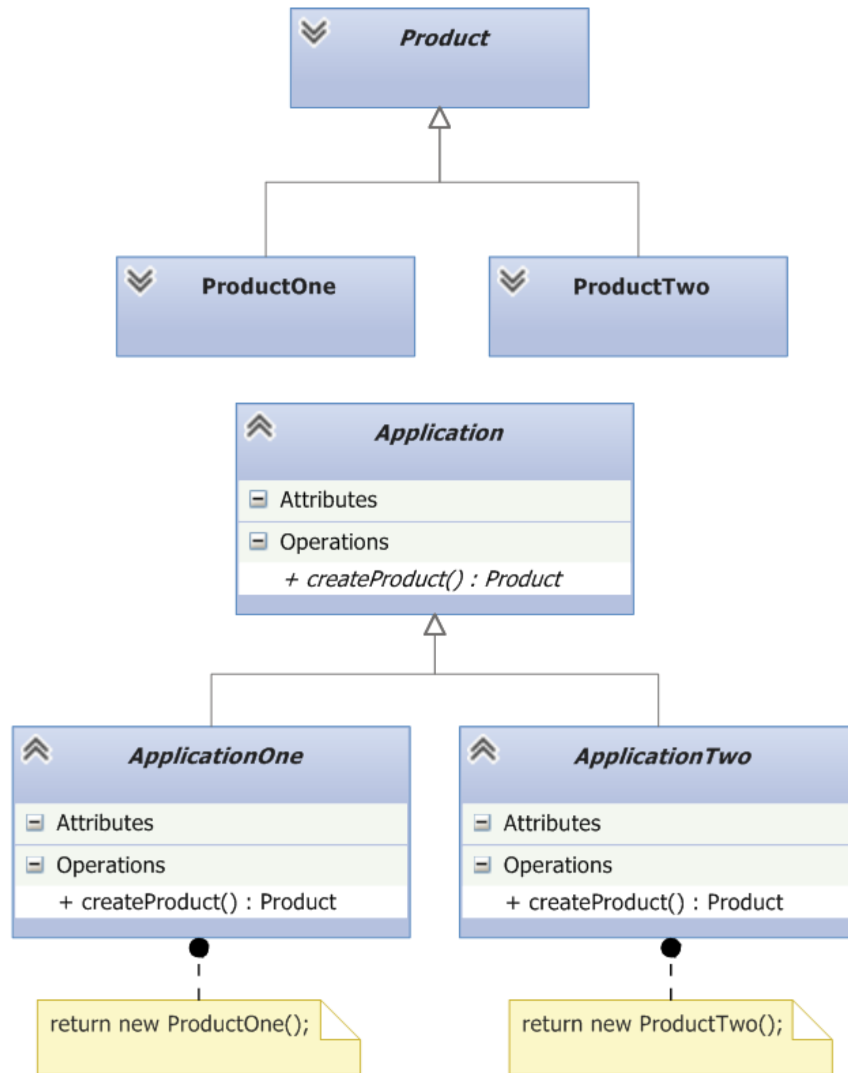


Figura 2 – Estrutura do Padrão de Projeto Factory Method (GAMMA et al., 1999)

2.1.3. Template Method

Template Method é um padrão de projeto comportamental que define uma interface de um algoritmo em um método em uma determinada superclasse, transferindo alguns passos para subclasses. O padrão permite que subclasses possam redefinir os passos do algoritmo sem alterar a estrutura previamente definida (GAMMA et al., 1999).

O padrão Template Method pode ser usado para reduzir a duplicação de código. Quando subclasses implementam um conjunto de métodos em uma certa ordem, usando a mesma sequência de passos, esses passos podem formar um único método na superclasse, que invoca métodos polimórficos que são então redefinidos nas subclasses. Nesse caso, o padrão elimina a duplicação causada pela invocação da mesma sequência de passos em subclasses.

Em outras palavras, a classe base declara *placeholders* para algoritmos, e as classes derivadas implementam esses *placeholders* (GAMMA et al., 1999).

Participantes:

- **AbstractClass** – define as operações primitivas abstratas que as subclasses concretas definem para implementar passos de um algoritmo; implementa um método-template que define o esqueleto de um algoritmo;
- **ConcreteClass** – implementa as operações primitivas para executar os passos específicos invariantes do algoritmo;

A Figura 3 mostra a estrutura do padrão Template Method. Existe uma superclasse chamada *BaseClass* com um conjunto de instruções, tais como chamadas de método (*Operation1*, *Operation2*, *Operation3*) e um método template que chama esses métodos em uma sequência imutável. Em seguida, usando polimorfismo, as subclasses (*ConcreteClass1* e *ConcreteClass2*) podem sobrescrever esses métodos já invocados pelo método template da classe base. Neste caso, as subclasses não precisam duplicar a sequência de chamadas de método, basta implementar cada um (GAMMA et al., 1999).

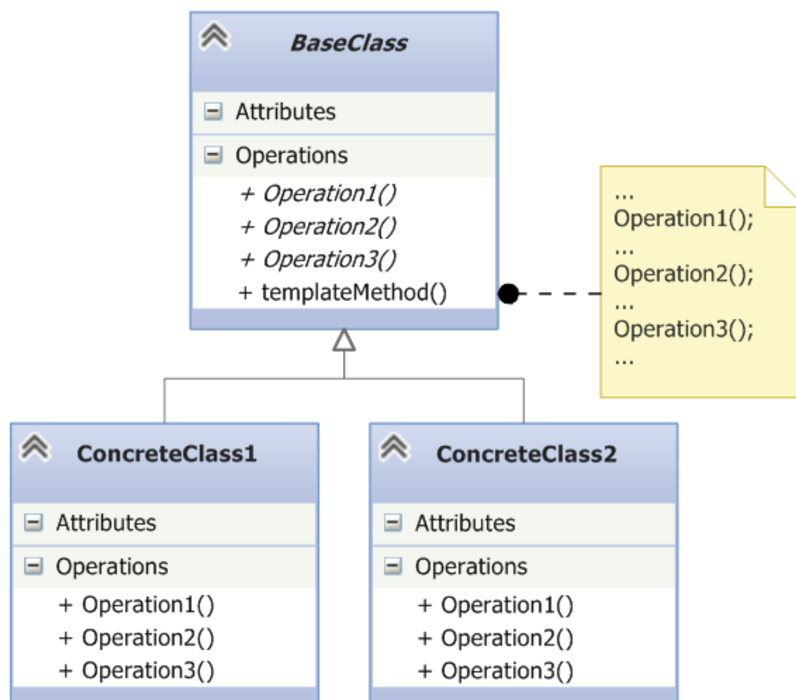


Figura 3 – Estrutura do Padrão de Projeto Template Method (GAMMA et al., 1999)

2.1.4. Creation Method

A intenção do padrão Creation Method é definir um método específico para instanciar uma determinada classe, sendo que este método é usado no lugar de um construtor padrão. O padrão pode ser aplicado quando uma classe possui inúmeros construtores, recebendo diferentes parâmetros, variando em quantidade e tipo (KERIEVSKY, 2008).

Na maioria das linguagens orientada a objetos, como C# e Java, o construtor deve ter o mesmo nome da classe. Um programador pode então criar *n* versões sobrecarregadas do construtor para permitir a criação do objeto de acordo com determinados atributos, como por exemplo, instanciar um objeto usando alguns valores iniciais para seus atributos, que são justamente recebidos nessa construção. O problema ocorre quando uma classe possui muitos construtores, pois fica difícil para o programador identificar aquele que se adequa melhor com o que precisa ser passado como parâmetro, pois o construtor não expressa de forma clara sua intenção (KERIEVSKY, 2008).

Entre as principais vantagens do uso de Creation Method, podemos citar que ele não possui as limitações de linguagem impostas para construtores, como por exemplo, usar o mesmo nome da classe e não poder ter dois construtores com o mesmo número e tipo de argumentos (o que causaria ambiguidade). Além disso, um método de criação usado no lugar de um construtor pode tornar mais claro o que realmente está sendo inicializado naquele objeto, ou seja, torna mais clara a sua intenção. E finalmente, definir métodos de criação torna mais fácil encontrar construtores que não estejam mais sendo utilizados. Uma desvantagem é que essa abordagem torna a programação menos padronizada, pois o desenvolvedor deixa de utilizar a sintaxe padrão de construção da linguagem (KERIEVSKY, 2008).

Na Figura 4 mostra é exibida a estrutura do padrão Creation Method. Existe um construtor padrão para uma classe chamada *Song* que recebe parâmetros para inicializar todos os seus atributos, tais como *Album*, *Author* e *Title*. Este construtor é privado à classe. Então existem alguns construtores públicos específicos que expressam melhor a intenção do que estão inicializando na classe, como por exemplo *newSongTitle*, que cria uma instância da classe para representar uma música com um determinado título (KERIEVSKY, 2008).

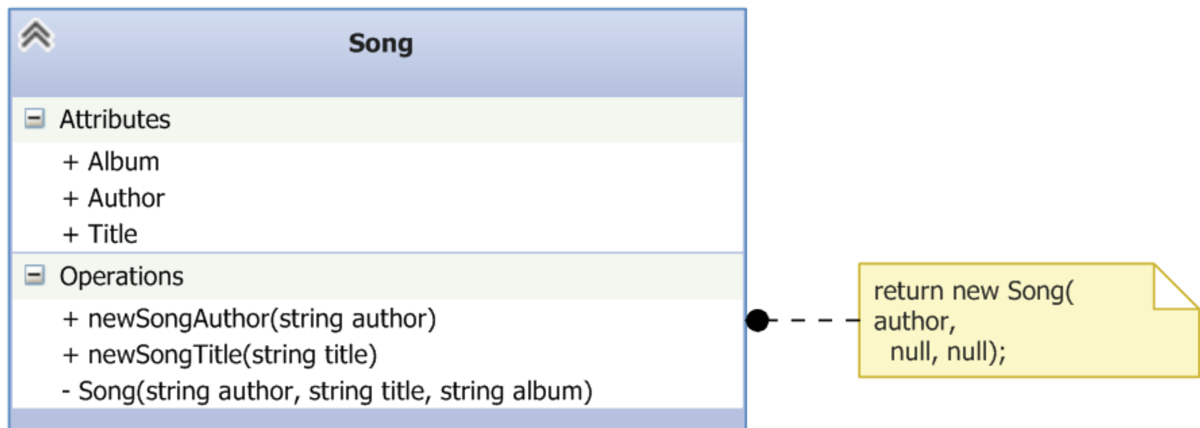


Figura 4 – Estrutura do Padrão de Projeto Creation Method (KERIEVSKY, 2008)

2.1.5. Chain Constructors

A intenção do padrão Chain Constructors (KERIEVSKY, 2008) é reduzir código duplicado relacionado à construção de objetos. Normalmente, nas linguagens orientadas a objetos, uma classe possui *n* sobrecargas de construtores, destinadas a inicializar alguns membros internos ou invocar métodos para inicialização de dados necessários à classe. O problema ocorre quando uma classe possui muitos construtores e esses construtores repetem código de inicialização.

A Figura 5 mostra a estrutura do padrão Chain Constructors. Ela define um construtor padrão para uma classe chamada *Employee* que recebe parâmetros para inicializar todos os seus atributos, tais como *EmployeeName*, *Department* e *Salary*. Existem então outros construtores que são então "encadeados", de forma a inicializar algo específico, porém utilizando o construtor genérico para inicializar os dados comuns, sem replicar código (KERIEVSKY, 2008).

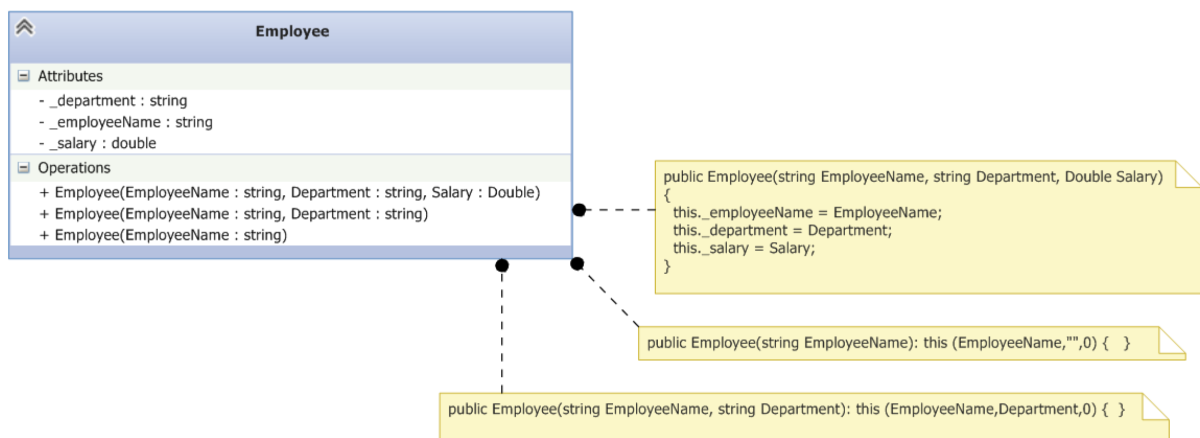


Figura 5 – Estrutura do Padrão de Projeto Chain Constructors (KERIEVSKY, 2008)

2.2. Refatoração

Pode-se definir o termo refatorar como reestruturar um sistema de software aplicando-se uma série de transformações sem modificar seu comportamento observável. Refatoração é uma mudança feita na estrutura interna de um sistema de software para torná-lo mais fácil de entender e modificar. Cada pequeno passo melhora o projeto do código, mantendo seu comportamento externo observável (FOWLER, 1999).

Existem várias motivações para o uso de refatorações. Primeiro, elas tornam mais fácil a adição de novos trechos de código. Quando se inclui uma nova funcionalidade em um sistema de software, existem basicamente duas opções: programar rapidamente sem se preocupar o quanto ela se encaixa bem no projeto existente, ou pode-se modificar o projeto existente a fim de melhor acomodar essa funcionalidade. No primeiro caso, ocorre um débito de projeto, o qual pode ser pago mais tarde com refatoração (KERIEVSKY, 2008).

Segundo, as refatorações melhoraram um projeto de código existente, no momento que tornam o código mais simples, claro e mais fácil de trabalhar, manter e evoluir. A refatoração contínua envolve procurar constantemente por problemas e removê-los imediatamente (KERIEVSKY, 2008).

Por fim, refatorações ajudam a se obter um melhor entendimento do código. Se um código está difícil de ser compreendido, programadores normalmente comentam este código para tornar mais clara sua intenção. Se o código não está claro, ele pode ser aprimorado por meio de refatoração (KERIEVSKY, 2008).

A refatoração tende a quebrar grandes problemas em problemas menores (por exemplo, objetos). Porém, mais objetos precisam ser gerenciados. E objetos nesse caso normalmente costumam delegar tarefas a outros objetos, indiretamente. Por exemplo, ao colocar a lógica de uma tarefa em um método chamado por vários objetos, utiliza-se uma boa prática de compartilhamento. Além disso, a intenção do método e sua implementação são feitas separadamente. Com isso, isola-se algo que pode mudar, evitando replicação ao mesmo tempo em que deixa o código limpo.

A refatoração serve como uma alternativa ao chamado *big design up-front* (Grande Projeto Antecipado, ou G.P.A.) (KERIEVSKY, 2008). Nesse cenário, não se cria inicialmente um grande projeto, que tente prever todas as possibilidades de mudanças que possam ocorrer no sistema de software. Utiliza-se a abordagem *up-front*, porém não se tenta achar a solução para tudo. Ao desenvolver de forma evolutiva e incremental, o conceito de solução pode mudar, ou melhorar.

Nesse contexto, uma boa prática é utilizar a refatoração para modificar um código existente, tendo como resultado final dessa transformação um código em conformidade com a estrutura de um padrão de projeto. As seções a seguir apresentam o conjunto de cinco refatorações destinadas a aplicar padrões de projeto para resolver problemas no código. Estas refatorações são a *Substituir Lógica Condicional por Strategy, Introduzir Criação Polimórfica com Factory Method, Formar Template Method, Substituir Construtores por Métodos de Criação* e *Encadear Construtores* (KERIEVSKY, 2008).

2.2.1. Substituir Lógica Condicional por Strategy

A refatoração *Substituir Lógica Condicional por Strategy* (KERIEVSKY, 2008) tem a finalidade de aplicar o padrão de projeto Strategy para remover um teste condicional que decide a escolha de um algoritmo em particular. Sua aplicação consiste em encapsular cada um dos algoritmos dentro de uma família de classes, chamadas estratégias, e cada classe representa uma variante do algoritmo. A classe cliente, que anteriormente tinha uma ligação direta com o algoritmo, deve usar uma classe *Context* que recebe uma *ConcreteStrategy*, para que se possa eliminar a lógica condicional utilizando polimorfismo.

Às vezes, as expressões condicionais tornam o código complexo, especialmente se vários testes condicionais forem aninhados. O uso de refatoração neste cenário traz vantagens, tais como a facilidade de alterar o código-fonte em tempo de execução, uma vez que a classe cliente está ligada a uma abstração. Além disso, ele faz com que o código fonte se torne mais claro porque não são utilizadas condicionais estáticas - a escolha pela execução de um determinado algoritmo é feita de forma dinâmica usando-se polimorfismo (KERIEVSKY, 2008).

A Figura 6 mostra um exemplo onde a refatoração *Substituir Lógica Condicional por Strategy* pode ser aplicada. Nela podemos ver uma classe chamada *Loan*, que tem uma lógica específica para calcular os montantes de empréstimos de acordo com várias condições testadas em um bloco switch (KERIEVSKY, 2008).

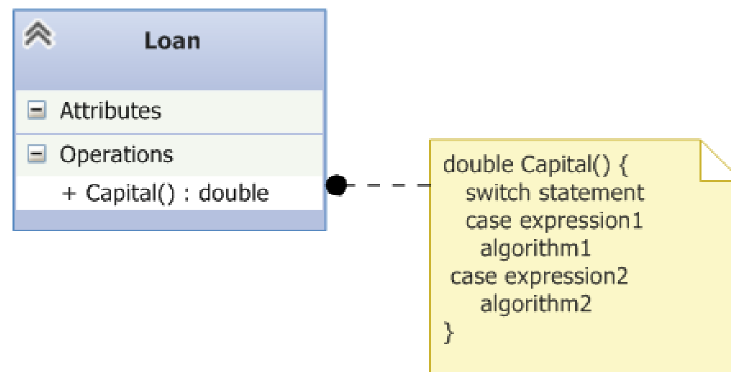


Figura 6 – Substituir Lógica Condicional por Strategy, exemplo antes da refatoração

A Figura 7 mostra o exemplo refatorado. Cada código relacionado com um expressão *case* do bloco condicional pode ser transformado em um método dentro de uma classe concreta (estratégia).

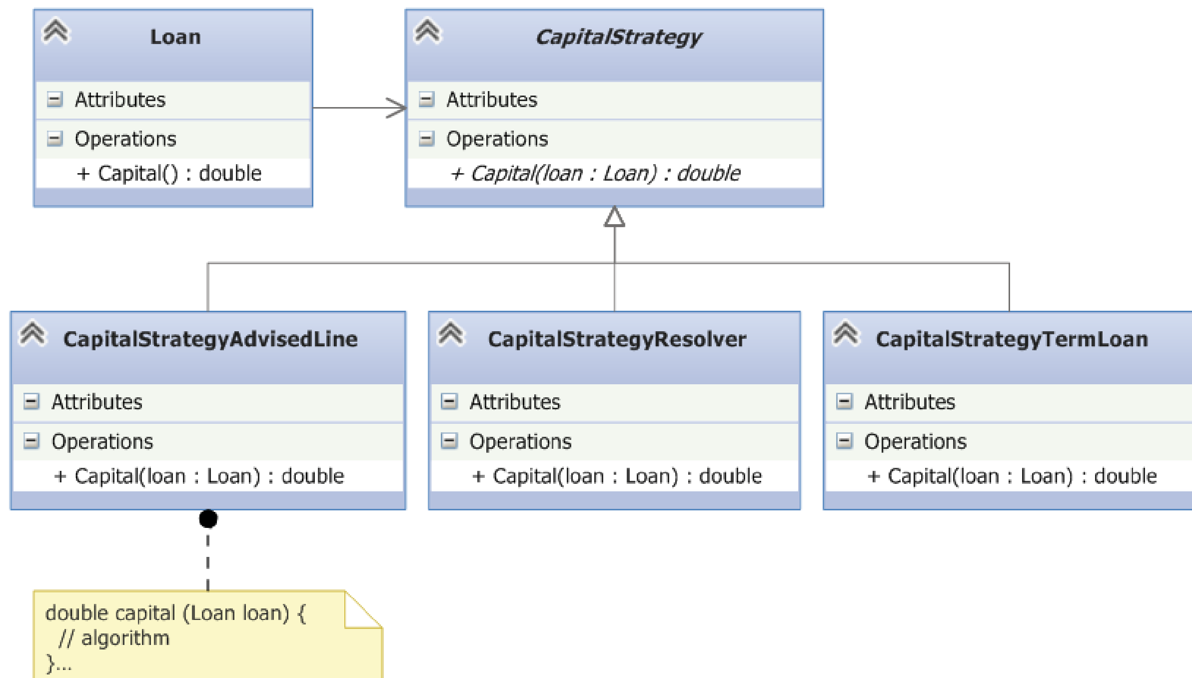


Figura 7 –Substituir Lógica Condicional por Strategy, exemplo depois da refatoração

Dessa forma, a classe cliente pode se comunicar com qualquer estratégia de forma transparente, por meio de uma abstração, como podemos ver no centro da figura.

2.2.2. Introduzir Criação Polimórfica com Factory Method

A aplicação da refatoração *Introduzir Criação Polimórfica com Factory Method* (KERIEVSKY, 2008) tem muitas motivações. Um dos mais importantes é reduzir código

duplicado, especialmente quando várias subclasses têm métodos para executar passos semelhantes em sua implementação. Normalmente, exceto pela decisão de qual objeto deve ser criado, esses passos semelhantes podem ser movidos para um método na classe base, contendo todas essas etapas. A criação do objeto específico, normalmente usando o operador *new*, pode ser transformado em um método virtual (o método de fábrica) e substituído por subclasses, que podem retornar objetos específicos usando-se polimorfismo.

Outra forma de refatoração que tem a intenção de aplicar o padrão Factory Method é a refatoração *Substituir Construtor por Método de Fábrica* (FOWLER, 1999). A principal motivação para aplicá-lo é substituir "códigos de tipo" (type codes) por subclasses. Códigos que representam tipos podem ser descritos como inteiros, enumerações ou strings, normalmente tipos primitivos. Construtores podem receber esses type codes como parâmetro, por exemplo, um número, e retornar uma subclasse específica.

A Figura 8 mostra a primeira parte de um exemplo de como aplicar a refatoração *Introduzir Criação Polimórfica com Factory Method*. No topo da figura podemos ver uma classe chamada *Document*. Na parte inferior existem duas subclasses (*XMLDocument* e *TXTDocument*), cada uma tem um método de mesmo nome que permite abrir documentos (*OpenDocument*). Usando o operador *new*, cada método cria uma classe específica e concreta para ler documentos do disco. A classe *XMLDocument* cria um *XMLReader* para executar esta operação, enquanto a classe *TXTDocument* cria um *TXTReader* para executar a operação similar. Exceto pela criação destas classes específicas para ler arquivos do disco, todo o restante do código dos métodos *OpenDocument* são semelhantes.

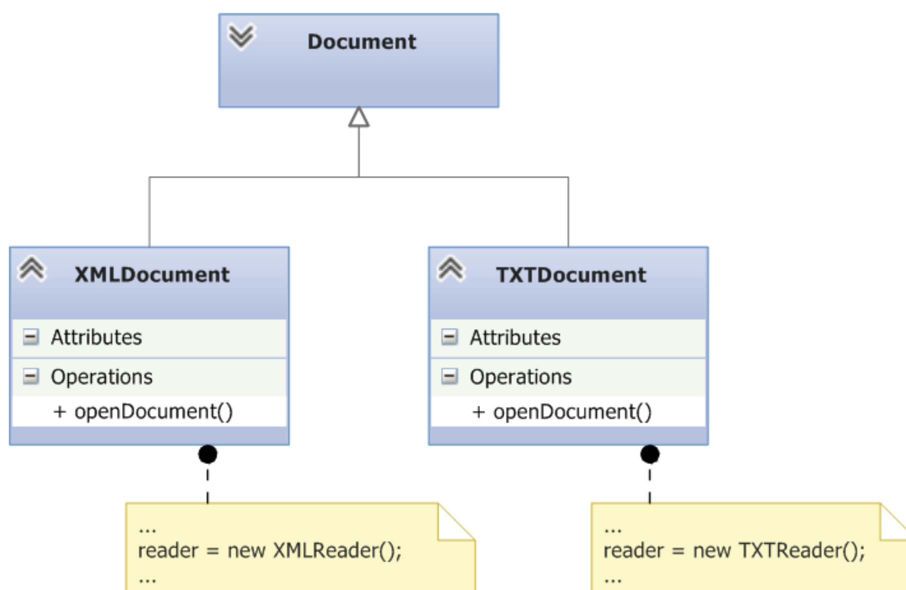


Figura 8 –Introduzir Criação Polimórfica com Factory Method, exemplo antes da refatoração

A Figura 9 mostra o exemplo reformulado pela aplicação da refatoração *Introduzir Criação Polimórfica com Factory Method*.

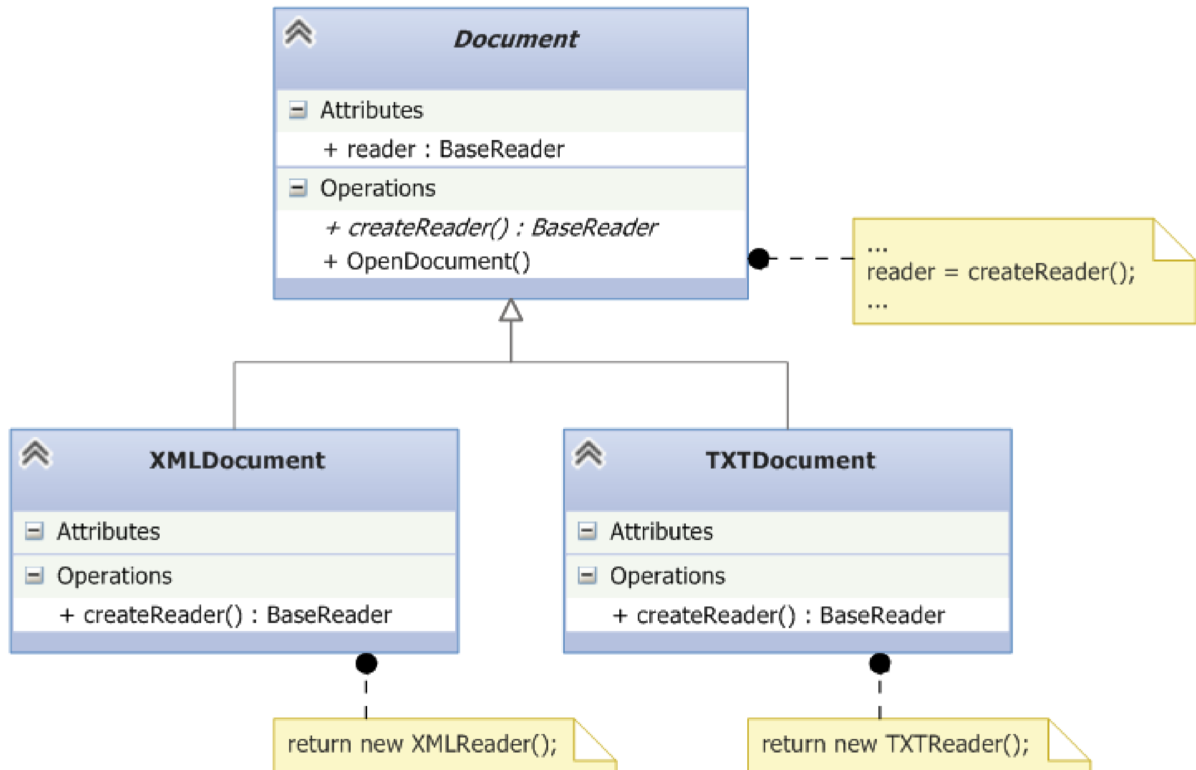


Figura 9 – Introduzir Criação Polimórfica com Factory Method, exemplo depois da refatoração

A classe *Document* torna-se uma classe abstrata com um método polimórfico para criar classes de leitura de arquivos (*XMLReader* ou *TXTReader*), que sempre retorna um tipo mais genérico chamado *BaseReader*, que é uma abstração. *OpenDocument* funciona como um método template, de modo que ele contém todas as etapas semelhantes para abrir documentos, mas precisa chamar um método polimórfico para delegar para subclasses a criação de classes de leitura específicas, reduzindo a duplicação de código. As subclasses *XMLDocument* e *TXTDocument* podem sobrescrever o método de fábrica para retornar classes de leitura específicas (*XMLReader* e *TXTReader*), todos os outros passos semelhantes relacionados são movidos para o método template na classe base (*OpenDocument*).

2.2.3. Formar Template Method

A refatoração *Formar Template Method* (KERIEVSKY, 2008) define as partes invariantes de um algoritmo de uma determinada superclasse, deixando a tarefa de implementar o comportamento variável para suas subclasses (GAMMA et al., 1999). O

comportamento invariável pode estar duplicado em subclasses, por exemplo, uma sequência de chamadas de métodos idêntica.

Aplicando a refatoração *Formar Template Method*, o comportamento invariável de subclasses pode ser refatorado, sendo movido para um método genérico em uma superclasse, eliminando a duplicação de código. Esta refatoração deixa mais clara a intenção do método de template, indicando o que o algoritmo faz por meio da sequência de chamada de métodos. Finalmente, esta refatoração permite que subclasses possam personalizar facilmente alguns passos de acordo com seu propósito, usando polimorfismo.

A Figura 10 mostra parte de um exemplo no qual pode ser aplicado a refatoração *Formar Template Method*. Na parte superior pode-se ver uma classe base chamada *BaseReport*, que define características comuns para imprimir relatórios. Essa classe tem duas subclasses, chamadas *FinancialReport* e *AcademicReport*. A implementação de ambas as subclasses é muito semelhante. Para gerar um relatório, o método *PrintReport* em ambas as subclasses chama uma sequência de métodos na mesma ordem, como imprimir o cabeçalho, detalhes, rodapé, etc.

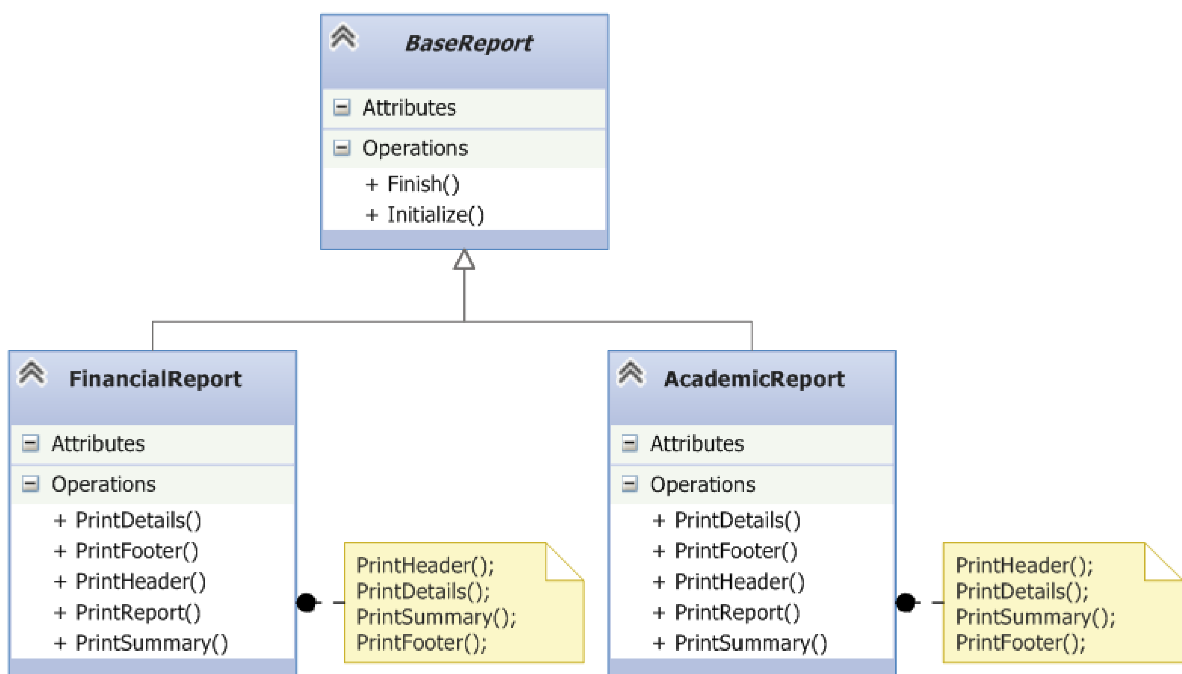


Figura 10 –Formar Template Method, exemplo antes da refatoração

A Figura 11 mostra a aplicação da refatoração *Formar Template Method* com base no exemplo anterior. A parte invariante das subclasses pode ser extraída para um novo método (o template method) na superclasse, chamado *PrintReport*, que define a sequência de etapas para

imprimir um relatório. Todas as etapas comuns são definidas como métodos virtuais na superclasse, então podem ser sobrescritos nas subclasses.

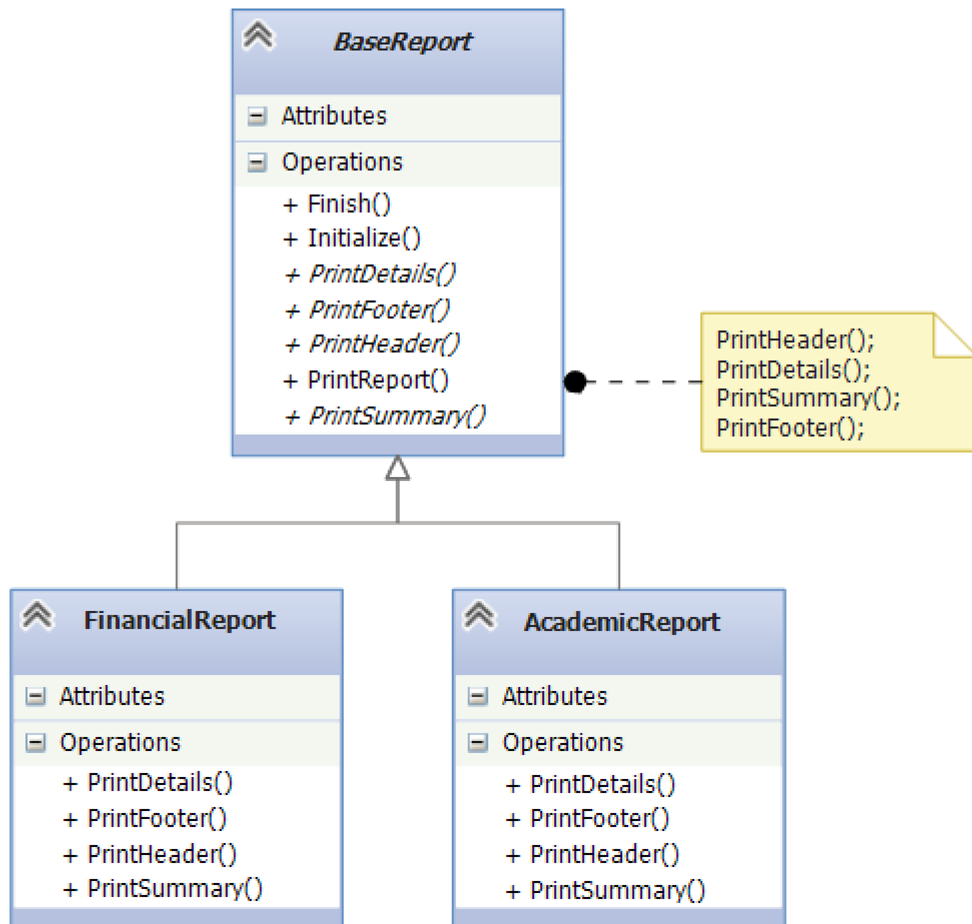


Figura 11 – Formar Template Method, exemplo depois da refatoração

Dessa forma, *BaseReport* pode invocar cada método virtual no corpo do template method. Usando polimorfismo, subclasses podem determinar um comportamento diferente para gerar cada parte do relatório, mas seguindo sempre a mesma ordem pré-definida, que não precisa ser replicada em cada subclasse.

2.2.4. Substituir Construtores por Métodos de Criação

A refatoração *Substituir Construtores por Métodos de Criação* (KERIEVSKY, 2008) tem a intenção de eliminar o uso de uma série de construtores sobrecarregados em uma determinada classe por métodos explícitos que exprimem melhor a intenção do que estão inicializando no objeto. Quando uma classe tem *n* construtores sobrecarregados, muitos podem estar obsoletos. Podem confundir o programador na escolha de qual versão do construtor utilizar. Dessa forma. essa refatoração deixa mais clara a intenção do “construtor” ao transformá-lo num método de criação.

Uma desvantagem da aplicação desta refatoração é que ela deixa o código menos padronizado, visto que programadores não utilizarão mais o operador padrão de construção da linguagem para instanciar objetos. Nesse caso, uma boa prática é incluir a palavra "new" como prefixo no nome do próprio método de criação, por exemplo "newDocument".

Além disso, essa refatoração pode resolver algumas limitações relacionados ao uso excessivo de parâmetros, como por exemplo, a limitação *Lista Longa de Parâmetros* (FOWLER, 1999), que ocorre quando um método recebe vários argumentos que obrigariamente devem ser passados, mesmo se os valores sejam nulos.

Passar valores nulos não é considerada uma boa prática, pois indica que programadores não identificaram qual versão correta do construtor chamar, caso exista uma. Esse tipo de limitação também causa problemas quando há uma mudança na assinatura do construtor, como por exemplo, a eliminação, a adição ou a troca do tipo de dados de um parâmetro. Nesse caso, todas as classes clientes que usam essa versão do construtor serão afetadas.

A Figura 12 mostra parte de um exemplo no qual pode ser aplicado a refatoração *Substituir Construtores por Métodos de Criação*. No exemplo temos uma classe chamada *Document*, com vários construtores sobrecarregados, tendo o mesmo nome, que é uma convenção da maioria das linguagens orientadas a objeto, e variando os parâmetros em quantidade e tipo. Por exemplo, existem versões do construtor para criar um documento inicializando seu autor, já outra versão recebe por parâmetro o autor e o título.

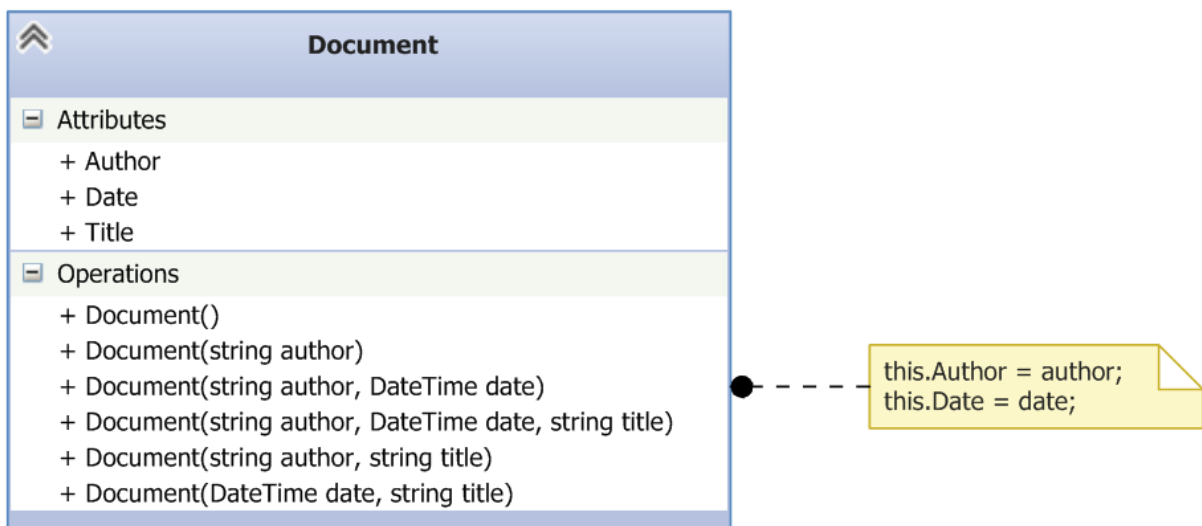


Figura 12 – Substituir Construtores por Métodos de Criação, exemplo antes da refatroração

A Figura 13 mostra a aplicação da refatoração *Substituir Construtores por Métodos de Criação* com base no exemplo anterior. Foram adicionados dois métodos de criação, um que recebe um autor como parâmetro para a criação do documento, o outro recebe um título.

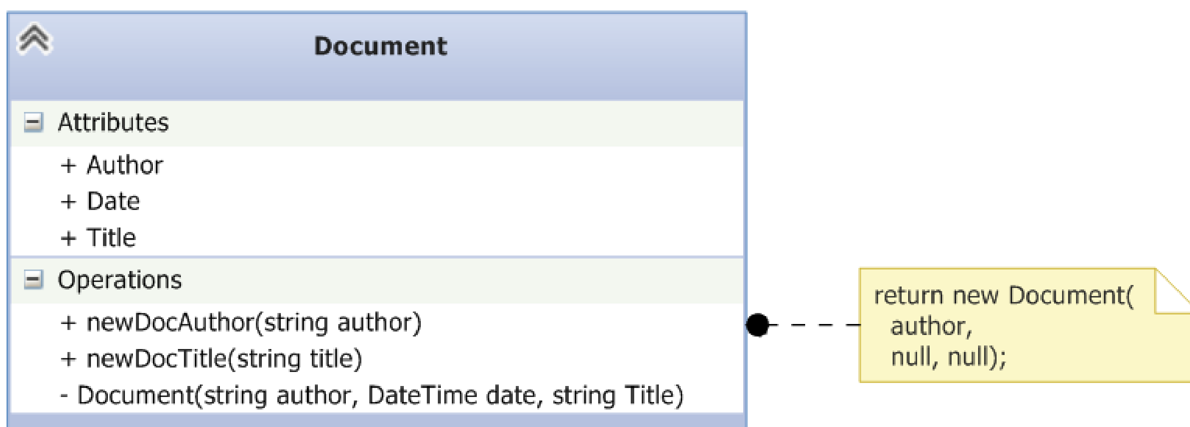


Figura 13 – Substituir Construtores por Métodos de Criação, exemplo depois da refatoração

É mantido um único construtor para a classe, que recebe todos os parâmetros, no entanto este se torna privado e pode ser invocado apenas pelos respectivos métodos de criação, que decidem o que deve ou não ser passado como parâmetro.

2.2.5. Encadear Construtores

A refatoração *Encadear Construtores* (KERIEVSKY, 2008) tem a intenção de aplicar o padrão Chain Construtors para eliminar duplicação de código em construtores. Código duplicado ao longo de construtores tende a originar problemas. Se um programador adiciona uma nova inicialização a um construtor e esquece de replicar este comportamento em outro construtor, a classe poderá apresentar comportamentos inesperados.

O encadeamento de construtores permite que construtores mais específicos chamem um construtor genérico de propósito mais geral, até que o objeto final da construção seja alcançado. O construtor ao final da cadeia será o principal, pois manipula todas as chamadas, devido ao fato de normalmente aceitar o maior número possível de parâmetros.

A Figura 14 mostra parte de um exemplo no qual pode ser aplicado a refatoração *Encadear Construtores*. No exemplo temos uma classe chamada *Employee*, com vários construtores sobrecarregados, tendo o mesmo nome, que é uma convenção da maioria das linguagens orientadas a objeto, e variando os parâmetros em quantidade e tipo. Por exemplo, existem versões do construtor para criar um funcionário inicializando seu nome, já outra versão recebe por parâmetro o nome e o salário. Nesse caso, a atribuição ao nome do funcionário está duplicada.

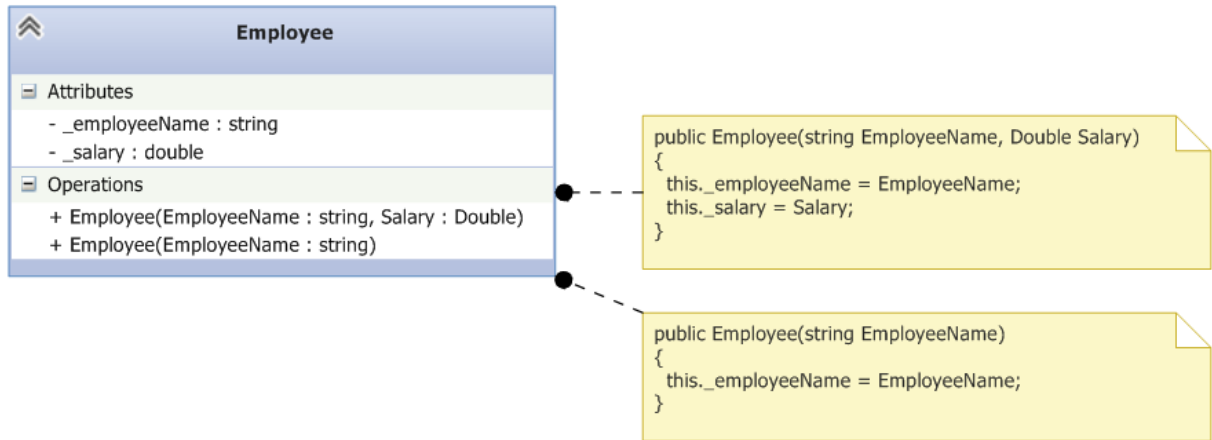


Figura 14 –Encadear Construtores, exemplo antes da refatoração

A Figura 15 mostra a aplicação da refatoração *Encadear Construtores* com base no exemplo anterior.

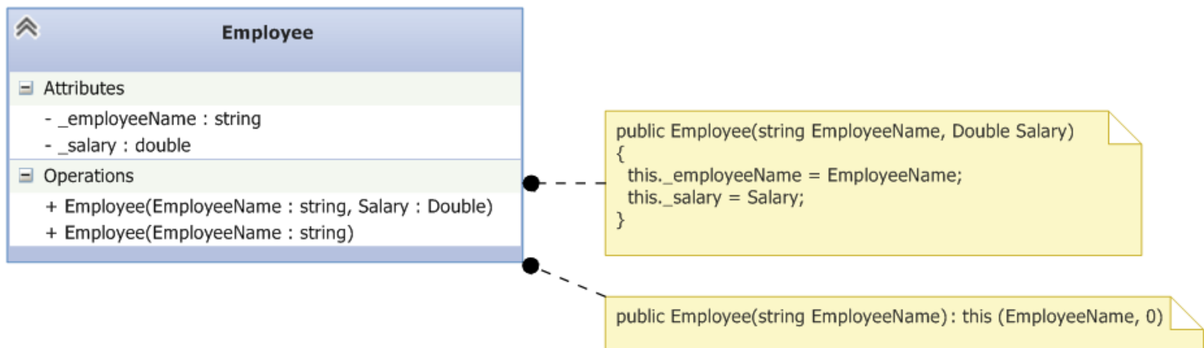


Figura 15 – Encadear Construtores, exemplo depois da refatoração

Os construtores sobrecarregados mais específicos invocam agora o construtor de uso mais geral, repassando os parâmetros apropriados e fornecendo um valor padrão para o que não for informado na inicialização.

2.3. Oportunidades de Refatoração

Uma oportunidade de refatoração pode ser descrita como uma potencial melhoria em um determinado trecho de código de fonte em relação a um atributo de qualidade, ou em um local específico, onde um refatoração pode ser aplicada. Descobrir *onde é o local* apropriado no código-fonte e *qual refatoração* aplicar em um sistema de software não é uma tarefa simples, e baseia-se na experiência do desenvolvedor.

Assim, o uso de refatoração como uma abordagem simples estimulou vários esforços para desenvolver abordagens semi-automáticas para detectar falhas de projeto. A aplicação correta de refatorações apropriadas em um determinado contexto aumenta a qualidade do projeto, sem alterar o seu comportamento. No entanto, a identificação de inconsistências no

código fonte não é uma tarefa simples, tais como métodos, fragmentos de métodos e atributos que devem ser movidos para outras classes.

A motivação para melhorar o projeto de um sistema de software é encontrar problemas no código fonte e usar refatoração como uma possível solução. As limitações (*bad smells*) descrevem problemas e uma lista de refatorações relacionadas que podem ajudar a melhorar o código. A refatoração se concentra principalmente no tratamento dessas limitações, mas a implementação de melhorias depende das habilidades do desenvolvedor, que realiza manutenções de software. Encontrar limitações pode envolver inspecionar todo o código fonte, o que pode se tornar impraticável para sistemas de médio e grande porte. Neste cenário, o apoio semi-automático para detectar falhas é essencial.

Usando algumas dessas técnicas, Refactoring Browser (ROBERTS, 1999) foi uma das primeiras ferramentas para fornecer suporte semi-automático para aplicar refatorações. Hoje em dia a maioria dos ambientes de desenvolvimento (IDEs) fornece algum suporte para refatoração, o que pode reduzir o esforço.

Algumas abordagens semi-automáticas (MUNRO, 2005; SIMON; STEINBRUCKNER; LEWERENTZ, 2001) tentam sugerir melhorias usando métricas para identificar lugares onde uma refatoração pode ser necessária, sendo que os desenvolvedores são responsáveis por determinar quais mudanças devem ser feitas com precisão. Buscar por oportunidades de refatoração para aplicação de padrões de projeto se torna assim uma técnica interessante para ser utilizada no ciclo de vida de sistemas de software orientados a objetos.

2.4. Trabalhos Relacionados

Existem muitas pesquisas e esforços para otimizar a busca semi-automática por oportunidades de refatoração. Mens e Tourwe apresentam um amplo panorama das pesquisas existentes no campo da refatoração de sistemas de software (MENS; TOURWÉ, 2004).

Tsantalis e Chatzigeorgiou propõem um método para identificar oportunidades para a aplicação da refatoração *Mover Método* (FOWLER, 1999), com o objetivo de minimizar a limitação *Feature Envy* (FOWLER, 1999). A abordagem baseia-se no algoritmo que calcula a distância entre entidades, tais como atributos, métodos e classes (TSANTALIS; CHATZIGEORGIOU, 2009).

Trabalho semelhante foi realizado por Simon que define uma métrica que mede a coesão entre os atributos e métodos baseados na sua distância (SIMON et al., 2001). O principal objetivo é identificar os métodos que usam características de outras classes no

sistema. Os resultados das distâncias calculadas são exibidos em uma perspectiva tridimensional, o que ajuda o desenvolvedor a identificar manualmente oportunidades de refatoração, como *Mover Atributo, Mover Método* e *Extrair Classe* (FOWLER, 1999).

El-Sharqwi propõe uma abordagem para refatorar um modelo de software utilizando padrões de projeto. Os autores sugerem uma estrutura XML que contém um projeto com problema, regras de transformação (refatorações), e um modelo de estrutura que consiste na aplicação de padrões (EL-SHARQWI; MAHDI; EL-MADAH, 2010).

Da mesma forma, Kim apresenta uma abordagem para refatorar modelos de software usando padrões para melhorar a qualidade por meio de três componentes: um problema, uma transformação e uma solução (KIM, 2008).

Balazinska propõe um método para refatorar sistemas de software orientados a objetos por meio da identificação de clones em código-fonte, como duplicação (BALAZINSKA, 2000).

Mens e Tourwe mostram como o suporte automatizado pode ser usado para identificar oportunidades de refatoração para detectar limitações, como um parâmetro obsoleto ou uma interface inadequada, mas eles não indicam se um padrão de projeto poderia ser aplicado (TOURWÉ; MENS, 2003).

Seng propõe um método baseado em buscas, o que pode ser útil a ajudar um engenheiro de software a melhorar a estrutura de um sistema de software (SENG et al., 2006). Isso é feito por meio da sugestão de uma lista de refatoração usando algoritmos evolutivos que simulam essas refatorações. Pode ajudar na tarefa de definir a refatoração para melhorar a estrutura de classe em sistemas orientados a objetos. Também detecta o bad smell *Classe Larga* (FOWLER, 1999).

Tekin define uma abordagem baseada na mineração de grafos, com a finalidade de detectar estruturas semelhantes em sistemas orientados a objetos, que podem fornecer informações úteis sobre o projeto, tais como padrões comumente utilizados, bem como defeitos de projeto e clones (TEKIN et al., 2012).

Piveta fornece um processo detalhado para refatoração, incluindo um mecanismo para a seleção e criação de modelos de qualidade, a seleção de padrões de refatoração, a criação e utilização de regras heurísticas, a busca de oportunidades de refatoração e priorização, a avaliação dos efeitos da refatoração na qualidade de software, e, finalmente, a análise de impacto da aplicação de refatorações. Este trabalho estende tal pesquisa a fim de identificar oportunidades de refatoração para aplicar padrões de projeto (PIVETA, 2009).

Pauli e Piveta apresentam uma abordagem baseada em funções heurísiticas e o uso de ASTs para detectar oportunidades para a aplicação do padrão de projeto Strategy, reduzindo complexidade relacionada à lógica condicional. São utilizadas métricas como IM e CC para avaliar os resultados obtidos após a refatoração para os padrões sugeridos (PAULI; PIVETA, 2014).

O'Keeffe e Cinneide apresentam dois trabalhos relacionados com a refatoração baseada em buscas (O'KEEFFE; CINNÉIDE, 2008). Os autores propõem uma pesquisa semi-automática para oportunidades de refatoração, como *Tornar uma Superclasse Abstrata* e *Substituir Herança por Delegação* (FOWLER, 1999).

Com base na observação dos trabalhos relacionados existentes, encontramos muitos trabalhos que definem abordagens para detectar alguma oportunidade para aplicar algumas refatorações primitivas, como *Mover Método* ou *Extrair Classe* (FOWLER, 1999), mas nenhum deles têm um padrão de projeto como resultado final da transformação. Por exemplo, se algumas oportunidades de refatoração primitivas fossem combinadas, isso poderia indicar a aplicação de um padrão de projeto, como Adapter, Template Method, Strategy ou Observer (GAMMA et al., 1999).

Por exemplo, se um método é movido para uma classe, sendo esta uma classe base abstrata e este método pode ser substituído em classes descendentes para definir comportamentos diferentes para o mesmo resultado, um padrão Strategy poderia ser sugerido como resultado final da refatoração. Se dois métodos em subclasses executam passos semelhantes na mesma ordem, um Template Method poderia ser sugerido.

Além disso, muitas pesquisas propõem aumentar a qualidade de sistemas de software baseando-se na aplicação de alguns padrões de projeto no código fonte, mas elas não fornecem uma ferramenta automática de apoio à busca. Com base nisso, estendemos as abordagens citadas para i) propor uma ferramenta automática para apoiar os esforços para localizar onde a refatoração pode ser aplicada, ii) sugerir uma refatoração para um padrão de projeto (como Template Method, Strategy e Factory Method) em vez de simplesmente sugerir uma refatoração primitiva (como *Mover Método*).

3. BUSCA POR OPORTUNIDADES DE REFATORAÇÃO PARA APLICAÇÃO DE PADRÕES DE PROJETO

Este capítulo descreve um conjunto de atividades para buscar por oportunidades de refatoração para a aplicação de padrões de projeto. Em um primeiro momento, são apresentadas as atividades envolvidas na definição de uma busca, incluindo etapas como a identificação de indícios, escolha das métricas apropriadas para medir o impacto da refatoração escolhida, bem como a definição de um padrão a ser aplicado. Também são apresentados alguns refinamentos para algumas limitações em estruturas de código fonte. E finalmente, são apresentadas as mecânicas de cada busca por oportunidade, incluindo passos relizados pelas funções heurísticas implementadas e apresentadas a seguir.

Uma mecânica descreve os passos em alto-nível utilizados por uma função heurísitica para chegar a uma oportunidade de refatoração, com base em indícios. Descreve como um código é analisado a fim de se obterem os resultados desejados, por exemplo, como a função deve iterar sobre elementos de um conjunto, como elementos são comparados, como indícios são observados, entre outros. Em outras palavras, a mecânica funciona como um “pseudo-código” para uma futura implementação da função usando uma determinada linguagem, no caso deste trabalho, a linguagem C#.

3.1. Definição das atividades para a criação de heurísticas de busca

Esta seção descreve a nossa abordagem para detectar automaticamente limitações no código utilizando funções heurísticas e se estes problemas podem sugerir uma possível oportunidade de refatoração, avaliada por funções heurísticas.

De forma a criar heurísticas que permitam a busca por oportunidades de aplicação de padrões, definimos um conjunto de passos que podem ser usados para auxiliar nessa tarefa:

1. Determinação de critérios (indícios) que ajudem a identificar oportunidades em código fonte;
2. Determinação do padrão de projeto que auxilia na resolução da limitação por meio de uma refatoração;
3. Enumeração das motivações para a escolha do padrão;
4. Definição da refatoração adequada para o padrão escolhido;
5. Definição da mecânica que define os passos a serem executados a fim de identificar uma possível oportunidade de refatoração para um padrão de projeto;

6. Definição de um conjunto de métricas que ajudem a medir o impacto da refatoração no sistema de software;
7. Implementação da função heurística usando as mecânicas;
8. Utilizando a implementação, avaliar a função em um conjunto de sistemas de software;
9. Nos locais indicados como uma possível oportunidade, aplicar a refatoração sugerida;
10. Certificar que a refatoração manteve o comportamento externo observável, por exemplo, rodando os testes fornecidos pelos sistemas de código aberto;
11. Avaliar a melhoria dos das métricas no sistema de software.

A Figura 16 mostra o diagrama das atividades envolvidas, divididas em duas etapas: definição e implementação.

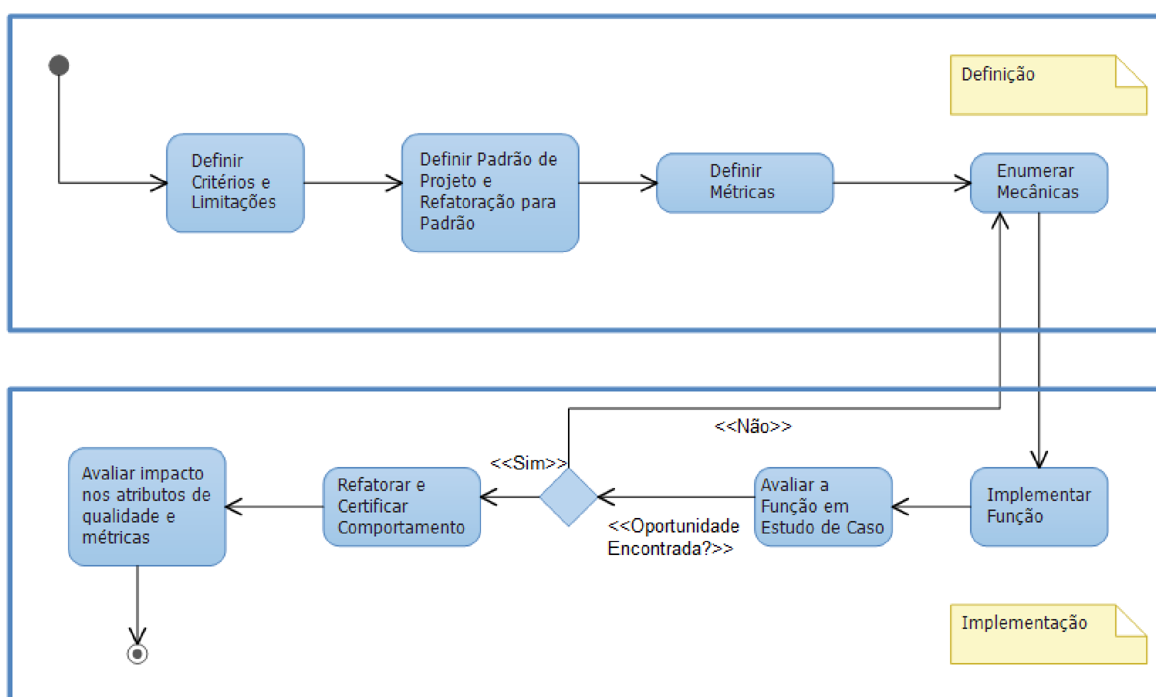


Figura 16 – Atividades envolvidas no processo de buscas por oportunidades de refatoração

O objetivo dessas atividades é auxiliar desenvolvedores a definir novas heurísticas para a busca de oportunidades para aplicar padrões de projetos. Neste processo não está incluída a atividade de aplicação da refatoração de forma automática, devendo esta ser feita pelo desenvolvedor utilizando o suporte oferecido pelos IDEs de desenvolvimento.

3.2. Buscando oportunidades para aplicar o padrão Strategy

A motivação para usar o padrão Strategy se deve ao fato de que ele simplifica expressões condicionais, como if e switch / cases, eliminando a limitação *Sentenças Switch*. Este tipo de construção pode gerar custos e esforços no processo de manutenção. Usando polimorfismo e o princípio "Programação para uma interface / abstração" (GAMMA et al., 1999), o padrão Strategy é capaz de auxiliar na eliminação de testes condicionais.

Mecânica:

1 – Receber como entrada um conjunto de arquivos representando a AST do código fonte, com um conjunto de classes CJ;

2 – Testar para cada classe C de CJ, se a mesma possui sentenças condicionais SC, como blocos switch;

3 – Para cada teste condicional TC na SC, verificar se ela usa type codes, como enumerações, inteiros, strings ou outros tipos primitivos;

4 – Armazenar o número total de testes condicionais TC;

5 – Armazenar o número total de linhas no bloco condicional SC;

6 – Medir a complexidade do bloco condicional usando as variáveis TC e SC comparando com uma constante de complexidade definida pelo desenvolvedor;

7 – Retornar uma interpretação positiva se a sentença condicional SC for avaliada como complexa e utilizar type codes;

Na Figura 17 pode ser visualizado um exemplo que apresenta, em alto nível, os indícios usados pela função heurística para avaliar a oportunidade de aplicação de um padrão Strategy. O exemplo mostra que uma classe precisa serializar um objeto que representa um *cliente*. Então, por meio do método *Serializar* ela testa um tipo enumerado para descobrir qual o formato a ser utilizado (um *type code*). Algoritmos específicos então se encarregam de persistir este objeto em diferentes formatos. A função analisa o tamanho do bloco, o número de testes e se a condicional usa *type codes*.

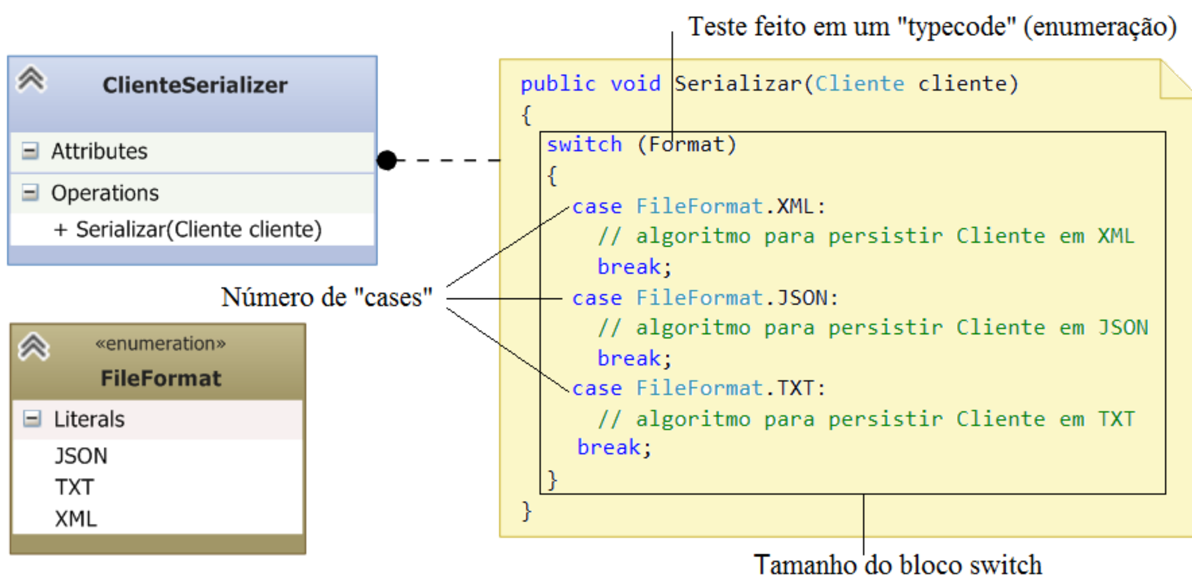


Figura 17 – Análise da complexidade condicional com base no tamanho do bloco, número de testes e uso de "*type codes*"

3.3. Buscando oportunidades para aplicar o padrão Factory Method

Uma motivação para usar o padrão Factory Method é devido ao fato de que ele simplifica testes condicionais quando esses testes são utilizados para escolher qual o tipo de objeto precisa ser criado. Usando polimorfismo, o padrão Factory Method é capaz de eliminar condicionais redundantes e complexas. Outra motivação é devido ao fato de que o Factory Method pode reduzir a duplicação de código quando usado em combinação com o padrão Template Method.

Mecânica:

1 – Receber como entrada um conjunto de arquivos representando a AST do código fonte, com um conjunto de classes CJ;

2 - Testar se uma classe C contida em CJ tem sentenças condicionais SC, como blocos switch;

3 - Para cada teste condicional TC na sentença SC, verificar se ele usa *type codes*, como enumerações, inteiros, strings ou outros tipos primitivos;

4 - Para cada teste condicional TC na sentença SC, verificar se ele cria um objeto de um determinado tipo T;

5 - Verificar e armazenar a classe base CB do tipo T;

6 - Verificar se outros testes condicionais na mesma instrução condicional SC criam objetos de uma mesma classe base CB;

7 - Retorna uma interpretação positiva se a instrução condicional SC cria *n* objetos de uma mesma classe base BC;

Na Figura 18 pode ser visualizado um exemplo que apresenta, em alto nível, os indícios usados pela função heurística para avaliar a oportunidade de aplicação de um padrão Factory Method. Neste caso, uma sentença switch está testando um tipo primitivo (string) a fim de decidir qual objeto deve instanciar. Todos os objetos criados e retornados são de tipos que possuem uma mesma classe base. Neste exemplo, a sentença condicional basicamente faz o mapeamento de um tipo primitivo para um tipo específico.

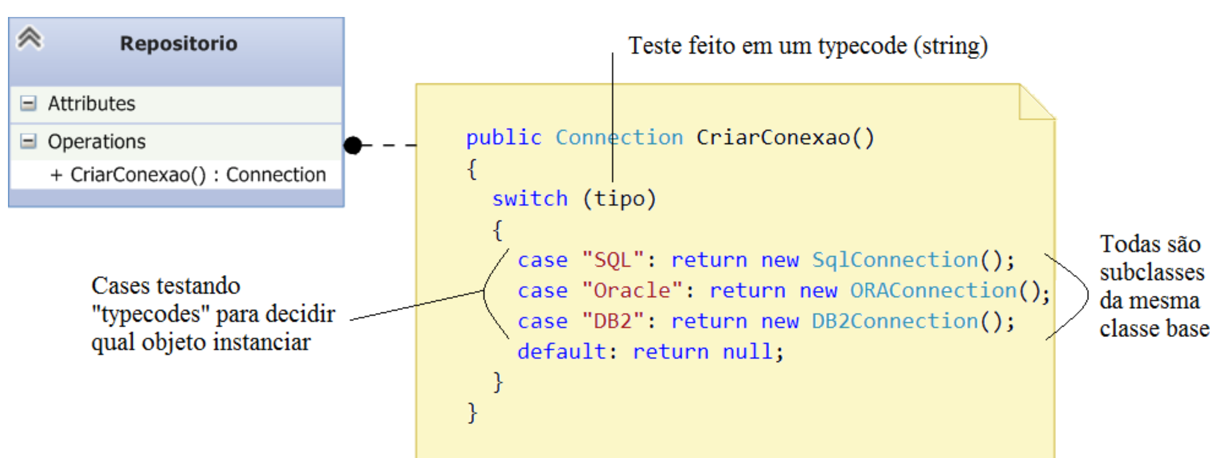


Figura 18 – Análise do bloco switch usando “type codes” para criar subclasses

3.4. Buscando oportunidades para aplicar o padrão Template Method

Uma motivação para aplicar o padrão de projeto Template Method é o fato de que este padrão pode reduzir a duplicação de código em corpo de métodos, que pode ser uma sequência idêntica de chamadas de método em subclasses, por exemplo. Esse tipo de código-fonte gera custos e esforços no processo de manutenção.

Mecânica:

1 – Receber como entrada um conjunto de arquivos representando a AST do código fonte, com um conjunto de classes CJ;

2 - Verificar se cada classe C de CJ tem subclasses SC;

3 - Iterar todas as subclasses SC da classe C;

4 - Iterar todos os métodos M de cada subclasse SC;

5 - Guardar em uma lista L todas as declarações encontradas no corpo de cada método M;

6 - Iterar a lista L, iniciando a busca por possíveis duplicações de código;

7 - Para cada elemento E na lista L, gerar duas listas L1 e L2, contendo todas as declarações a serem comparadas;

8 - Gerar um subconjunto S1 da lista L1, contendo todas as possíveis sequências de declarações, começando com dois elementos, até o tamanho máximo de L1;

9 - Gerar um subconjunto S2 da lista L2, que contém todas as possíveis sequências de declarações, começando com dois elementos, até o tamanho máximo de L2;

10 - Comparar os dois subconjuntos gerados S1 e S2;

11 - Testar se a sequência de instruções em S1 é igual à sequência de instruções em S2.

A figuras apresentadas a seguir mostram como a função heurística trabalha para realizar a comparação entre duas subcoleções de expressões para dois diferentes métodos de diferentes subclasses de uma mesma classe base.

Na Figura 19 pode ser visualizado um exemplo de como a função heurística pode avaliar as duas primeiras posições de uma determinada lista de instruções com todas as posições em outra lista. A função heurística do exemplo não detectou uma oportunidade de refatorar as duas primeiras declarações da segunda lista, mas encontrou uma sequência igual nas duas últimas posições da lista. A função heurística encontrou uma oportunidade para aplicar uma refatoração para o padrão Template Method, porque a sequência de instruções é idêntica para ambas as subclasses.

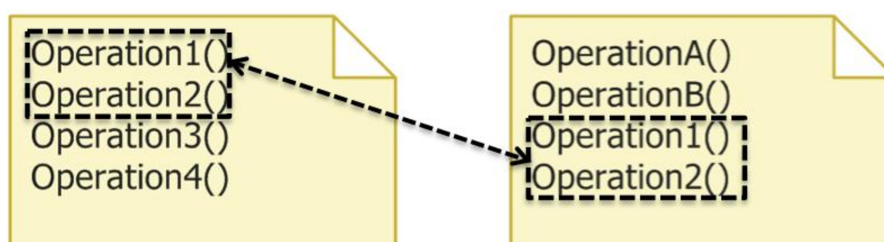


Figura 19 - Comparação de pilhas de chamadas de métodos para detectar uma oportunidade para aplicar o Template Method (Exemplo 1)

Na Figura 20 pode ser visualizado um exemplo de como a função pode detectar mais oportunidades fazendo a iteração por todas as instruções na lista. Três expressões em sequência na primeira lista são comparadas com expressões na segunda lista. A função heurística encontrou uma oportunidade para aplicar o padrão Template Method nas três primeiras posições da segunda lista.

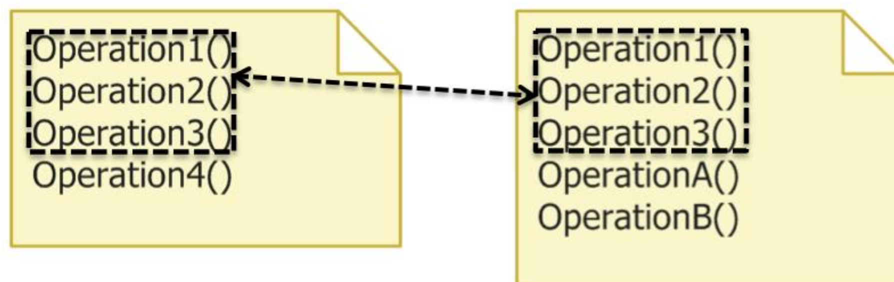


Figura 20 - Comparação de pilhas de chamadas de métodos para detectar uma oportunidade para aplicar o Template Method (Exemplo 2)

Usando o mesmo processo, a função heurística pode avaliar todas as combinações possíveis até o tamanho máximo da lista, começando em dois.

3.5. Buscando oportunidades para aplicar o padrão Creation Method

Uma motivação para aplicar o padrão de projeto Creation Method é a redução da complexidade relacionada à criação de objetos, que pode levar em conta a necessidade de se escolher uma dentre várias versões de um construtor para uma dada classe.

Normalmente, em uma classe que contém n atributos, os desenvolvedores tendem a criar várias versões do construtor, que na maioria das linguagens obriga a utilização do mesmo nome da classe, o que deixa menos clara a intenção de cada construtor. Cada construtor recebe uma sequência de parâmetros, variando em sua quantidade e tipo, caso contrário, o compilador poderia acusar uma ambiguidade. Esses parâmetros normalmente são então utilizados no corpo do construtor para iniciar os atributos da classe. O problema reside em detectar se uma classe possui construtores desnecessários, criando complexidade desenecessária.

Mecânica:

1 – Receber como entrada um conjunto de arquivos representando a AST do código fonte, com um conjunto de classes CJ;

2 – Verificar se cada classe C em CJ possui construtores CS sobrecarregados;

3 – Iterar todos construtores CS sobrecarregados;

4 – Armazenar em uma lista L todos os parâmetros distintos usados nos construtores;

5 – Verificar se cada classe C em CJ possui mais de um construtor CS sobrecarregado;

6 - Testar se a lista de parâmetros L contém mais de um parâmetro distinto;

Na Figura 21 pode ser visualizado um exemplo em alto nível dos indícios usados pela função heurística para avaliar a oportunidade de aplicação de um padrão Creation Method.

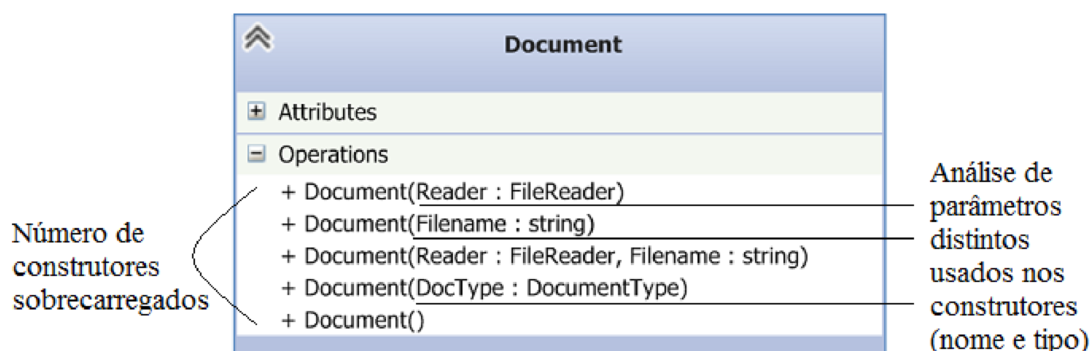


Figura 21 – Análise dos construtores considerando número de sobrecargas e parâmetros

A função considera o número de construtores sobrecarregados, bem como a combinação de parâmetros com mesmo nome e tipo, a fim de buscar por classes que possuam uma longa lista de construtores que possam causar complexidade desnecessária ou mesmo dificultar pela escolha de uma versão do construtor quando for necessária a instanciação.

3.6. Buscando oportunidades para aplicar o padrão Chain Constructors

A busca por oportunidades para aplicar o padrão Chain Constructors por meio da refatoração *Encadear Construtores* consiste em varrer todas as expressões utilizadas em diferentes versões sobrecarregadas de construtores de uma mesma classe, procurando por código inicialização duplicado.

Mecânica:

1 – Receber como entrada um conjunto de arquivos representando a AST do código fonte, com um conjunto de classes CJ;

2 – Iterar todos construtores CS sobrecarregados de cada classe C de CJ;

3 – Armazenar em uma lista L todos os construtores CS e seus respectivos blocos de inicialização (implementação);

4 – Varrer a lista L comparando o construtor atual A com o próximo construtor B da lista L;

5 - Testar se a lista de inicialização do construtor A contém uma expressão idêntica inicializada no construtor B;

O exemplo da Figura 22 apresenta em alto nível os indícios usados pela função heurística para avaliar a oportunidade de aplicação de um padrão Chain Constructors.

```
public class Employee
{
  private string _employeeName;
  private string _department;
  private Double _salary;
  public Employee(string EmployeeName, string Department, Double Salary)
  {
    this._employeeName = EmployeeName;
    this._department = Department;
    this._salary = Salary;
  }
  public Employee(string EmployeeName, string Department)
  {
    this._employeeName = EmployeeName;
    this._department = Department;
  }
}
```

Comparação linha a linha

Código duplicado, atributos já são inicializados no construtor padrão

Figura 22 – Análise de construtores com código de inicialização duplicado

A função considera o número de construtores sobrecarregados, comparando o corpo (*body*) de implementação de código de cada um, procurando por duplicações.

4. IMPLEMENTAÇÃO

Este capítulo apresenta a implementação das funções heurísticas baseadas nas mecânicas previamente apresentadas, usando a linguagem C# e ferramentas de apoio. É apresentado um framework extensível para a implementação das buscas definidas. É apresentada também uma ferramenta gráfica que utiliza esse framework, que permite avaliar projetos e buscar pelas oportunidades de refatoração para aplicação de padrões de projeto.

4.1. A ferramenta AROS

As funções heurísticas propostas foram avaliadas por meio do desenvolvimento de uma ferramenta em C#, chamada AROS[1], que pode identificar de forma automatizada algumas oportunidades de refatoração para padrões, tal como a refatoração *Formar Template Method* ou *Substituir Lógica Condicional por Strategy*. A ferramenta utiliza uma biblioteca de código aberto chamada NRefactory [2], que inclui suporte para a manipulação de código fonte usando-se ASTs.

Uma motivação para o uso de ASTs para a busca por oportunidades de refatoração na abordagem proposta, é que estas estruturas permitem que a ferramenta possa analisar detalhadamente cada elemento da árvore sintática elaborada a partir do código C#. Dessa forma, pode examinar desde simples declarações de classes até expressões mais complexas encontradas dentro da implementação do corpo de métodos, por meio de uma varredura profunda no código. Permite examinar assim expressões que vão desde a criação de objetos, testes condicionais, invocação de métodos, dentro outros, permitindo coletar indícios suficientes para encontar oportunidades, o que provavelmente não poderia ser obtido por meio de outra técnica, como por exemplo, a simples análise da relação entre classes em um diagrama de classes UML, ou diagramas de sequência, visto que estes não expõem todas as informações necessárias para detectar as limitações usadas neste trabalho.

A ferramenta desenvolvida não precisa de integração com um IDE e pode ser executada como um aplicativo independente, sendo possível analisar projetos C# inteiros ou arquivos isolados (se aplicável). A ferramenta lê e processa cada arquivo no projeto, fazendo o processamento de sua AST. Quando uma limitação é identificada na AST, a interface principal da ferramenta fornece uma interpretação e indica se a aplicação de um padrão de projeto é uma solução adequada, usando a função heurística apropriada.

---

[1] O projeto AROS está hospedado no SourceForge no endereço http://sourceforge.net/projects/aros2dp/
[2] https://github.com/icsharpcode/NRefactory

Decidimos não integrar a ferramenta na forma de um plug-in a um IDE em específico, como o Visual Studio, visto que algumas versões do mesmo são de licença comercial, além de isto gerar uma dependência. Uma limitação é que o desenvolvedor precisa obter as informações da ferramenta, como oportunidades encontradas, e localizar as mesmas no IDE usando as opções adequadas. Em contrapartidada, uma possível integração com um um IDE, como o Visual Studio, poderia automatizar algumas tarefas, como localizar no editor o código fonte com a limitação e exibir automaticamente as métricas para a classe selecionada usando a API da ferramenta.

A motiviação para a escolha da linguagem C# se deve ao fato de que a mesma é utilizada em projetos de média e larga escala, mesmo em projetos de código aberto, como o NHibernate, NUnit e NAnt (versões para .NET das ferramentas Hibernate, JUnit e Ant do Java). A linguagem C# é gratuita, com especificação aberta, utilizada tanto em projetos comerciais quanto na comunidade acadêmica. Os projetos avaliados nesta linguagem puderam fornecer indícios suficientes para a detectação de oportunidades para aplicação de refatorações para padrões de projeto.

A Figura 23 mostra a tela principal da ferramenta. Ela permite que o desenvolvedor possa abrir um projeto ou um único arquivo. Após esta etapa, as respectivas ASTs são criadas e são apresentadas graficamente para o usuário para permitir a navegação no código fonte. Quando um nó é selecionado, seu código associado também é selecionado, a fim de ajudar o desenvolvedor na análise do código-fonte.

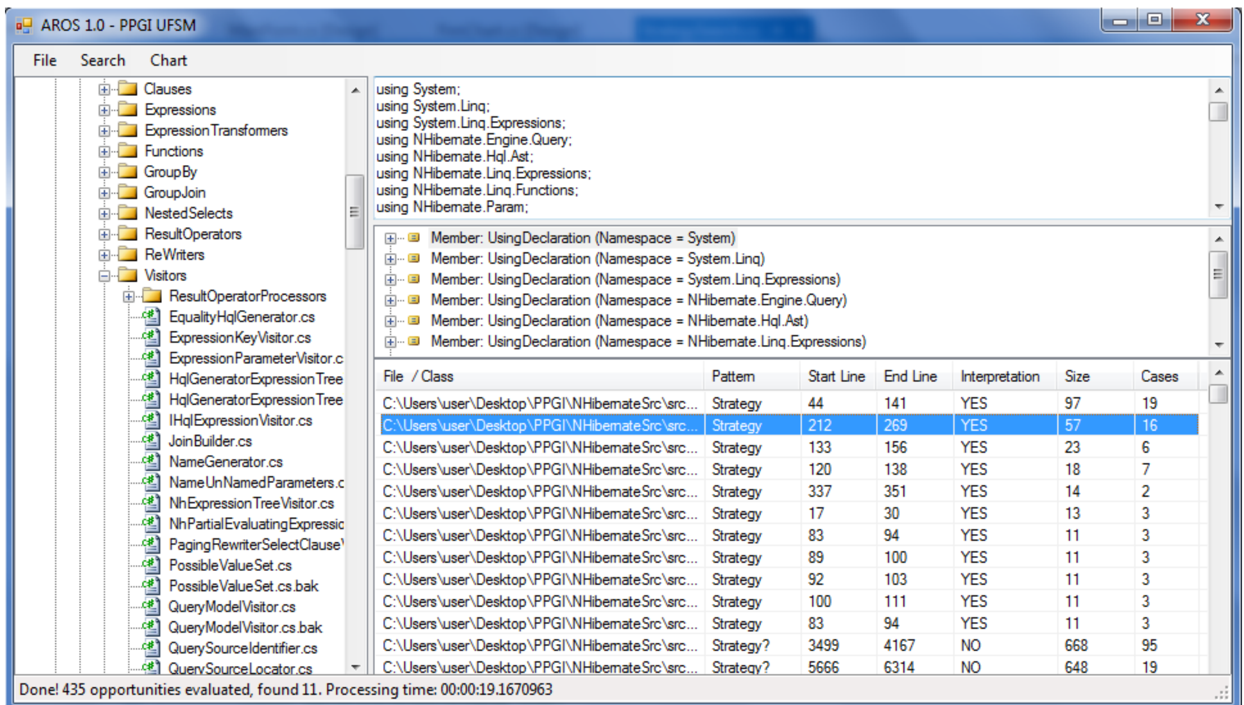


Figura 23 – AROS: uma ferramenta para buscas por oportunidades de refatoração para aplicação de padrões de projeto

Ao processar a busca por oportunidades de refatoração, a ferramenta pode indicar ao final quais arquivos têm limitações que podem sugerir a aplicação de uma refatoração para um padrão de projeto. Neste caso, a ferramenta informa na tela principal uma lista das oportunidades, sendo que cada oportunidade contém informações sobre:

- O nome do arquivo / classe que contém a limitação, como sentenças switch e código duplicado;
- O número da linha de código exata no fonte onde a limitação foi encontrada, ou seja, *o local onde a refatoração pode ser aplicada*;
- Sugestão de *qual padrão de projeto poderia ser aplicado* no local indicado, como Strategy, Template Method ou Factory Method;
- Evidências consideradas na avaliação, como número de expressões condicionais, tamanho do bloco de código, número de linhas repetidas, complexidade na construção de objetos etc.

A Figura 24 mostra as principais classes do framework desenvolvido para buscar por oportunidades de refatoração para padrões de projeto. A classe base *BaseSearch* define um método virtual abstrato que recebe como parâmetro uma AST a ser processada. A classe também contém um atributo que armazena uma lista de oportunidades encontradas, como por exemplo, informações sobre a localização no código fonte e qual padrão foi sugerido.

Subclasses especializadas então podem herdar dessa classe base e implementar o método base de busca por meio de polimorfismo. As classes concretas, como *Strategy*, *FactoryMethod*, *TemplateMethod*, *CreationMethod* e *ChainConstructors*, podem então implementar as funções heurísticas usando as mecânicas propostas, varrendo as ASTs e buscando por oportunidades segundo os indícios considerados.

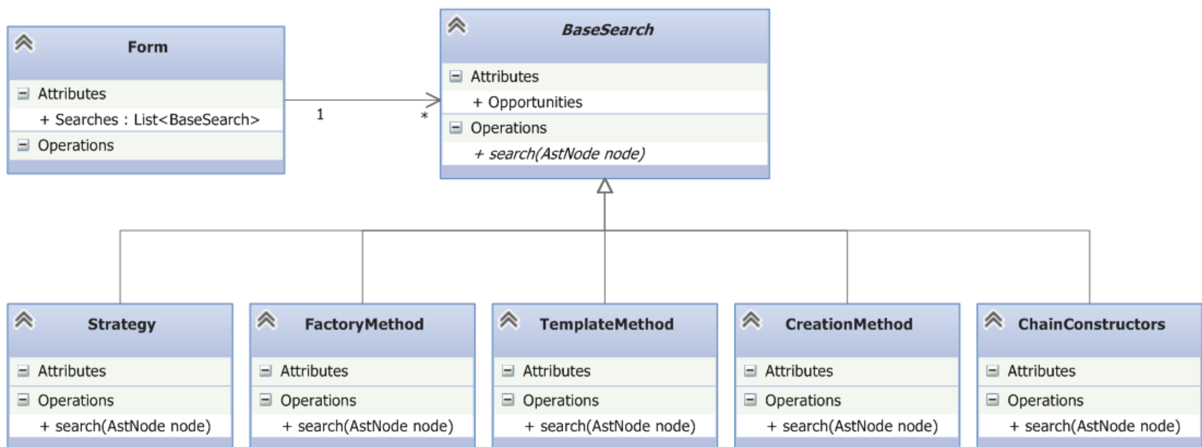


Figura 24 – Framework com classes para buscar oportunidades de refatoração

A interface principal contém uma coleção de objetos de busca, que são instanciados e adicionados dinamicamente à coleção. Dessa forma, é possível escolher quais padrões podem ser considerados na busca. O formulário referencia uma coleção de buscas abstratas e resolve qual padrão procurar pela aplicação de polimorfismo no método *Search*, o que pode ser considerado como a aplicação do próprio padrão Strategy. Aqui a estratégia é a função heurística que encapsula o algoritmo utilizado na busca, permitindo que este mesmo framework possa ser estendido para usar outros padrões.

4.2. Funções para detectar oportunidades para aplicar padrões de projeto

As funções apresentadas nas seções a seguir implementam, usando a linguagemn C#, as mecânicas previamente apresentadas para detectar oportunidades de refatoração para cada um dos padrões considerados. Alguns passos são comuns a todas funções e são listados aqui.

Cada uma das funções recebe como parâmetro uma referência para um nó na AST que está sendo atualmente processada pela ferramenta AROS. A classe principal que representa a interface de usuário do aplicativo envia a esta função a respectiva AST para cada arquivo processado no projeto. A partir daí, cada função itera sobre os tipos (classes) recebidos nesta representação AST, buscando por oportunidades de refatoração com base nos indícios definidos nas mecânicas.

4.3. Função para detectar oportunidades para aplicar o padrão Strategy

A função da Figura 25 faz um laço na AST recebida em busca de elementos que representem sentenças switch (linha 03). A seguir, a função conta quantas expressões de teste *case* estão sendo feitas neste bloco e armazena em uma variável para futura avaliação (linha 05).

```
01. public override bool search(AstNode node)
02. {
03.   foreach (AstNode child in node.Descendants.OfType<SwitchStatement>())
04.   {
05.     int numberOfCases = child.Descendants.OfType<CaseLabel>().ToList().Count;
06.     int sizeOfSwitch = child.EndLocation.Line - child.StartLocation.Line;
07.     bool isTypeCode = SwitchUsesTypeCode(child);
08.     if (((sizeOfSwitch / numberOfCases) > switchComplexityIndex) && (isTypeCode))
09.       return true;
10.   }
11.   return false;
12. }
```

Figura 25 – Função para detectar oportunidades para aplicar o padrão Strategy

De forma semelhante, o tamanho total do bloco switch é armazenado em uma variável (linha 06). A seguir, um teste é feito para verificar se este switch está fazendo teste em um tipo código (type code), como strings, enumerações ou inteiros (linha 07). A função então faz uma avaliação nestas informações, verificando o nível de complexidade do bloco switch com base no número de testes e tamanho total do bloco, e se o mesmo está usando type codes (linha 08).

Se o teste for verdadeiro, isso indica que *o bloco switch contém lógica complexa fazendo testes em type codes para decidir pela execução de algoritmos*, uma oportunidade para aplicar um padrão Strategy.

4.4. Função para detectar oportunidades para aplicar o padrão Factory Method

A função da Figura 26 faz um laço na AST buscando elementos que representem sentenças switch (linha 03). A seguir, um teste é feito para verificar se este switch está fazendo teste em um tipo código (type code), como strings, enumerações ou inteiros (linha 05). Uma coleção do tipo *Dictionary* (dicionário) é criada para armazenar então todas as expressões *case* a serem processadas no bloco switch (linha 06). Para cada teste encontrado, a função verifica se este contém expressões que criam objetos (linha 07).

```
01. public override bool search(AstNode node)
02. {
03.   foreach (AstNode testNode in node.Descendants.OfType<SwitchStatement>())
04.   {
05.     if (!SwitchUsesTypeCode(testnode)) continue;
```

```
06.     Dictionary<string, string> subClasses = new Dictionary<string, string>();
07.     foreach (AstNode objCreationNode in testNode.Descendants.OfType<ObjectCreateExpression>())
08.     {
09.       var targetClass = (objCreationNode as ObjectCreateExpression).Type.ToString();
10.       var baseClass = ClassHierarchyHelper.getBaseClass(targetClass.ToString());
11.       subClasses.Add(targetClass, baseClass);
12.     }
13.     removeClasses(subclasses);
14.     return (subClasses.Count >= 2);
15.   }
16.   return false;
17. }
```

Figura 26 - Função para detectar oportunidades para aplicar o padrão Factory Method

A função armazena então informações sobre o que este bloco *case* está criando, como o tipo de objeto (linha 09) e a sua classe base (linha 10). A seguir, adiciona essas informações à coleção previamente criada (linha 11). A função precisa agora fazer uma verificação nessa coleção que até agora basicamente armazenou informações sobre objetos criados dentro deste bloco switch. Essa varredura remove todas as classes que não têm a mesma classe base, ou seja, mantém somente subclasses que foram criadas no switch que têm a mesma classe base (linha 13).

Se ao final ficarem pelo menos duas classes nessa coleção (linha 14), isso indica que *o bloco switch está fazendo um teste em um type code para decidir pela criação de classes de uma mesma classe base*, uma oportunidade para aplicar um padrão Factory Method.

4.5. Função para detectar oportunidades para aplicar o padrão Template Method

A função na Figura 27 faz um laço na AST recebida buscando elementos que representem tipos, como classes (linha 03). Para cada tipo encontrado, a função verifica se este tipo é uma classe (não considera interfaces ou enumerações) e se essa classe tem subclasses (linha 06).

```
01. public override bool search(AstNode node)
02. {
03.   foreach (AstNode n in node.Descendants.OfType<TypeDeclaration>())
04.   {
05.     TypeDeclaration c = n as TypeDeclaration;
06.     if (c.ClassType != ClassType.Class || !ClassHierarchyHelper.classHasSubClasses(c)) continue;
07.     List<StackInfo> subClassMethodsStatements = new List<StackInfo>();
08.     foreach (TypeDeclaration sc in ClassHierarchyHelper.getSubClasses(c))
09.     {
10.       StackInfo scInfo = new StackInfo();
11.       scInfo.subClass = sc;
12.       foreach (MethodDeclaration method in sc.Descendants.OfType<MethodDeclaration>())
13.       {
14.         List<Expression> statements = new List<Expression>();
15.         foreach (Expression statement in method.Descendants.OfType<Expression>())
16.           statements.Add(statement);
17.         if (statements.Count == 0) continue;
18.         scInfo.Statements.Add(method, statements);
19.       }
20.       subClassMethodsStatements.Add(scInfo);
21.     }
```

```
22.     if (subClassMethodsStatements.Count == 0) continue;
23.     for (int iActualSubClass = 0; iActualSubClass < subClassMethodsStatements.Count;
iActualSubClass++)
24.     {
25.       StackInfo actual = subClassMethodsStatements[iActualSubClass];
26.       for (int iNextSubClass = iActualSubClass + 1; iNextSubClass <
subClassMethodsStatements.Count; iNextSubClass++)
27.       {
28.         StackInfo next = subClassMethodsStatements[iNextSubClass];
29.         for (int iActualStatement = 0; iActualStatement < actual.Statements.Count;
iActualInvocation++)
30.         {
31.           var actualStaments = actual.Statements.ElementAt(iActualStatement);
32.           for (int iNextStatement = 0; iNextStatement < next.Statements.Count; iNextStatement++)
33.           {
34.             var nextStaments = next.Statements.ElementAt(iNextStatement);
35.             List<string> actualCallStackStrList = getMethodStatements(actualStaments.Key);
36.             List<string> nextCallStackStrList = getMethodStatements(nextStaments.Key);
37.             for (int iActualPartialCallStack = 0; iActualPartialCallStack <
actualCallStackStrList.Count - 1; iActualPartialCallStack++)
38.             {
39.               for (int iSizeCallStackToCompare = 2; iSizeCallStackToCompare <=
actualCallStackStrList.Count - iActualPartialCallStack; iSizeCallStackToCompare++)
40.               {
41.                 List<string> actualPartialCallStack =
actualCallStackStrList.GetRange(iActualPartialCallStack, iSizeCallStackToCompare);
42.                 for (int iNextPartialCallStack = 0; iNextPartialCallStack <
nextCallStackStrList.Count - 1; iNextPartialCallStack++)
43.                 {
44.                   if (iSizeCallStackToCompare > nextCallStackStrList.Count - iNextPartialCallStack)
continue;
45.                   List<string> nextPartialCallStack =
nextCallStackStrList.GetRange(iNextPartialCallStack, iSizeCallStackToCompare);
46.                   return (actualPartialCallStack.SequenceEqual(nextPartialCallStack));
47.                 }
48.               }
49.             }
50.           }
51.         }
52.       }
53.     }
54.   }
55.   return false;
56. }
```

Figura 27 - Função para detectar oportunidades para aplicar o padrão Template Method

Se a classe tem subclasses, a função processa cada uma delas (linha 08) para procurar os códigos duplicados no corpo de cada um de seus métodos. Para cada subclasse processada, a função armazena todos os seus métodos em uma coleção (criada na linha 07). Para cada método, a função armazena todas as declarações encontradas em seu corpo (linhas 14 a 18). Essa coleção é usada para futura comparação de código duplicado. Após o processamento de todas as subclasses, a função precisa testar se existe código duplicado, fazendo uma comparação de expressões por similaridade.

Após o armazenamento de todas as subclasses de uma determinada superclasse, é necessário iterar as ASTs em busca de sequências idênticas de expressões que são duplicadas nessas subclasses (linha 23). Para isso, a pesquisa é feita em todas as subclasses armazenadas anteriormente. Para cada subclasse processada, um laço é feito para permitir a comparação

com a subclasse subsequente na coleção (linha 26), fazendo assim uma comparação de todas as sequências possíveis, uma por uma. Uma coleção de expressões é obtida para cada método de cada subclasse processada (linha 31), que é comparada com as expressões de outra subclasse (linha 34).

Finalmente, um subconjunto com as expressões (linha 41) é comparado com outro subconjunto (linha 45). A criação de subcoleções de expressões com todas as combinações possíveis dos passos é necessária devido ao fato de a sequência poder ser formada por *n* passos (2, 3, 4 ou mais).

Se a função encontra uma sequência de expressões idênticas na lista (linha 46), isso indica que *dois métodos em diferentes subclasses e com a mesma classe base têm código duplicado* (ou seja, a mesma sequência de instruções), o que poderia formar um método comum na classe base. Esta é uma oportunidade para aplicar um Template Method, porque a mesma sequência de instruções é duplicada em duas subclasses.

4.6. Função para detectar oportunidades para aplicar o padrão Creation Method

A função na Figura 28 busca na AST recebida elementos que representem tipos como classes (linha 03). Para cada classe encontrada, a função cria uma lista (linha 05) para armazenar os parâmetros distintos do conjunto de construtores da classe sendo processada.

```
01. public override bool search(AstNode node)
02. {
03.   foreach (AstNode child in node.Descendants.OfType<TypeDeclaration>())
04.   {
05.     List<ParameterDeclaration> DistinctParameters = new List<ParameterDeclaration>();
06.     int numberOfConstructors = 0;
07.     foreach (AstNode ctor in child.Descendants.OfType<ConstructorDeclaration>())
08.     {
09.       if (ctor.Parent != child) continue;
10.       ConstructorDeclaration ctord = ctor as ConstructorDeclaration;
11.       foreach (var param in ctord.Parameters)
12.         if (!DistinctParameters.Contais(param))
13.           DistinctParameters.Add(param);
14.        numberOfConstructors++;
15.     }
16.     return ((numberOfConstructors >= 2) && (DistinctParameters.Count >= 2))
18.   }
19.   return false;
20. }
```

Figura 28 - Função para detectar oportunidades para aplicar o padrão Creation Method

Uma varredura é feita em todos os construtores da classe (linha 07). Se o construtor sendo processado não pertencer à classe atual, por exemplo, de uma classe interna (*inner class*), este é descartado (linha 09). A função então obtém informações sobre o construtor (linha 10), e faz uma varredura em todos os seus parâmetros (linha 11). Para cada parâmetro encontrado neste construtor, a função verifica se o mesmo está na lista de parâmetros distintos

(linha 12), caso contrário, adiciona este parâmetro na coleção (linha 13) e incrementa a variável que armaneza o número total de construtores encontrados naquela classe (linha 14).

Como resultado final, a função retorna como uma possível oportunidade *se a classe tem mais de um construtor sobrecarregado usando mais de um tipo de parâmetro distinto, formado pela combinação do nome e tipo* (linha 16).

4.7. Função para detectar oportunidades para aplicar o padrão Chain Constructors

A função na Figura 29 busca na AST recebida elementos que representem tipos como classes (linha 03). Para cada classe encontrada, a função cria uma lista (linha 05) para armazenar as informações sobre o conjunto de construtores, como a assinatura e expressões encontradas no corpo da sua implementação.

```
01. public override bool search(AstNode node)
02. {
03.   foreach (AstNode Class in node.Descendants.OfType<TypeDeclaration>())
04.   {
05.     List<ConstructorInfo> ClassConstructors = new List<ConstructorInfo>();
06.     foreach (AstNode Constructor in Class.Descendants.OfType<ConstructorDeclaration>())
07.     {
08.       if (Constructor.Parent != Class) continue;
09.       ConstructorInfo ConstructorInfo = new ConstructorInfo();
10.       ConstructorInfo.Constructor = Constructor as ConstructorDeclaration;
11.       foreach (AstNode Body in Constructor.Descendants.OfType<BlockStatement>())
12.         foreach (AstNode Statement in Body.Descendants.OfType<Statement>())
13.           ConstructorInfo.Statements.Add(Statement.ToString());
14.       ClassConstructors.Add(ConstructorInfo);
15.     }
16.     for (int iActualConstructor = 0; iActualConstructor < ClassConstructors.Count;
iActualConstructor++)
17.     for (int iNextConstructor = iActualConstructor + 1; iNextConstructor < ClassConstructors.Count;
iNextConstructor++)
18.       foreach (string Statement in ClassConstructors[iActualConstructor].Statements)
19.         if (ClassConstructors[iNextConstructor].Statements.Contains(Statement))
20.           return true;
21.   }
22.   return false;
23. }
```

Figura 29 - Função para detectar oportunidades para aplicar o padrão Chain Constructors

Uma varredura é feita em todos os construtores da classe (linha 06). Se o construtor sendo processado não pertencer à classe atual, este é descartado (linha 08). A função então obtém informações sobre o construtor (linha 10), e faz uma varredura em todas as expressões encontradas em sua implementação (linha 12). Para cada expressão encontrada neste construtor, a função a adiciona em uma coleção (linhas 13 e 14).

Tendo armazenado todas as informações necessárias sobre os construtores sobrecarregados na classe, a partir da linha 16 é iniciada uma busca por código de implementação duplicado. Isso é feito por meio de dois laços (linhas 16 e 17), que permitem

comparar as informações do construtor atualmente sendo processado com o posterior. Para cada expressão encontrada no corpo de implementação do construtor atual (linha 18), a função testa se existe uma equivalência no construtor seguinte (linha 19), retornando nesse caso uma interpretação positiva.

Isso indica que *dois ou mais construtores contêm código duplicado em sua implementação*, uma oportunidade para eliminar a duplicação encadeando-se os construtores.

5. ESTUDO DE CASO

A fim de investigar o uso das funções heurísticas propostas, foram analisados alguns projetos de código fonte aberto. A coleta de projetos heterogêneos, com diferentes tamanhos e aplicações, proporcionou avaliar uma maior variedade de diferentes estilos de codificação, limitações e estruturas de código com problemas e, portanto, uma maior cobertura na avaliação proposta. A Tabela 1 apresenta as ferramentas avaliadas, bem como suas versões e tamanhos em número de linhas de código.

Tabela 1 – Projetos utilizados na avaliação

Sistema de Software	Versão	Tamanho em linhas de código
NUnit	2.6.2	27.825
NAnt	0.92	27.375
NHibernate	3.3.3	402.634

Os experimentos foram realizados utilizando três sistemas de software de código aberto: NUnit[3], NAnt[4] e NHibernate[5]. NUnit é uma ferramenta para a criação de testes de unidade. NAnt é uma ferramenta de linha de comando usada no processo de desenvolvimento fornecendo mecanismos automatizados para compilar, testar e executar aplicativos. NHibernate é uma ferramenta para o objeto-relacional mapeamento e persistência de objetos.

Em muitos casos é impraticável mostrar todos os resultados devido ao grande volume de classes envolvidas em cada avaliação dos sistemas de software. Neste caso, apresentamos e discutimos os resultados mais relevantes para cada oportunidade de busca de acordo com o padrão sugerido. Para cada seção que apresenta as oportunidades encontradas para cada padrão de projeto, mostramos quantas oportunidades foram selecionadas e apresentadas neste trabalho a partir de um conjunto maior que compõe o total de oportunidades sugeridas pela ferramenta.

Os critérios considerados para a seleção consideram basicamente a escolha das classes com as maiores deficiências nos atributos de qualidade considerados, como limitações em sentenças condicionais (maior número de expressões de teste), maior número de linhas de código duplicado, índice de manutenção mais crítico etc. Em outras palavras, foram selecionados as oportunidades que apresentavam limitações mais críticas comparadas às outras encontradas.

---

[3] http://www.nunit.org/
[4] http://nant.sourceforge.net/
[5] http://nhforge.org/

As tabelas vistas a seguir utilizam algumas métricas para indicar como as refatorações aplicadas ajudaram a melhorar alguns atributos de qualidade do sistema de software, como por exemplo, a complexidade ciclomática (CC) (MCCABE, 1976) e o índice de manutenção (IM) (ISO/IEC 9126, 1999-2002). Para o índice CC, valores mais baixos são melhores e indicam menos complexidade, enquanto para o índice IM valores mais altos são melhores. Este conjunto de métricas ajuda a medir a eficácia e precisão da abordagem proposta.

Como motivação, consideramos o uso dessas duas métricas (CC e IM), pois elas podem ajudar a medir duas limitações comumente encontradas em sistemas de software, que dizem respeito a complexidade condicional e problemas relacionados a acoplamento e duplicação de código. Além disso, são apresentadas as linhas no código fonte onde a ferramenta encontrou a oportunidade, bem como o tamanho total da classe em número de linhas de código fonte (LOCC). Podemos dizer então, que a aplicação da refatoração sugerida nos locais indicados pela ferramenta, tem por objetivo minimizar a complexidade ciclomática, maximizar o índice de manutenção, e reduzir o tamanho da classe / método em número de linhas de código quando houver duplicação. Neste caso, as métricas IM, CC e LOCC são usadas como exemplo para medir a precisão da aplicação das refatorações sugeridas.

Na maior parte dos casos, quando aplicável, na célula de cada métrica, são apresentados dois valores: o primeiro representa o valor do índice atual da classe (ainda com a limitação), o segundo após a seta "→" indica o valor para o mesmo índice calculado após a aplicação da refatoração sugerida.

5.1. Resultados das avaliações para o padrão Strategy

A Tabela 2 indica os resultados da avaliação para busca por oportunidades para o padrão Strategy, indicando que a função heurística interpretou uma instrução switch com uma limitação e uma possível oportunidade de aplicar o padrão.

Tabela 2 – Resultados da avaliação das buscas pelo padrão Strategy

Classe	Linhas	Tamanho do bloco switch	Número de Testes Condicionais	CC	IM
NUnit.Core.Builders.ProviderReference	71-92	20	3	17 → 11	67 → 73

NUnit.ConsoleRunner.EventCollector	79-121	42	3	37→28	69 → 73
NUnit.Core.Builders.DatapointProvider	77-98	21	3	432 → 311	57 → 64
NAnt.Core.ExpressionEvalBase	183-422	239	6	190 → 140	66 → 72
NAnt.Core.Tasks.LoopTask	272-393	121	4	69 → 43	61 → 63
NAnt.VSNet.VcProjectConfiguration	255-285	30	5	111 → 86	64 → 67
NHibernate.Linq.Visitors.HqlGeneratorExpressionTreeVisitor	44-141	97	19	119 → 78	65 → 73
NHibernate.Linq.Visitors.HqlGeneratorExpressionTreeVisitor	212-269	57	16	119 → 83	65 → 71

Os resultados da avaliação são calculados com base no numéro de expressões *case* encontradas e no tamanho total do bloco *switch*, que combinados fornecem o índice de complexidade do bloco, usado na avaliação pela função heurística. Para este experimento, são apresentados oito resultados de um total de 23 oportunidades encontradas. Estes oito resultados representam as classes que possuem o maior quantidade de blocos *switch* combinados com os blocos *case*.

É possível observar na tabela, que para cada índice (CC e IM), são apresentados dois valores, separados por uma seta. O primeiro indica o valor atual no sistema de software para esta métrica, enquanto o segundo indica o valor obtido após ter sido aplicada a respectiva refatoração. Nota-se com isso, uma melhoria nos atributos de qualidade, indicando que a ferramenta encontrou oportunidades para aplicar os padrões propostos. Nos exemplos da tabela, o índice de complexidade ciclomática foi minimizado e o índice de manutenção foi

maximizado, pois as instruções condicionais foram substituídas por classes com métodos virtuais, dispensando o teste estático, sendo este resolvido dinamicamente por polimorfismo.

Das várias declarações switch encontradas pela ferramenta, a abordagem indicou os possíveis candidatos para se aplicar o padrão Strategy, que são mostrados na Tabela 2. Além disso, é possível observar pela ferramenta algumas instruções switch avaliadas, onde um Strategy não seria bem aplicado, criando-se uma complexidade desnecessária. Por exemplo, não faz sentido criar dezenas de classes para eliminar uma instrução switch que apresenta algumas linhas de código em cada teste.

Nos principais casos indicados pela ferramenta, foi aplicada a refatoração *Substituir Lógica Condicional por Strategy* no código fonte. Depois de executar os testes de unidade fornecidos pelas ferramentas, temos certo grau de confiança de que o comportamento observável externo não foi afetado e todos os testes foram executados com sucesso.

Além disso, observou-se que após a aplicação da refatoração, o índice de complexidade ciclomática das classes foi reduzido consideravelmente, o que mostra que a abordagem identificou corretamente e com precisão os lugares no código que tinha lógica condicional complexa passível de ser substituído por um padrão Strategy. Consequentemente, o índice de manutenção para cada classe teve seu valor aumentado, o que também indica que a nossa abordagem melhora a forma como os sistemas de software podem ser evoluídos, eliminando declarações condicionais desnecessárias que podem ser substituídas pelo padrão Strategy.

Concluimos com estes resultados que vários fragmentos de código usam um código de tipo (*type code*) na lógica condicional para avaliar uma possibilidade de execução por algum código/algoritmo, e que seria mais apropriado encapsular este código em uma classe específica para cada tipo testado, de forma que cada teste possa então ser transferido para um método polimórfico sobrescrito nessas subclasses mediante a aplicação do padrão Strategy. Finalmente, como detectado pela função como uma interpretação negativa (não sugere a aplicação do padrão), vemos que algumas instruções switch têm muitos testes condicionais, mas as mesmas contêm pequenos blocos de código associados, tal como uma única chamada de método ou uma simples declaração de variável. Ou seja, a estrutura lógica é muito simples para ser substituída pelo padrão Strategy. Nesses casos, a criação de uma classe para cada teste condicional geraria uma arquitetura complexa desnecessária, que seria difícil de manter.

Por exemplo, examinando o processo para avaliar a primeira classe mostrada na Tabela 2 utilizando a abordagem proposta, a classe *ProviderReference*, conforme indicado

pela ferramenta, tem uma limitação com uma oportunidade para aplicar o padrão Strategy entre as linhas 71 e 92, como pode ser visto na Figura 30 (código original extraído da ferramenta NUnit). Essa classe tem o valor 17 para a complexidade ciclomática e 67 para o índice de manutenção. Depois de aplicar a refatoração *Substituir Lógica Condicional por Strategy* no local indicado pela ferramenta, o índice de CC foi reduzido para 11, enquanto que o índice IM foi aumentado para 73, o que indica que a ferramenta detectou com sucesso oportunidades para aplicar o padrão Strategy, melhorando dois atributos importantes de um sistema de software: manutenção e complexidade.

```
private object GetProviderObjectFromMember(MemberInfo member)
{
  object providerObject = null;
  object instance = null;
  switch (member.MemberType)
  {
    case MemberTypes.Property:
      PropertyInfo providerProperty = member as PropertyInfo;
      MethodInfo getMethod = providerProperty.GetGetMethod(true);
      if (!getMethod.IsStatic)
        instance = Reflect.Construct(providerType, providerArgs);
      providerObject = providerProperty.GetValue(instance, null);
      break;
    case MemberTypes.Method:
      MethodInfo providerMethod = member as MethodInfo;
      if (!providerMethod.IsStatic)
        instance = Reflect.Construct(providerType, providerArgs)
        providerObject = providerMethod.Invoke(instance, null);
      break;
    case MemberTypes.Field:
      FieldInfo providerField = member as FieldInfo;
      if (!providerField.IsStatic)
        instance = Reflect.Construct(providerType, providerArgs);
      providerObject = providerField.GetValue(instance);
      break;
  }
  return providerObject;
}
```

Figura 30 – Sentença Switch no NUnit, uma oportunidade para aplicar um padrão Strategy

A Figura 31 mostra o código refatorado para este exemplo. A Figura 32 mostra o diagrama de classes UML para o exemplo refatorado com o padrão Strategy aplicado para eliminar o switch. Basicamente, cada *case* usado na instrução switch foi transformado em uma classe concreta de *Strategy*. De modo semelhante, o IM para a classe *EventCollector* (segunda classe da tabela) foi aumentado de 69 para 73, enquanto que o índice CC diminuiu de 37 para 28.

```
private object GetProviderObjectFromMember(MemberInfo member)
{
  object providerObject = null;
  object instance = null;
  var ctx = new Context(Factory.GetInstance(member.MemberType));
  ctx.AlgorithmInterface(member, ref providerObject, ref instance);
```

```
  return providerObject;
}
...
// implementação de context de acordo com o padrão Strategy
public class Context...
public abstract class Strategy
{
  public abstract void AlgorithmInterface(
        MemberInfo member, ref object providerObject, ref object instance);
}
public class StrategyProperty : Strategy
{
  public override void AlgorithmInterface(...)
  {
     // Este algoritmo estava antes em um case, agora é um método polimórfico
    PropertyInfo providerProperty = member as PropertyInfo;
    MethodInfo getMethod = providerProperty.GetGetMethod(true);
    if (!getMethod.IsStatic)
      instance = Reflect.Construct(providerType, providerArgs);
    providerObject = providerProperty.GetValue(instance, null);
  }
}
public class StrategyMethod : Strategy
{
  public override void AlgorithmInterface(...)
  {
    // Este algoritmo estava antes em um case, agora é um método polimórfico
    MethodInfo providerMethod = member as MethodInfo;
    if (!providerMethod.IsStatic)
      instance = Reflect.Construct(providerType, providerArgs);
    providerObject = providerMethod.Invoke(instance, null);
  }
}
public class StrategyField : Strategy
{
  public override void AlgorithmInterface(...)
  {
     // Este algoritmo estava antes em um case, agora é um método polimórfico
     FieldInfo providerField = member as FieldInfo;
     if (!providerField.IsStatic)
       instance = Reflect.Construct(providerType, providerArgs);
     providerObject = providerField.GetValue(instance);
  }
}
```

Figura 31 – Código após a refatoração Substituir Lógica Condicional por Strategy

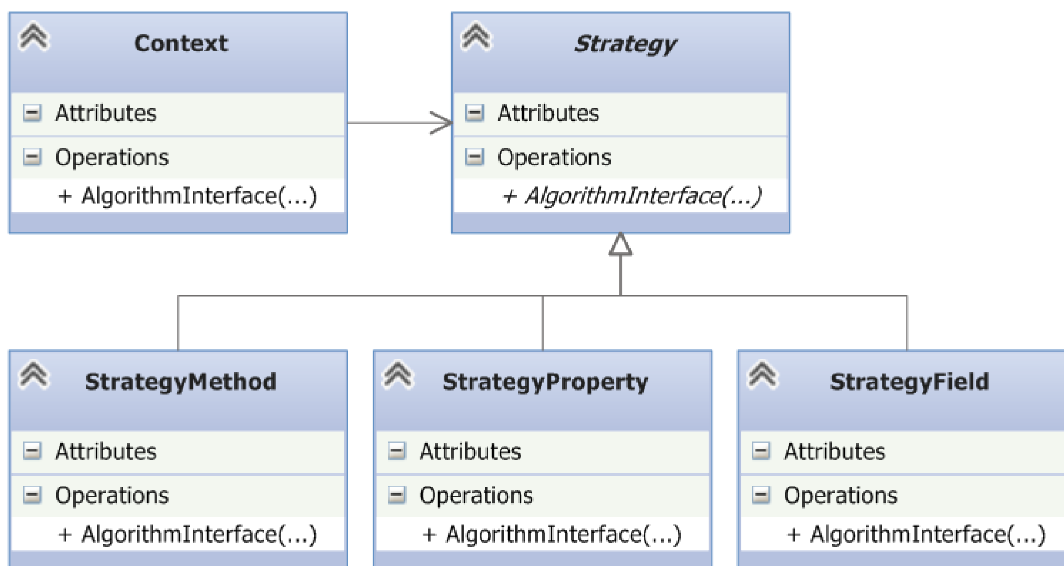


Figura 32 – Diagrama de classes UML após aplicação da refatoração para o padrão Strategy

A Figura 33 mostra um exemplo de onde a função heurística retornou uma interpretação negativa para se aplicar o padrão Strategy, que consiste de um um conjunto de pequenos testes que não representam uma oportunidade vantajosa para refatorar.

```
Switch (platformName.ToUpper())
{
  case "WIN":
  case "WIN32":
    isSupported = os.IsWindows;
    break;
  case "WIN32S":
    isSupported = os.IsWin32S;
    break;
  case "WIN32WINDOWS":
    isSupported = os.IsWin32Windows;
    break;
  case "WIN32NT":
    isSupported = os.IsWin32NT;
  break;
  case "WINCE":
    isSupported = os.IsWinCE;
    break;
  case "WIN95":
    isSupported = os.IsWin95;
    break;
  case "WIN98":
    isSupported = os.IsWin98;
    break;
  case "WINME":
    isSupported = os.IsWinME;
    break;
  case "NT3":
    isSupported = os.IsNT3;
    break;
  case "NT4":
    isSupported = os.IsNT4;
    break;
  case "NT5":
    isSupported = os.IsNT5;
    break;
  case "WIN2K":
    isSupported = os.IsWin2K;
    break;
  case "WINXP":
    isSupported = os.IsWinXP;
    break;
  case "WIN2003SERVER":
    isSupported = os.IsWin2003Server;
    break;
  case "NT6":
    isSupported = os.IsNT6;
    break;
  case "VISTA":
    isSupported = os.IsVista;
    break;
  case "WIN2008SERVER":
    isSupported = os.IsWin2008Server;
    break;
  case "WIN2008SERVERR2":
    isSupported = os.IsWin2008ServerR2;
    break;
  case "WIN2012SERVER":
    isSupported = os.IsWin2012Server;
    break;
  case "WINDOWS7":
    isSupported = os.IsWindows7;
    break;
  case "WINDOWS8":
    isSupported = os.IsWindows8;
```

```
    break;
  case "UNIX":
  case "LINUX":
    isSupported = os.IsUnix;
    break;
  #if (CLR_2_0 || CLR_4_0) && !NETCF
  case "XBOX":
    isSupported = os.IsXbox;
    break;
  case "MACOSX":
    isSupported = os.IsMacOSX;
    break;
  #endif
  default:
    isSupported = IsRuntimeSupported(platformName);
    break;
}
```

Figura 33 – Sentença Switch no NUnit, porém não é uma oportunidade vantajosa para aplicar o padrão Strategy

Uma desvantagem da aplicação do padrão Strategy para eliminar sentenças condicionais é que o projeto conterá novas classes para comportar estes algoritmos, o que de certa forma adiciona complexidade ao projeto. Mesmo reduzindo os índices de CC e IM da classe principal que contém o teste, novas classes serão criadas, cada uma com seu próprio IM. Isto também cria um acoplamento, mesmo que baixo, entre as classes envolvidas no padrão, já que se comunicam por meio de uma abstração, o *Strategy*. No entanto, a sentença condicional mesmo assim pode ser eliminada por meio do uso de polimorfismo, melhorando índices de CC.

5.2. Resultados das avaliações para o padrão Factory Method

A Tabela 3 apresenta as principais classes avaliadas pela ferramenta para o padrão Factory Method. Para este experimento, são apresentados quatro resultados de um total de oito oportunidades encontradas, que apresentam a maior quantidade de código duplicado na criação de objetos de uma mesma classe base.

Tabela 3 – Resultados da avaliação das buscas pelo padrão Factory Method

Classe	Linhas	LOCC	CC	IM
NAnt.VSNet.VcProject	456-474	837 → 834	340 → 336	49 → 53
NUnit.Util.InProcessTestRunnerFactory	34-44	10 → 7	6 → 3	78 → 85
NHibernate.Hql.Ast.ANTLR.Tree.HqlSqlWalkerTreeAdaptor	25–160	93 → 25	54 → 9	53 → 74
NHibernate.Hql.Ast.HqlTreeBuilder	155-176	182 → 175	96 → 87	81 → 82

Observamos que após a aplicação da refatoração, o índice de complexidade ciclomática das classes foi reduzido, o que mostra que nossa abordagem identificou como

lógica condicional complexa que decide pela criação de objetos de uma mesma subclasse, sendo assim uma oportunidade para aplicar o Factory Method. Consequentemente, o índice de manutenção para cada classe teve seu valor aumentado. Finalmente, em certos casos, o acoplamento de classe foi minimizado, o que indica que a ferramenta pode também ajudar a reduzir os problemas relacionados com a dependência devido à criação direta de objetos com o operador *new*, sem a utilização de uma fábrica.

Por exemplo, vamos examinar o processo para avaliar a primeira classe na Tabela 3, utilizando a nossa abordagem. A classe *VcProject*, de acordo com os dados indicados pela ferramenta, tem uma limitação com uma oportunidade para aplicar o padrão Factory Method entre as linhas 456 e 474, como podemos ver na Figura 34 (código original retirado da ferramenta NAnt). A classe *VcProjectClass* tem 837 linhas de código (LOCC), 340 para o índice de CC e 49 para o IM. Após a aplicação da refatoração *Introduzir Criação Polimórfica com Factory Method* no local indicado pela ferramenta, os valores das métricas citadas tiveram seus valores melhorados. Tais valores indicam que a ferramenta detectou com sucesso a oportunidade para aplicar o padrão de projeto Factory Method.

```
public class VcProject: ProjectBase
{
...
protected virtual ReferenceBase CreateReference(SolutionBase solution, XmlElement xmlDefinition)
{
  ...
  // switch usando um type code para decidir que tipo de objeto criar
  switch (xmlDefinition.Name)
  {
    case "ProjectReference":
      // project reference
      return new VcProjectReference(xmlDefinition, ReferencesResolver,
          this, solution, solution.TemporaryFiles, GacCache,
          OutputDir);
    case "AssemblyReference":
      // assembly reference
      return new VcAssemblyReference(xmlDefinition, ReferencesResolver,
          this, GacCache);
    case "ActiveXReference":
      // ActiveX reference
      return new VcWrapperReference(xmlDefinition, ReferencesResolver,
          this, GacCache);
    default:
      throw new BuildException(string.Format(CultureInfo.InvariantCulture,
          "\"{0}\" reference not supported.", xmlDefinition.Name),
          Location.UnknownLocation);
  }
}
```

Figura 34 –Switch no NAnt criando subclasses, oportunidade para aplicar o padrão Factory Method

A Figura 35 mostra o código refatorado para este exemplo. Na instrução switch do método *CreateReference*, os códigos incluídos em cada teste condicional responsável por verficar o valor de uma string e decidir então por qual tipos de objeto criar, como por exemplo

VcProjectReference, *VcAssemblyReference* e *VcWrapperReference*, foram transferidos para uma família de classes. Foi criada uma nova classe *FactoryBase* com um método chamado *createInstance* abstrato que retorna um tipo *ReferenceBase* como resultado. Esta é uma classe base para classes *VcProjectReference*, *VcAssemblyReference* e *VcWrapperReference*, ou seja, em suma estamos aplicando o padrão Factory Method. *FactoryBase* tem três subclasses que podem então sobrescrever o método polimórfico da classe base (o método de fábrica, também chamado de construtor virtual) para fornecer objetos mais específicos como resultado. As declarações condicionais podem então ser removidas completamente, podendo agora usar uma chamada polimórfica para solicitar produtos concretos da *FactoryBase*.

```
public class VcProject: ProjectBase
{
...
protected virtual ReferenceBase CreateReference(
  SolutionBase solution, XmlElement xmlDefinition)
{
  ...
  FactoryBase factory =
    FactoryCreator.getFactory(xmlDefinition.Name);
  // Retorna um produto concreto usando um método polimórfico
  return factory.createInstance(...);
}
...
// Implementação do Factory Method
public abstract class FactoryBase
{
  public abstract ReferenceBase createInstance(...);
}

public class FactoryVcProjectReference: FactoryBase
{
  public override ReferenceBase createInstance(...)
  {
    return new VcProjectReference(...);
  }
}

public class FactoryVcAssemblyReference : FactoryBase
{
  public override ReferenceBase createInstance(...)
  {
    return new VcAssemblyReference(...);
  }
}

public class FactoryVcWrapperReference : FactoryBase
{
  public override ReferenceBase createInstance(...)
  {
    return new VcWrapperReference(...);
  }
}
```

Figura 35 – Código refatorado após a aplicação do padrão Factory Method

A Figura 36 mostra o diagrama de classes UML para este exemplo refatorado, com o padrão Factory Method aplicado.

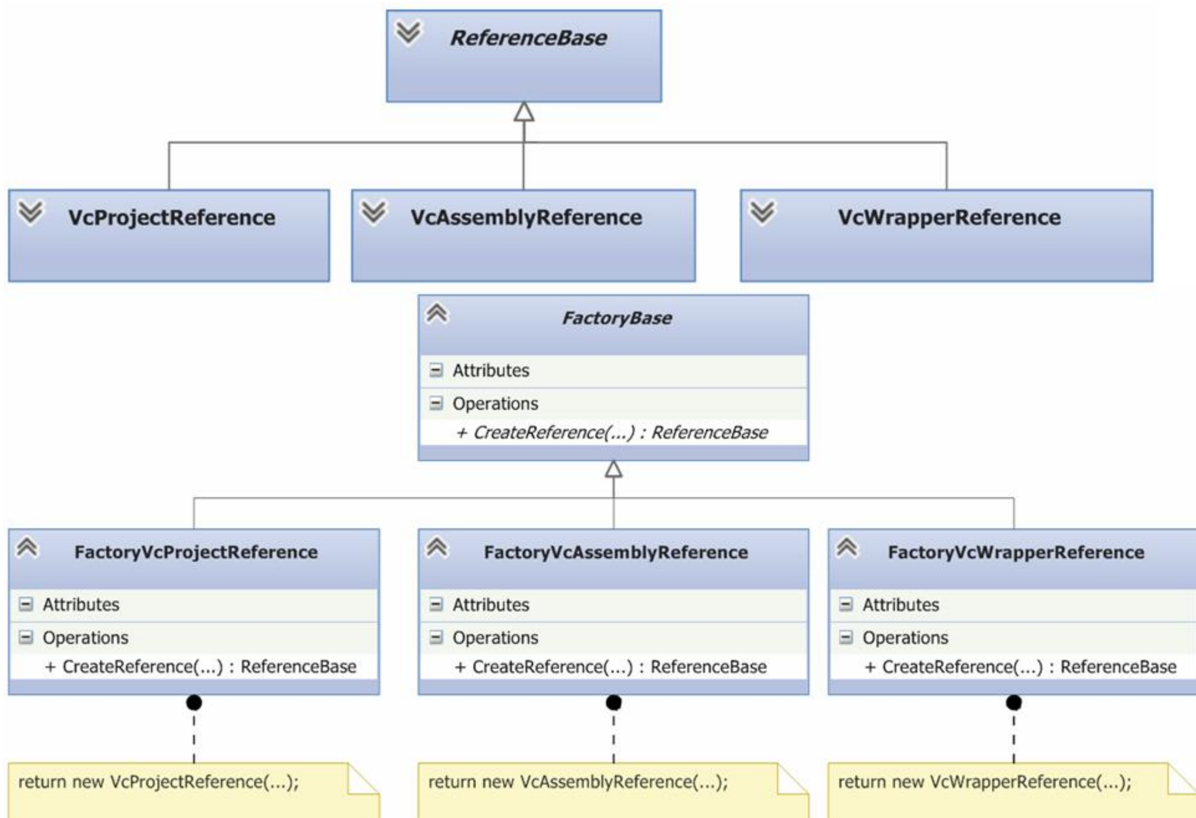


Figura 36 – Diagrama de classes após refatoração para o padrão Factory Method

Cada subclasse agora fica responsável por retornar a instância de um tipo concreto com base em uma abstração previamente definida.

5.3. Resultados das avaliações para o padrão Template Method

A Tabela 4 apresenta as principais classes avaliadas pela ferramenta com oportunidades para aplicar o padrão Template Method. Para este experimento, são apresentados oito resultados de um total de 22 oportunidades encontradas. Estes oito resultados representam as classes que continham a maior quantidade de código duplicado em número de linhas de código.

Tabela 4 – Resultados da avaliação das buscas pelo padrão Template Method

Subclasse	Superclasse	Método	Linhas	LOCC
NAnt.VSNet.ManagedWrapperReference	*NAnt.VSNet.WrapperReferenceBase*	*ImportActiveXLibrary*	305-355	85 → 68
NAnt.VSNet.VcWrapperReference	*NAnt.VSNet.WrapperReferenceBase*	*ImportActiveXLibrary*	242-284	65 → 41

NUnit.Framework.Constraints.AndConstraint	*NUnit.Framework.Constraints.Constraint*	*WriteDescriptionTo*	78-87	15 → 13
NUnit.Framework.Constraints.OrConstraint	*NUnit.Framework.Constraints.Constraint*	*WriteDescriptionTo*	137-146	7 → 5
NUnit.Framework.Constraints.BinarySerializableConstraint	*NUnit.Framework.Constraints.Constraint*	*Matches*	28-56	16 → 7
NUnit.Framework.Constraints.XmlSerializableConstraint	*NUnit.Framework.Constraints.Constraint*	*Matches*	102-136	19 → 7
NAnt.DotNet.Tasks.IlasmTask	*NAnt.Core.Tasks.ExternalProgramBase*	*NeedsCompiling*	536-592	98 → 78
NAnt.DotNet.Tasks.CompilerBase	*NAnt.Core.Tasks.ExternalProgramBase*	*NeedsCompiling*	1167-1277	433 → 414

Como resultado final, após a aplicação da refatoração *Formar Template Method* nas linhas indicadas pela ferramenta, a métrica LOCC para cada classe teve seu valor reduzido, o que indica que a abordagem indicou com sucesso locais com duplicação de código que podem ser removidos para uma classe base e compartilhados por subclasses. Isso pode ser feito por meio da eliminação do código repetido desnecessário que pode então ser eliminado pela aplicação do padrão Template Method.

Por exemplo, examinando o processo para avaliar as duas primeiras classes da Tabela 4, vemos que as subclasses *ManagedWrapperReference* e *VcWrapperReference*, ambas descendentes da mesma superclasse *WrapperReferenceBase*, conforme indicado pela ferramenta, têm a limitação de código duplicado e uma oportunidade para aplicar o padrão Template Method. Essa limitação está localizada entre as linhas 305 e 355 para a classe *ManagedWrapperReference* e entre as linhas 242 e 284 para a classe *VcWrapperReference*, como pode ser visto na Figura 37 (código original da ferramenta NAnt) e na Figura 38 (diagrama de classes UML).

```
// Subclasse
public class ManagedWrapperReference : WrapperReferenceBase {
...
protected override void ImportActiveXLibrary() {
  // Começo do código duplicado
  AxImpTask axImp = new AxImpTask();
  axImp.Parent = SolutionTask;
  axImp.Project = SolutionTask.Project;
  axImp.NamespaceManager = SolutionTask.NamespaceManager;
  axImp.Verbose = SolutionTask.Verbose;
  axImp.InitializeTaskConfiguration();
  axImp.OcxFile = new FileInfo(GetTypeLibrary());
```

```
  axImp.OutputFile = new FileInfo(WrapperAssembly);
  // Fim do código duplicado
  if (ProjectSettings.AssemblyOriginatorKeyFile != null) {
    axImp.KeyFile = new FileInfo(
      FileUtils.CombinePaths(
        Parent.ProjectDirectory.FullName,
        ProjectSettings.AssemblyOriginatorKeyFile)
    );
  }
  if (ProjectSettings.AssemblyKeyContainerName != null) {
    axImp.KeyContainer =
      ProjectSettings.AssemblyKeyContainerName;
  }
  // Mais código duplicado
  string rcw = PrimaryInteropAssembly;
  if (rcw == null) {
    rcw = FileUtils.CombinePaths(
      Parent.ObjectDir.FullName,
      "Interop." + TypeLibraryName + ".dll");
  }
  if (File.Exists(rcw)) {
    axImp.RcwFile = new FileInfo(rcw);
  }
  axImp.Project.Indent();
  try
  {
    axImp.Execute();
  }
  finally
  {
    axImp.Project.Unindent();
  }
}
// Subclasse
public class VcWrapperReference : WrapperReferenceBase
{
...
protected override void ImportActiveXLibrary()
{
  // Começo do código duplicado
  AxImpTask axImp = new AxImpTask();
  axImp.Parent = SolutionTask;
  axImp.Project = SolutionTask.Project;
  axImp.NamespaceManager = SolutionTask.NamespaceManager;
  axImp.Verbose = SolutionTask.Verbose;
  axImp.InitializeTaskConfiguration();
  axImp.OcxFile = new FileInfo(GetTypeLibrary());
  axImp.OutputFile = new FileInfo(WrapperAssembly);
  string rcw = PrimaryInteropAssembly;
  if (rcw == null)
  {
    rcw = FileUtils.CombinePaths(
     Parent.ObjectDir.FullName,
     "Interop." + TypeLibraryName + ".dll");
  }
  if (File.Exists(rcw))
  {
    axImp.RcwFile = new FileInfo(rcw);
  }
  axImp.Project.Indent();
  try
  {
    axImp.Execute();
  }
  finally
  {
    axImp.Project.Unindent();
  }
  // Fim do código duplicado
}
```

Figura 37 – Código duplicado no NAnt, uma oportunidade para aplicar o padrão Template Method

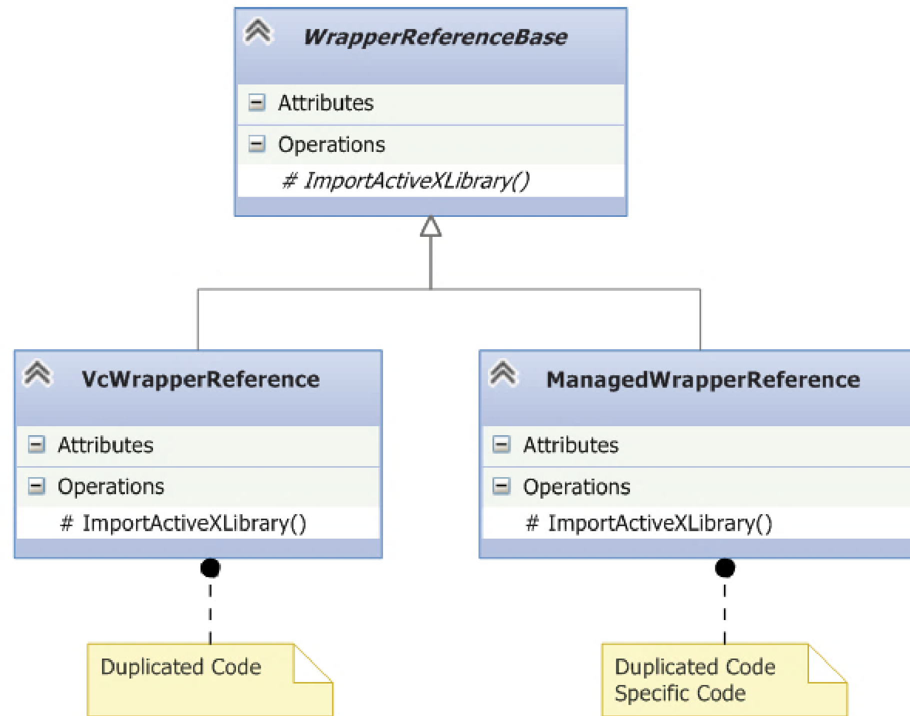


Figura 38 – Diagrama de classes para o estudo de caso no NAnt, com código duplicado em subclasses

Como podemos ver, ambas as classes têm um trecho considerável de código duplicado. A classe *ManagedWrapperReference* tem 85 linhas de código (LOCC), enquanto a classe *VcWrapperReference* tem 65 linhas de código. Após a aplicação da refatoração *Formar Template Method* no local indicado pela ferramenta em ambas as subclasses, a métrica LOCC foi reduzida para 68 para o *ManagedWrapperReference* e 41 para a classe *VcWrapperReference*, o que indica que a ferramenta detectou com sucesso a oportunidade de aplicar o padrão Template Method, e reduziu consideravelmente o tamanho das classes em linhas de código.

A Figura 39 mostra o código refatorado para este exemplo, enquanto a Figura 40 mostra o diagrama final classe UML. A sequência similar de passos encontradas no método *ImportActiveXLibrary* nas subclasses *ManagedWrapperReference* e *VcWrapperReference* foi transferida para um novo método na classe base *WrapperReferenceBase*. Este é o Template Method, chamado aqui de *ImportActiveXLibraryTemplateMethod*, que invoca a sequência antiga de passos duplicados encontrados em ambas subclasses. Todas as etapas distintas podem permanecer nas subclasses, sendo então invocadas por polimorfismo no template method.

```
// Classe base
public abstract class WrapperReferenceBase : FileReferenceBase
{
...
// Método polimórfico
protected abstract void ImportActiveXLibrary(AxImpTask axImp);
// Template Method
protected void ImportActiveXLibraryTemplateMethod()
{
  // Passos similares das subclasses foram movidos para cá
  AxImpTask axImp = new AxImpTask();
  axImp.Parent = SolutionTask;
  axImp.Project = SolutionTask.Project;
  axImp.NamespaceManager = SolutionTask.NamespaceManager;
  axImp.Verbose = SolutionTask.Verbose;
  axImp.InitializeTaskConfiguration();
  axImp.OcxFile = new FileInfo(GetTypeLibrary());
  axImp.OutputFile = new FileInfo(WrapperAssembly);
  // Agora chama um método polimórfico para performar passos específicos
  ImportActiveXLibrary(axImp);
  // Mais passos similares movidos das para cá
  string rcw = PrimaryInteropAssembly;
  if (rcw == null)
  {
    rcw = FileUtils.CombinePaths(
      Parent.ObjectDir.FullName,
      "Interop." + TypeLibraryName + ".dll");
  }
  if (File.Exists(rcw))
  {
    axImp.RcwFile = new FileInfo(rcw);
  }
  axImp.Project.Indent();
  try
  {
    axImp.Execute();
  }
  finally
  {
    axImp.Project.Unindent();
  }
}
// Subclasse, refatorada
...
public class ManagedWrapperReference : WrapperReferenceBase {
...
protected override void ImportActiveXLibrary(AxImpTask axImp)
 {
   if (ProjectSettings.AssemblyOriginatorKeyFile != null)
   {
     axImp.KeyFile = new FileInfo(
       FileUtils.CombinePaths(
         Parent.ProjectDirectory.FullName, ProjectSettings.AssemblyOriginatorKeyFile)
      );
   }
   if (ProjectSettings.AssemblyKeyContainerName != null)
   {
     axImp.KeyContainer = ProjectSettings.AssemblyKeyContainerName;
   }
 }
// Subclasse, refatorada
public class VcWrapperReference : WrapperReferenceBase {
...
protected override void ImportActiveXLibrary(AxImpTask axImp)
{
  // este método ficou sem implementação
  // todos os passos estavam duplicados
  // e agora usam o código compartilhado pelo Template Method
}
```

Figura 39 – Código refatorado após a aplicação do padrão Template Method

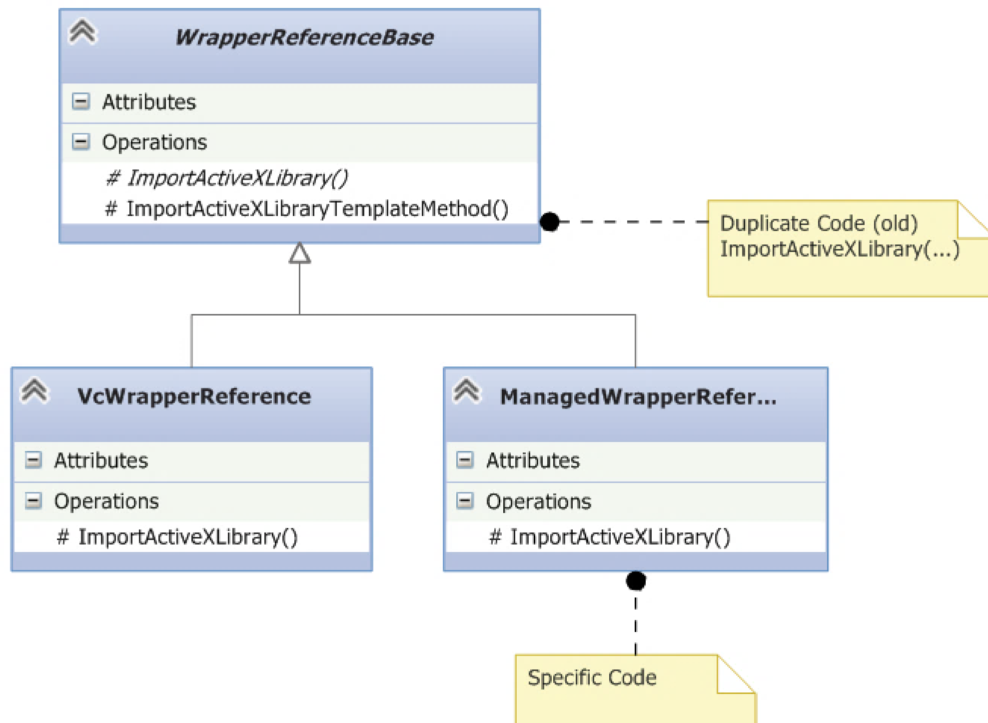


Figura 40 – Diagrama de classes para do exemplo no NAnt, com aplicação do padrão Template Method, reduzindo a duplicação de código

O código do método *ImportActiveXLibrary* da subclasse *VcWrapperReference* ficou praticamente sem código algum em sua implementação, restando somente o cabeçalho, pois estava duplicado em sua totalidade. É necessário manter a assinatura do método com *override* na subclasse porque a linguagem C# obriga que métodos declarados com *abstract* na classe base sejam sobrescritos em classes descendentes.

5.4. Resultados das avaliações para o padrão Creation Method

A Tabela 5 apresenta as principais classes avaliadas pela ferramenta como oportunidades para aplicar o Creation Method. Além de apresentar o local (linhas) onde a refatoração é sugerida, apresenta também o número total de parâmetros distintos que foram combinados na declaração de cada construtor, usando para isso a combinação do nome e tipo. Para este experimento, são apresentados oito resultados de um total de 25 oportunidades encontradas. Estes oito resultados representam a classes com maior número de construtores e parâmetros variados.

Tabela 5 – Resultados da avaliação das buscas pelo padrão Creation Method

Classe	Linhas	Número de Construtores	Parâmetros Distintos
NAnt.Core.BuildEventArgs	141, 148, 157, 167	4	3
NAnt.Core.DataTypeBaseDictionary	33, 40, 44, 48, 56, 60, 64, 68, 72, 76, 80	11	6
NAnt.Core.ExpressionParseException	40, 41, 42, 43, 48, 53, 58	7	7
NAnt.Core.Project	165, 181, 204, 227, 258, 297	6	6
NHibernate.Engine.QueryParameters	24, 26, 34, 37, 40, 45, 50, 60, 78	9	18
NHibernate.Test.LogSpy	17, 34, 39, 40, 41, 43, 44	7	5
NHibernate.Impl.SqlQueryImpl	32, 57, 65, 80, 83	5	11
NHibernate.Mapping.ByCode. ModelMapper	18, 20, 22, 25, 28	5	4

Como resultado final, após a aplicação da refatoração *Substituir Construtores por Métodos de Construção* nas linhas indicadas pela ferramenta, cada "construtor" pode então expressar melhor o que realmente está criando de acordo com os parâmetros passados. Em algumas classes refatoradas foram encontrados construtores não utilizados, como sugere um dos benefícios dessa refatoração, o que ajuda a reduzir também o tamanho das classes em número de linhas de código.

Por exemplo, examinando o processo para avaliar a primeira classe da Tabela 5, vemos que a classe *BuildEventArgs*, conforme indicado pela ferramenta, tem várias versões sobrecarregadas de seu construtor, que leva o mesmo nome da classe, não expressando sua intenção que permita ao desenvolvedor escolher o construtor que mais se adequa a sua necessidade. A classe tem quatro construtores de mesmo nome, com três parâmetros de tipos diferentes como indicado na tabela, sendo um default (sem parâmetros) e outros três que

recebem como parâmetro um dos seguintes tipos, respectivamente: *Project*, *Target* ou *Task*. Esta é uma oportunidade para aplicar o padrão Creation Method. Como indicado pela ferramenta, os construtores estão localizados nas linhas 141, 148, 157 e 167.

A Figura 41 mostra o código original com a limitação presente na ferramenta NAnt, enquanto a Figura 42 mostra o código refatorado, com os construtores renomeados expressando a sua real intenção. Cada parâmetro pode ser utilizado para nomear o método de acordo com o seu nome e / ou tipo de dados.

```
public class BuildEventArgs : EventArgs
{
  public BuildEventArgs()
  {
  }

  public BuildEventArgs(Project project)
  {
    _project = project;
  }

  public BuildEventArgs(Target target)
  {
    _project = target.Project;
    _target = target;
  }

  public BuildEventArgs(Task task)
  {
    _project = task.Project;
    _target = task.Parent as Target;
    _task = task;
  }
...
```

Figura 41 – Sobrecargas de construtores no NAnt, uma oportunidade para aplicar o padrão Creation Method

```
public class BuildEventArgs : EventArgs
{
  private BuildEventArgs(Project project, Target target, Task task)
  {
    this._project = project;
    this._target = target;
    this._task = task;
  }

  public BuildEventArgs newWithProject(Project project)
  {
    return new BuildEventArgs(project, null, null);
  }

  public BuildEventArgs newWithTarget(Target target)
  {
    return new BuildEventArgs(target.project, target, null);
  }

  public BuildEventArgs newWithTask(Task task)
  {
    return new BuildEventArgs(task.project, task.Parent as Target, task);
  }
...
```

Figura 42 – Código refatorado após a aplicação do padrão Creation Method: métodos expressando melhor a intenção

Com estas avaliações, concluímos que existem várias oportunidades para aplicar o padrão Creation Method de forma a tornar mais clara a intenção do que está sendo inicializado em cada sobrecarga.

5.5. Resultados das avaliações para o padrão Chain Constructors

A Tabela 6 apresenta as principais classes avaliadas pela ferramenta. Para cada classe, a tabela apresenta o número de construtores encontrados e o número total de linhas de código (LOCC) antes e após a a aplicação da refatoração *Encadear Construtores*. Para este experimento, são apresentados sete resultados de um total de 179 oportunidades encontradas. Estes sete resultados representam os construtores com maior número de código duplicado na inicialização.

Tabela 6 – Resultados da avaliação das buscas pelo padrão Chain Constructors

Classe	Linhas	Número de Construtores	LOCC	IM
NHibernate.DomainModel.FooComponent	41, 50, 56, 64	4	55 → 47	82 → 88
NHibernate.Util.StringTokenizer	20, 32, 45	3	16 → 10	79 → 83
NHibernate.Engine.CollectionEntry	116, 129, 139, 157,	4	84 → 76	81 → 84
NHibernate.Hql.Ast.ANTLR.Tree.ASTNode	20, 23, 31	3	198 → 194	76 → 79
NAnt.VSNet.VcFileConfiguration	36, 61	2	61 → 58	64 → 67
PNUnit.Framework. PNUnitTestInfo	58, 72	2	17 → 12	70 → 77
NUnit.UiException.ErrorItem	51, 67	2	63 → 61	75 → 77

Após a aplicação da refatoração *Encadear Construtores* nas linhas indicadas pela ferramenta, os construtores com código duplicado tiveram seu tamanho reduzido. Em algumas classes refatoradas, foram encontrados constutores não utilizados, como sugere um dos benefícios dessa refatoração. Além disso, a função heurística foi capaz de detectar não somente a duplicação de inicialização de atributos, mas outras expressões idênticas, como

invocação de métodos da mesma classe. Nesse caso, aplicando-se a refatoração *Extrair Método* (FOWLER, 1999) pode-se também reduzir código duplicado. Além disso, eliminando código duplicado, o índice IM das classes teve seu valor melhorado.

Por exemplo, examinando o processo para avaliar a primeira classe da Tabela 6, vemos que a classe *FooComponent*, conforme indicado pela ferramenta, tem várias versões sobrecarregadas de seu construtor, que leva o mesmo nome da classe, tendendo a duplicar código de inicialização. A classe tem quatro construtores de mesmo nome, variando seus parâmetros em quantidade e tipos, sendo um default (sem parâmetros) e outros três que recebem como parâmetro combinações dos seguintes parâmetros: *name (string), count (int), dates(DateTime[]), subComponent(FooComponent)* e *fee(Fee)*. Todos os construtores basicamente repassam os valores recebidos por parâmetro para atributos internos privados à classe, incluindo *_name(String), _count(Int32), _importantDates(DateTime[]), _subcomponent(FooComponent)* e *_fee(Fee)*, o que gera código duplicado em todos eles. Esta é uma oportunidade para aplicar o padrão Chain Constructors. Como indicado pela ferramenta, estes construtores estão localizados nas linhas 41, 50, 56, 64.

A Figura 43 mostra o código original com a limitação presente na ferramenta NHibernate, enquanto a Figura 44 mostra o código refatorado, com os construtores encadeados e com todo código duplicado eliminado.

```
public class FooComponent
{
  // fields
  private String _name;
  private Int32 _count;
  private DateTime[] _importantDates;
  ...
  private FooComponent _subcomponent;
  private Fee _fee = new Fee();
  ...

  // construtor default
  public FooComponent()
  {
  }

  // construtor de propósito específico
  public FooComponent(String name, Int32 count)
  {
    // código duplicado
    _name = name;
    _count = count;
  }

 // construtor de propósito específico
 public FooComponent(String name, int count, DateTime[] dates, FooComponent subcomponent)
 {
    // código duplicado
    _name = name;
    _count = count;
    _importantDates = dates;
    _subcomponent = subcomponent;
```

```
  }

  // construtor de propósito geral
  public FooComponent(String name, int count, DateTime[] dates, FooComponent subcomponent, Fee fee)
  {
    _name = name;
    _count = count;
    _importantDates = dates;
    _subcomponent = subcomponent;
    _fee = fee;
  }
...
```

Figura 43 – Construtores com código duplicado no NHibernate, uma oportunidade para aplicar o padrão Chain Constructors

```
public class FooComponent
{
  // fields
  private String _name;
  private Int32 _count;
  private DateTime[] _importantDates;
  ...
  private FooComponent _subcomponent;
  private Fee _fee = new Fee();
  ...
  // construtor default
  public FooComponent()
  {
  }

  // construtor de propósito específico agora está encadeado ao geral
  public FooComponent(String name, Int32 count): this (name,count,null,null)
  {
    // código duplicado removido deste local
  }

  // construtor de propósito específico agora está encadeado ao geral
  public FooComponent(String name, int count, DateTime[] dates, FooComponent subcomponent): this
(name,count,dates,subcomponent,null)
  {
    // código duplicado removido deste local
  }

  // construtor de propósito geral
  public FooComponent(String name, int count, DateTime[] dates, FooComponent subcomponent, Fee fee)
    {
      _name = name;
      _count = count;
      _importantDates = dates;
      _subcomponent = subcomponent;
      _fee = fee;
    }
...
```

Figura 44 – Código refatorado: construtores encadeados repassando parâmetros na assinatura sem duplicar código na implementação

A redução do tamanho da classe *FooComponent* em número de linhas de código (LOCC) nesse caso foi de 55 para 47, devido à eliminação do código duplicado em dois construtores específicos, que ficaram sem implementação alguma, repassando pela própria assinatura com *this* os parâmetros recebidos para o construtor de propósito mais geral.

6. CONCLUSÃO

Essa dissertação apresentou uma abordagem para buscar oportunidades de refatoração para aplicação de padrões de projeto, mais especificamente os padrões *Strategy, Factory Method, Template Method, Creation Method* e *Chain Constructors*, usando as refatorações *Substituir Lógica Condicional por Strategy, Introduzir Criação Polimórfica com Factory Method, Formar Template Method, Substituir Construtores por Métodos de Criação* e *Encadear Construtores.*

A aplicação de padrões de projeto, dentro de um contexto evolutivo baseado em técnicas de desenvolvimento ágil como refatoração, se torna uma atividade atraente no ciclo de vida de uma aplicação, pois elimina complexidade desnecessária na concepção de um sistema de software, ao mesmo tempo em que evita a escassez de engenharia, que seria evoluir um sistema de software sem nunca aplicar um padrão. Refatoração é um aspecto chave neste processo evolutivo.

A falta de abordagens semi-automáticas que auxiliem o programador a localizar limitações no código-fonte foi o principal estímulo para o desenvolvimento deste trabalho. O apoio fornecido pela ferramenta AROS, por meio da implementação da abordagem e funções propostas, indicando *qual* padrão de projeto aplicar e o exato *local* no código fonte, se torna importante na atividade de melhorar a estrutura de projetos existentes.

As principais motivações para a escolha destes padrões são que eles resolvem algumas das principais limitações comumente encontradas em sistemas de software, tais como complexidade condicional, duplicação de código e forte acoplamento. Essas limitações tornam sistemas de software difíceis de serem mantidos e evoluídos, comprometendo ainda a reutilização. As abordagens propostas têm como objetivo ajudar na identificação destas limitações e sugerir refatorações que tenham como resultado final da transformação um código em conformidade com um padrão de projeto, que é uma solução já testada e documentada.

A arquitetura utilizada na construção da ferramenta e do framework, bem como na descrição das atividades de buscas, permite ainda a adição de novas funções para outros padrões de projeto úteis que podem ser implementados no futuro, tornando assim este projeto extensível.

Com os estudos de caso, verificamos que ferramentas de código fonte aberto como NAnt, NUnit e NHibernate, mesmo sendo projetos já maduros e evoluídos, apresentam

possíveis melhorias a serem realizadas, principalmente por meio da aplicação dos padrões de projeto sugeridos, conforme demonstram os resultados obtidos por meio das avaliações. Nota-se que para alguns destes projetos, principalmente de larga-escala, a inspeção manual do código-fonte a fim de localizar oportunidades para aplicar refatorações para um padrão de projeto seria uma tarefa inviável. O apoio de uma ferramenta semi-automática para auxiliar nessa busca é fundamental.

6.1. Trabalhos futuros

- **Estender o framework para adição de novas refatorações para outros padrões de projeto.** Alguns padrões de projeto úteis podem ser adicionados ao framework de buscas, utilizando o processo definido. Alguns destes padrões poderiam incluir o *Command*, *Abstract Factory* ou *Visitor* (GAMMA et al., 1999), também destinados a resolver problemas relacionados à lógica condicional, criação de objetos, acoplamento e reutilização.
- **Novas funcionalidades na ferramenta AROS ou integração com IDEs.** A ferramenta AROS foi desenvolvida de forma a ser executada de forma independente, sem a necessidade do uso de um IDE em específico, como o Visual Studio. Após encontrar uma oportunidade de refatoração, o programador pode utilizar as ferramentas integradas oferecidas por estes IDEs para aplicar as refatorações sugeridas, usando as respectivas mecânicas. Neste caso, por meio de uma API, a ferramenta AROS poderia mostrar uma pré-visualização da refatoração, retirando do programador a tarefa de aplicar a refatoração manualmente. Usando esta mesma API de integração, o AROS poderia apresentar automaticamente as métricas como IM e CC diretamente em sua interface. Uma limitação dessa abordagem seria a dependência da API do IDE do Visual Studio.
- **Considerar o histórico de versões de um código fonte como indício de busca.** Neste projeto, consideramos como indícios principais algumas limitações comumente encontradas, como sentenças condicionais complexas, problemas relacionados à criação de objetos e consequentemente acoplamento e código duplicado. Um bom indício que pode ser considerado em trabalhos futuros é a análise das diferentes versões de um determinado código fonte, de forma a examinar como aquele código se torna complexo à medida que evolui, justificando ou não a aplicação de um padrão de projeto. Por exemplo, um teste condicional adicionado entre uma versão e outra de um sistema de software, poderia ser mais um indício da aplicação de um padrão *Strategy, State, Factory Method* ou *Command* (GAMMA et al., 1999).

REFERÊNCIAS

BALAZINSKA, M. et al. Advanced clone-analysis to Support Object-Oriented System Refactoring. In: Working Conference on Reverse Engineering, WCRC, 7., 2000, Washington, USA. **Proceedings…** Los Alamitos: IEEE Press, 2000. P.98 – 107.

BAVOTA, G., DE LUCIA, A., OLIVETO, R. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. Journal of Systems and Software, P.397-414, 4 dez 2010.

BECK, K. **Test Driven Development**: By Example. [S.1.]: Addison-Wesley, 2002.

EL-SHARQWI, M., MAHDI, H., EL-MADAH, I. Pattern-based model refactoring. In: International Conference on Computer Engineering and Systems. Cairo, EG. **Proceedings…** IEEE, 2010. P.301-306.

FOWLER, M. **Refactoring**: Improving the design of existing code. Boston, MA, USA: Addison-Wesley, 1999.

GAMMA, E. et al. **Design Patterns**: elements of reusable object-oriented software. [S.1.: s.n.]: Addison-Wesley, 1999.

ISO/IEC 9126 – Software and System Engineering – Product quality – Part 1: Quality model. 1999-2002.

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

KIM, D. Software Quality Improvement via Pattern-Based Model Refactoring. In: High Assurance Systems Engineering Symposium. Nanjing, CN. **Proceedings…** IEEE, 2008. P.293-302.

MCCABE, T.J. A Complexity Measure. IEEE Transactions on Software Engineering, vol.SE-2, no.4, pp.308,320, Dec. 1976.

MENS, T.; TOURWÉ, T. **A survey of Software Refactoring**. Piscataway, NJ, USA: IEEE Press, 2004. 126-139p. v.30, n.2.

MUNRO, M.J. Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code, 2005. 11th IEEE International Symposium, vol., no., pp.15,15, 19-22 Sept. 2005.

NANT. **NAnt Home Page**. Disponível em: <http://nant.sourceforge.net>. 2014. Acesso em: 10 jun. 2014.

NHIBERNATE. **NHibernate Home Page**. Disponível em: <http://nhforge.org>. 2014. Acesso em: 10 jun. 2014.

NREFACTORY. **NRefactory Home Page.** Disponível em: <https://github.com/icsharpcode/NRefactory>. 2014. Acesso em: 10 jun. 2014.

NUNIT. **NUnit Home Page**. Disponível em: <http://www.nunit.org>. 2014. Acesso em: 10 jun. 2014.

O'KEEFFE, M.; CINNÉIDE, M. Ó. Search-based refactoring: an empirical study, Journal Software Maintenance and Evolution Research and Practice., 20: 345–364. 2008.

O'KEEFFE, M.; CINNÉIDE, M. Ó. Search-based refactoring for software maintenance, Journal of Systems and Software, Volume 81, Issue 4, Pages 502–516. 2008.

OPDYKYE, W.F. **Refactoring object-oriented frameworks**. 1992. Tese (Doutorado em Ciência da Computação), Champaign, IL, USA. UMI Order No. GAX93-05645.

PAULI, G. B.; PIVETA, E. K. **Searching for Refactoring Opportunities to Apply the Strategy Pattern**. 2014. In: Simpósio Brasileiro de Sistemas de Informação (SBSI), 2014, Londrina. Anais do Simpósio Brasileiro de Sistemas de Informação. Porto Alegre: SBC, 2014. Pages 1-15.

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.

ROBERTS, D. B. Practical analysis for refactoring. 1999. Tese (Doutorado em Ciência da Computação) , Champaign, IL, USA.

SIMON, F., STEINBRUCKNER, F., LEWERENTZ, C., Metrics based refactoring. In: European Conference on Software Maintenance and Reengineering, 5., 2001, Lisbon, Portugal. **Proceedings…** [S.1.: s.n.], 2001. P.30-38.

SENG, O., STAMMEL, J., BURKHART, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Annual Conference on Genetic and Evolutionary Computation, 8., 2006. **Proceedings…**[S.1.: s.n.], 2006. P.1909-1916.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of Move Method Refactoring Opportunities, IEEE Transactions on Software Engineering, vol.35, no.3, pp.347,367, May-June 2009.

TOURWÉ, T.; MENS, T. Identifying Refactoring Opportunities Using Logic Meta Programming. In: European Conference on Software Maintenance and Reengineering. CSMR, 7., 2003, Benevento, Italy. **Proceedings…** [S.1.: s.n.], 2003. P.91-100.

TEKIN, U., ERDEMIR, U., BUZLUCA, F. Mining object-oriented design models for detecting identical design structures. In: International Workshop on Software Clones, 6. 2012. P.43-49.

ANEXOS

ANEXO A – Catálogo de Bad Smells – Limitações - em código-fonte

Código Duplicado

Este problema consiste de duas expressões idênticas em dois métodos na mesma classe, ou duas classes não associadas. Essa duplicação pode ser eliminada aplicando-se a refatoração Extrair Método (FOWLER, 1999). Nesse caso, o código é implementado uma única vez e chamado no local de origem onde estava o código duplicado. Se o código duplicado encontra-se em duas classes distintas, pode-se aplicar a refatoração Extrair Classe (FOWLER, 1999).

Método Longo

Consiste de um método com uma implementação de lógica com várias, dezenas ou até centenas de linhas de código. Vários problemas podem ser originados da implementação de métodos longos. Primeiro, pode existir código duplicado. Segundo, será difícil reaproveitar parte do código em outro local. A solução é dividir o método longo em métodos menores, usando-se a refatoração Extrair Método (FOWLER, 1999).

Classe Larga

Uma boa prática da programação orientada a objetos sugere que uma classe deve ter uma única responsabilidade (princípio da responsabilidade única). Quando uma classe possui muitas instâncias no código, isso sugere que ela está fazendo muito, ou pelo menos realizando trabalho que deveria ser feito por outras classes. Algumas refatorações podem ajudar a reduzir classes longas, como por exemplo, Extrair Classe (FOWLER, 1999), Extrair Subclasse (FOWLER, 1999), entre outras.

Lista Longa de Parâmetros

Um método que recebe muitos parâmetros sugere que classes possam estar compartilhando muita informação, ou ainda, dados globais. Se um método com uma longa lista de parâmetros está sendo chamado em vários locais do código, uma mudança em um deles pode desencadear uma cadeia muito grande de alterações em vários locais do código. Algumas refatorações podem ajudar a resolver esse problema, por exemplo, a Substituir Parâmetro por Método e Introduzir Parâmetro Objeto (FOWLER, 1999). Neste último caso, por exemplo, uma longa lista de parâmetros se resume a uma única estrutura (classe, por exemplo), que internamente define atributos.

Mudança Divergente

Ocorre quando uma classe é alterada geralmente de diferentes formas por diferentes motivos. Normalmente, uma mudança requer a alteração de várias classes. Nesse caso, uma

boa solução seria isolar ou encapsular código passível de mudança para que comporte a mudança em um único local, quando necessária.

Tiro Cirúrgico

Esta situação é o oposto de Mudança Divergente (Divergent Change). Em uma pequena mudança de código, é necessário realizar várias pequenas mudanças em várias classes.

Recurso Invejoso

Ocorre quando um método realiza a maioria das suas funções consultando outros métodos ou mesmo outros objetos, a fim de obter dados, realizar cálculos, lógica etc.

Dados Agregados

Ocorre normalmente quando dados costumam aparecer juntos em várias situações: campos em classes, em lista de parâmetros em assinaturas de métodos etc. Uma solução seria aplicar a refatoração Introduzir Parâmetro Objeto (FOWLER, 1999).

Obsessão Primitiva

Existem basicamente dois tipos de dados na maioria das linguagens: os primitivos e as estruturas (normalmente classes, formadas a partir da composição de outros tipos primitivos). A obsessão primitiva ocorre normalmente com desenvolvedores com pouca experiência técnica em orientação a objetos. Refatorações como Substituir Dado por Objeto e Substituir Código de Tipo por State / Strategy (FOWLER, 1999) podem ajudar a solucionar problemas com obsessão primitiva.

Sentenças switch

Testar várias possibilidades com sentenças switch (case) tende a duplicar código. Se um novo teste for necessário na sentença, para comportar uma nova possibilidade, e esse teste for feito em vários locais, então várias alterações serão necessárias. Muitas vezes os testes podem ser substituídos por polimorfismo, usando a refatoração Substituir Condicional por Polimorfismo (FOWLER, 1999).

Hierarquia de Herança Paralela

Nesse caso, cada vez que uma classe se torna subclasse de outra, também é necessário criar uma subclasse de outra classe (pois existem hierarquias paralelas). As refatorações Mover Método e Mover Campo (FOWLER, 1999) ajudam a minimizar problemas relacionados a esta limitação.

Classe Preguiçosa

Cada classe criada no sistema de software tem um custo para ser mantida e evoluída. Classes que não estão fazendo o suficiente para pagarem seu próprio custo devem ser eliminadas. A refatoração Recolher Hierarquia (FOWLER, 1999) pode minimizar problemas associados a classes preguiçosas.

Generalidade especulativa

Este problema ocorre quando se projeta uma solução genérica para uma possível situação que possivelmente nunca ocorrerá, tornando objetos flexíveis o suficiente para tratar qualquer tipo de situação, por exemplo, criando-se classes abstratas que não realizam muito. A refatoração Recolher Hierarquia (FOWLER, 1999) pode minimizar problemas associados à generalidade.

Campo Temporário

Muitas vezes um objeto tem sua variável de instância configurada somente sobre certas circunstâncias. Tal código é difícil de entender. Um exemplo ocorre quando um algoritmo depende de uma série de variáveis. A refatoração Extrair Classe (FOWLER, 1999) pode minimizar problemas associados á essas variáveis "pobres".

Cadeia de Mensagens

Ocorre quando um cliente consulta um objeto a partir de outro objeto, que por sua vez repassa a consulta a outro objeto, que consulta outro, e assim por diante, causando uma cadeia de chamadas. Qualquer mudança nos relacionamentos intermediários pode ocasionar grandes problemas. As refatorações Extrair Método e Esconder Delegação (FOWLER, 1999) minimizam problemas desse tipo.

Homem do Meio

O encapsulamento é um dos principais recursos da programação orientada a objetos, a habilidade de esconder detalhes internos do resto do mundo. A delegação ocorre quando uma classe recebe chamadas e utiliza um objeto interno para realizar a maioria de suas funções (delegação). A refatoração Remover Homem do Meio (FOWLER, 1999) permite reduzir este problema.

Intimidade inadequada

Muitas vezes classes se tornam demasiadamente íntimas e perdem muito tempo investigando partes privadas da própria classe. As refatorações Mover Método (FOWLER, 1999) e Mover Campo (FOWLER, 1999) ajudam a separar partes para reduzir essa intimidade.

Classes Alternativas com Diferentes Interfaces

Ocorre com métodos que fazem a mesma coisa, mas com diferentes assinaturas. As refatorações Mover Método e Extrair Superclasse (FOWLER, 1999) ajudam a resolver esse tipo de problema.

Biblioteca de Classes Incompleta

Objetos são o princípio básico da reutilização de código. Bibliotecas de classes realizam boa parte de tarefas comuns usando esse conceito. Porém, normalmente não se pode alterar uma biblioteca base de classes. A refatoração Introduzir Método Estrangeiro (FOWLER, 1999) permite simular a inclusão de um método em uma classe que não pode ser estendida nem modificada.

Classe de Dados

Classes de dados normalmente contêm muitos campos públicos para representarem seus dados, e nada mais. Esse tipo de classe frequentemente trabalha como container de dados que são utilizados e repassados para outras classes. A refatoração Encapsular Campo (FOWLER, 1999) permite encapsular o acesso a esses campos.

Legado Recusado

Classes herdam métodos e dados de classes superiores, porém muitas vezes não precisam de todos os dados e comportamentos herdados para funcionar. Provavelmente há um problema na hierarquia. As refatorações Descer Método / Campo (FOWLER, 1999) permite deslocar métodos e campos para classes mais específicas, mantendo nas classes superiores somente o que realmente é comum a toda hierarquia.

Comentários

Comentários são bons, porém, um código com muitos comentários é um forte indício que ele está difícil de entender. Talvez seja mais apropriado dar nomes mais sugestivos a métodos, deixando mais claras suas responsabilidades, usando a refatoração Renomear Método (FOWLER, 1999).

ANEXO B – Catálogo de Padrões de Projeto (GoF)

Abstract Factory

Este padrão fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas (GAMMA et al., 1995).

Builder

Este padrão tem a intenção de separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações (GAMMA et al., 1995).

Factory Method

Este padrão tem a intenção de definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. Factory Method permite adiar a instanciação para subclasses (GAMMA et al., 1995).

Prototype

Este padrão especifica os tipos de objetos a serem criados usando uma ou mais instância prototípica e criar novos objetos copiando este protótipo (GAMMA et al., 1995).

Singleton

Este padrão garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela (GAMMA et al., 1995).

Adapter

A intenção do padrão Adapter, também conhecido como Wrapper, é converter a interface de uma classe em outra interface esperada pelos clientes. Permite que classes trabalhem em conjunto, pois de outra forma não poderiam devido a terem interfaces incompatíveis (GAMMA et al., 1995).

Bridge

A intenção do padrão Bridge é desacoplar uma abstração de sua implementação, de modo que os dois possam variar independentemente (GAMMA et al., 1995).

Composite

Este padrão tem por intenção compor objetos em estruturas de árvore para representar hierarquias parte ou todo. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme (GAMMA et al., 1995).

Decorator

Este padrão tem a intenção de, dinamicamente, agregar responsabilidades adicionais a um objeto. As classes "decoradoras" fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades (GAMMA et al., 1995).

Façade

A intenção do padrão Façade é Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais elevado que faz o subsistema mais fácil de usar (GAMMA et al., 1995).

Flyweight

A intenção desse padrão é usar compartilhamento para suportar um grande número de objetos semelhantes de forma eficiente (GAMMA et al., 1995).

Proxy

A intenção do padrão Proxy é fornecer um substituto ou espaço reservado para outro objeto para controlar o acesso a ele (GAMMA et al., 1995).

Chain of Responsability

A intenção deste padrão é evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate (GAMMA et al., 1995).

Command

Este padrão tem por objetivo encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas (GAMMA et al., 1995).

Interpreter

Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem (GAMMA et al., 1995).

Iterator

A intenção do padrão Iterator é fornecer uma maneira de acessar os elementos de um objeto agregado sequencialmente sem expor sua representação (GAMMA et al., 1995).

Mediator

A intenção do padrão Mediator é definir um objeto que encapsula como um conjunto de objetos que precisam interagir. Mediator promove o baixo acoplamento por manter objetos

sem se referir um ao outro de forma explícita, e que lhe permite variar sua interação independentemente (GAMMA et al., 1995).

Memento

A intenção do padrão Memento é, sem violar o encapsulamento, capturar e tornar externo (exportar) o estado interno de um objeto para que o objeto possa ser restaurado para este estado mais tarde (GAMMA et al., 1995).

Observer

A intenção do padrão Observer é definir uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente (GAMMA et al., 1995).

State

A intenção do padrão State é permitir que um objeto altere seu comportamento quando seu estado interno muda (GAMMA et al., 1995).

Strategy

Este padrão tem por objetivo definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizem (GAMMA et al., 1995).

Template Method

Este padrão tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo (GAMMA et al., 1995).

Visitor

Este padrão tem por objetivo representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre as quais opera (GAMMA et al., 1995).

ANEXO C – Catálogo de Refatorações para Padrões de Projeto

Encadear Construtores

Quando múltiplos construtores contêm código duplicado, a refatoração Encadear Construtores sugere encadear os construtores para obter a menor quantidade de duplicação (KERIEVSKY, 2008).

Compor Método

Nos casos em que a compreensão da lógica de um método está difícil, a refatoração Compor Método sugere transformar toda essa lógica em um pequeno número de passos que revelam a intenção e que contenham um mesmo nível de detalhe (KERIEVSKY, 2008).

Encapsular Classes com Factory

Quando classes clientes instanciam diretamente classes que residem em um pacote e implementam uma interface em comum, a refatoração Encapsular Classes com Factory sugere tornar os construtores da classe não-públicos e deixar os clientes criarem instâncias da classe usando uma fábrica (Factory) (KERIEVSKY, 2008).

Encapsular Composite com Builder

Construir um Composite pode ser uma tarefa repetitiva, complicada e propensa a erros. Nesse caso, a refatoração Encapsular Composite com Builder sugere simplificar essa criação deixando um padrão Builder manipular os detalhes (KERIEVSKY, 2008).

Extrair Adapter

Quando uma classe adapta múltiplas versões de um componente, biblioteca, API ou outra entidade, a refatoração Extrair Adapter sugere extrair um Adapter para uma versão individual do componente, biblioteca, API ou entidade (KERIEVSKY, 2008).

Extrair Composite

Quando subclasses em uma mesma hierarquia implementam o mesmo Composite, a refatoração Extrair Composite sugere extrair uma superclasse que implemente este Composite (KERIEVSKY, 2008).

Extrair Parâmetro

Um método ou construtor atribui um valor instanciado localmente para um atributo. Nesse caso, a refatoração Extrair Parâmetro sugere atribuir ao atributo um parâmetro fornecido por um cliente ao extrair metade da sentença de atribuição para um parâmetro (KERIEVSKY, 2008).

Formar Template Method

Se dois métodos em subclasses executam passos similares na mesma ordem, embora sejam diferentes, a refatoração Formar Template Method sugere generalizar os métodos extraindo seus passos em métodos com assinaturas idênticas, movendo os métodos generalizados para a superclasse visando formar uma Template Method (KERIEVSKY, 2008).

Internalizar Singleton

Quando um código precisa de acesso a um objeto, mas não necessita de um ponto de acesso global a ele, a refatoração Internalizar Singleton sugere mover as funcionalidades do Singleton para uma classe que armazene e forneça acesso ao objeto, eliminando então este Singleton (KERIEVSKY, 2008).

Introduzir Objeto Nulo

Muitas vezes, a lógica para manipular um objeto ou atributo nulo está espalhada e duplicada ao longo do código. Nesse caso, a refatoração Introduzir Objeto Nulo sugere substituir essa lógica que fornece o mesmo comportamento apropriado por um objeto "nulo" (Null Object) (KERIEVSKY, 2008).

Introduzir Criação Polimórfica com Factory Method

Se classes em uma mesma hierarquia implementam um método de maneira similar, exceto por um passo da criação de objetos, a refatoração Introduzir Criação Polimórfica com Factory Method pode ser usada para criar uma versão única do método na superclasse que chamada um Factory Method para manipulara instanciação (KERIEVSKY, 2008).

Limitar Instanciação com Singleton

Um código pode criar múltiplas instâncias de um objeto, e isso pode consumir muita memória e prejudicar o desempenho do sistema. Nesse caso, a refatoração Limitar Instanciação com Singleton sugere substituir as múltiplas instâncias por um Singleton (KERIEVSKY, 2008).

Mover Acumulação para Parâmetro Coletor

Quando um único método está grande e difícil de gerenciar devido ao acúmulo de informações sobre uma variável local, a refatoração Mover Acumulação para Parâmetro Coletor sugere acumular os resultados em um parâmetro coletor que passado para os métodos extraídos (KERIEVSKY, 2008).

Mover Acumulação para Visitor

Quando um método acumula informação de classes heterogêneas, a refatoração Mover Acumulação para Visitor sugere mover a tarefa de acumulação para um Visitor que possa visitar cada classe para acumular a informação (KERIEVSKY, 2008).

Mover Conhecimento de Criação para Factory

Quando os dados e o código utilizados para instanciar uma classe estão espalhados por diversas classes, a refatoração Mover Conhecimento de Criação para Factory sugere mover este comportamento para a uma única classe de fábrica (Factory) (KERIEVSKY, 2008).

Mover Embelezamento para Decorator

Quando um código fornece um embelezamento para uma responsabilidade básica de uma classe, a refatoração Mover Embelezamento para Decorator sugere mover esta lógica para uma classe de decoração, de acordo com o padrão Decorator (KERIEVSKY, 2008).

Substituir Envio Condicional por Command

Quando uma lógica condicional é usada para enviar requisições e executar ações, a refatoração Substituir Envio Condicional por Command sugere criar um comando (Command) para cada ação, armazenar estes comandos em uma coleção e substituir a lógica condicional por código para obter e executar comandos (KERIEVSKY, 2008).

Substituir Lógica Condicional por Strategy

Lógica condicional em um método controla qual dentre as diversas variantes de um cálculo é executada. Nesse caso, a refatoração Substituir Lógica Condicional por Strategy sugere aplicar um padrão Strategy para cada variante e fazer com que o método delegue o cálculo para uma instância de Strategy (KERIEVSKY, 2008).

Substituir Construtores por Métodos de Criação

Construtores em uma classe tornam mais difícil decidir qual construtor chamar durante o desenvolvimento. A refatoração Substituir Construtores por Métodos de Criação sugere substituir os construtores por métodos de criação que revelem a intenção do "construtor" e que retornem instâncias da classe (KERIEVSKY, 2008).

Substituir Notificações Hard-Coded por Observer

Quando subclasses são codificadas especificamente para notificar uma única instância de outra classe, a refatoração Substituir Notificações Hard-Coded por Observer sugere remover as subclasses ao tornar sua superclasse capaz de notificar uma ou mais instâncias de qualquer classe que implemente uma interface do tipo Observer (KERIEVSKY, 2008).

Substituir Linguagem Implícita por Interpreter

Se diversos métodos em uma classe combinam elementos de uma linguagem implícita, a refatoração Substituir Linguagem Implícita por Interpreter sugere a definição de classes para os elementos da linguagem implícita de maneira que as instâncias possam ser combinadas para formar expressões interpretáveis (KERIEVSKY, 2008).

Substituir Árvore Implícita por Composite

Se um código informa implicitamente uma estrutura de árvore, usando uma representação primitiva, tal como uma string, a refatoração Substituir Árvore Implícita por Composite sugere substituir essa representação por um padrão Composite (KERIEVSKY, 2008).

Substituir Distinções Um/Muitos por Composite

Quando uma classe processa objetos individuais e múltiplos usando trechos de códigos separados, a refatoração Substituir Distinções Um/Muitos por Composite sugere usar um Composite para produzir um trecho de código capaz de manipular objetos individuais ou múltiplos (KERIEVSKY, 2008).

Substituir Condicionais que Alteram Estado por State

Se as expressões condicionais que controlam as transições de um estado de um objeto são complexas, a refatoração Substituir Condicionais que Alteram Estado por State sugere substituir essas expressões por classes State que manipulam os estados específicos e as transições entre eles (KERIEVSKY, 2008).

Substituir Código de Tipo por Classe

Um tipo de um atributo, por exemplo, uma string ou inteiro, pode falhar em protegê-lo de atribuições inseguras e comparações de igualdades inválidas. Nesse caso, a refatoração Substituir Código de Tipo por Classe sugere restringir as atribuições e comparações de igualdade tornando o tipo do atributo uma classe (KERIEVSKY, 2008).

Unificar Interfaces

Quando é preciso que uma superclasse ou interface possa suportar a mesma interface de uma subclasse, a refatoração Unificar Interfaces sugere encontrar todos os métodos públicos na subclasse que estão ausentes na superclasse ou interface e adicionar cópias destes métodos ausentes à superclasse, alterando cada um deles para realizar comportamento nulo (KERIEVSKY, 2008).

Unificar Interfaces com Adapter

No caso em que classes clientes interagem com duas classes, uma das quais tem uma interface preferida, a refatoração Unificar Interfaces com Adapter sugere unificar as interfaces com um Adapter (KERIEVSKY, 2008).