PONTÍFICIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ENGENHARIA DE SOFTWARE

ANA ELIZA BARBOSA

UMA INVESTIGAÇÃO DA CORRELAÇÃO ENTRE *ANTI-PATTERNS*, MÉTRICAS E ODORES DE *SOFTWARE*

SÃO PAULO

2014

ANA ELIZA BARBOSA

UMA INVESTIGAÇÃO DA CORRELAÇÃO ENTRE *ANTI-PATTERNS*, MÉTRICAS E ODORES DE *SOFTWARE*

Monografia apresentada à Pontíficia Universidade Católica de São Paulo como requisito à conclusão curso de Engenharia de Software

Orientador: Daniel Couto Gatti

SÃO PAULO

2014

PONTÍFICIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ENGENHARIA DE SOFTWARE

ANA ELIZA BARBOSA

UMA INVESTIGAÇÃO DA CORRELAÇÃO ENTRE *ANTI-PATTERNS*, MÉTRICAS E ODORES DE *SOFTWARE*

Monografia aprovada em ____/____/_____ para obtenção do certificado do curso de Engenharia de Software.

_____

Professor Daniel Couto Gatti

Resumo

O objetivo deste trabalho é definir um conjunto de métricas e seus respectivos valores para evidenciar possíveis odores e *anti-patterns* em um projeto de *software*. Utilizando como metodologia a revisão bibliográfica e prova de conceito para identificação do odor *Middle Man*. Por meio da revisão bibliográfica são extraídos os conceitos de métricas para mensuração de propriedades do código fonte, e os odores e os *anti-patterns* que evidenciam problemas de *design* em um projeto. A prova de conceito demonstra o processo de identificação do odor em uma classe, a coleta e análise das métricas dessa classe e a definição do conjunto de métricas que evidenciam este odor. Termina por concluir que existe a possibilidade de se utilizar um conjunto de métricas para evidenciar possíveis odores em um projeto, mas que ainda é necessário realizar a inspeção do código fonte para confirmar ou não a presença do odor.

Palavras Chave: Métricas, Odores, *Anti-Patterns*.

Abstract

The objective of this work is to define a set of metrics and their values to show potential code smells and anti-patterns in a software project. Methodology is composed of literature review and proof of concept for identification of code smell Middle Man. Through literature review are extracted concepts of metrics for measuring the properties of the source code, and code smells and anti-patterns that evidence design problems in a project. The proof of concept demonstrates the process of code smell identification in a class, the gathering and analysis of metrics from that class and defining the set of metrics that evidence this code smell. Ends by concluding that there is a possibility of using a set of metrics to evidence potential code smells on a project, but it is still necessary to carry out the inspection of the source code to confirm or not the presence of the code smell.

Keywords: Metrics, Code Smells, Anti-Patterns.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

IDE *Integrated Development Enviroment*

Sumário

1. Introdução

O código fonte de um programa é uma sequência de instruções computacionais escrita de forma ordenada, utilizando uma linguagem que pode ser facilmente interpretada por humanos e que é armazenado em um arquivo.

Softwares são compostos por um ou mais arquivos de código fonte. O *design* de um código fonte busca organizar a distribuição das instruções de forma a permitir que os desenvolvedores compreendam a estruturação de um *software* e facilite a realização da manutenção.

A qualidade do *design* pode ser mensurada por meio de métricas, que são medidas quantitativas de alguma parte ou especificação de um *software*, por exemplo: total de linhas de código, *bugs* por linha de código, tempo de execução do *software*, entre outras.

As métricas são medidas objetivas, pois a aplicação de sua medição em uma classe ou pacote de classes gera um resultado mensurável que pode ser avaliado individualmente ou dentro de uma escala de valores conhecida. A análise do valor da medida pode ser subjetiva, pois a coleta dessa métrica não evidencia claramente problemas que permeiam o *design* do *software*, pode evidenciar apenas problemas pontuais (ex.: classes muito extensas).

Em um *software* com problemas de *design* pode-se encontrar pontos de melhoria, ou seja, trechos de código que não foram projetados corretamente na concepção do *software* ou trechos onde o *design* se deteriorou ao longo do tempo devido a manutenções. Esses pontos de melhoria podem ser classificados como odores ou *anti-patterns.*

Os odores de *software* são indícios que se manifestam no código fonte conforme o *design* da aplicação se deteriora. Essa deterioração do *design* pode dificultar o entendimento da codificação e a realização de manutenção do *software*.

Os *anti-patterns* são soluções de codificação recorrentes para os problemas de *design* conhecidos e que tornam os códigos ineficientes e contra produtivos, ou seja, é a aplicação de más práticas de codificação e que são amplamente adotadas pelos desenvolvedores.

Para encontrar odores e *anti-patterns* em um *software* é necessário a realização da inspeção do código fonte, portanto pode-se dizer que sua busca é subjetiva. O relacionamento dos odores e *anti-patterns* com o código fonte é objetiva, pois ao encontra-los e corrigi-los temos um ganho real de legibilidade e manutenabilidade do código fonte do *software*.

As métricas podem ser obtidas por meio da utilização de complementos das IDEs ou de *softwares* especializados de forma automática, porém a análise desses resultados não evidenciam os problemas de *design* da aplicação. Os odores e *anti-patterns* demandam esforço humano, utilizando a inspeção de código para serem encontrados, permitindo a correção do código e gerando uma melhoria da qualidade do código fonte.

Desta forma este trabalho vai identificar um conjunto de métricas que indique a existência de um odor ou um *anti-pattern* em um *software*. Aliando a objetividade na coleta das métricas ao ganho real de qualidade obtido ao corrigir um odor ou *anti-pattern*.

1.1. Objetivos

O objetivo geral deste trabalho é definir um conjunto de métricas e seus respectivos valores para evidenciar possíveis odores e *anti-patterns* em um projeto.

Como objetivos específicos temos:

- Levantar as definições de métricas de *software*;
- Levantar as definições de odores e *anti-patterns* com foco no *Middle Man*, que será o tema principal da pesquisa e;
- Levantar ferramentas para automatizar a coleta de métricas em um projeto de *software*.

2. Conceitos Fundamentais de Avaliação de Código Fonte

Neste capítulo são discutidos os principais conceitos sobre métricas, odores e *anti-patterns* utilizados no desenvolvimento da pesquisa.

2.1. Métricas

Métricas são indicações mensuráveis de algum aspecto quantitativo de um sistema, que geralmente compreende escopo, tamanho, custo, risco e tempo decorrido (DEMARCO, 1982). Esses valores não são obtidos através de observações de desenvolvedores ou gestores sobre o andamento do projeto, mas sim por meio de mensurações de artefatos gerados durante o projeto.

Segundo DEMARCO, 1982, uma métrica útil deve possuir quatro características primordiais. Deve ser:

- Mensurável: o fenômeno em questão deve ser passível de mensuração, caso contrário não poderá ser qualificado como uma métrica;
- Independente: sua mensuração não pode ser influenciada de forma consciente pela opinião da equipe do projeto de *software*;
- Explicável: sua mensuração bruta não deve permitir diferentes interpretações pela equipe do projeto de *software*;
- Precisa: a documentação da mensuração deve ser acrescida do grau de exatidão (quando necessário) para auxiliar a tomada de ações pela equipe do projeto de *software*.

Para obter as métricas necessárias para o desenvolvimento dessa pesquisa foi utilizado Eclipse Metrics Plugin 1.3.8[1], que consolidou em apenas uma ferramenta uma extensa lista de métricas coletadas em diversos livros.

Na Tabela 1 pode-se encontrar as métricas presentes no Eclipse Metrics Plugin 1.3.8, tradução livre do autor.

Número de Pacotes (NOP)	Total de pacotes de classes.
Número de Interfaces (NOI)	Total de interfaces.
Número de Classes (NOC)	Total de classes.
Número de Atributos (NOF)	Total de atributos.
Número de Métodos (NOM)	Total de métodos.
Número de Parâmetros (PAR)	Total de parâmetros de métodos.
Número de Métodos Sobrepostos (NORM)	Total de métodos sobrepostos em uma classe.
Número de Atributos Estáticos (NSF)	Total de atributos estáticos.

[1] Eclipse Metrics Plugin 1.3.8 é um *plugin* desenvolvido para a IDE Eclipse que permite a coleta de métricas do projeto, pacote de classes ou classe que o desenvolvedor estiver trabalhando. Todos os passos para instalação e utilização deste *plugin* encontram-se no site http://metrics2.sourceforge.net/ que é mantido por seus desenvolvedores.

Número de Métodos Estáticos (NSM)	Total de métodos estáticos.
Número de Filhos (NSC)	Total de subclasses diretas de uma classe.
Linhas de Código (LOC)	Total de linhas de código.
Linhas de Código do Método (MLOC)	Total de linhas de código dentro do corpo do método.
Profundidade de Blocos Aninhados (NBD)	Quantidade de níveis de profundidade de blocos de código aninhados em um método.
Profundidade da Árvore de Herança (DIT)	Distância da classe em questão até a classe *Object* na árvore de herança.
Índice de Especialização (SIX)	Indica a proporção de entre a quantidade métodos da superclasse que foram sobrescritos e a quantidade de métodos da classe em questão.
Falta de Coesão de Métodos (LCOM)	Indica a quantidade de métodos que não estão relacionados a sua classe através da utilização das variáveis de instância.
Complexidade Ciclomatica de McCabe (VG)	Total de fluxos possíveis em um método. É incrementada para instruções *if*, *for*, *while*, *do*, *case*, *catch*, operadores ternários e para operadores condicionais *&&* e ||.
Peso dos Métodos por Classe (WMC)	Soma da Complexidade Ciclomatica de McCabe de todos os métodos da classe.
Acoplamento Aferente (Ca)	Total de classes externas a um pacote de classes que dependem de classes internas a esse pacote.

Acoplamento Eferente (Ce)	Total de classes internas a um pacote de classes que dependem de classes externas a esse pacote.
Instabilidade (I)	Indica o grau de resistência a mudança/alteração de uma classe em um pacote de classes, onde pacotes com muitos acessos externos tendem a ser mais estáveis e mais difíceis de mudarem e os que possuem menos acessos externos são mais instáveis e mais passíveis de mudança.
Abstração (A)	Indica a proporção de classes abstratas e de classes concretas dentro de um pacote de classes.
Distância Normalizada da Sequência Principal (D')	Indica o equilíbrio do pacote de classes entre abstração e instabilidade, onde os pacotes de classes idealmente deveriam possuir apenas classes abstratas e sem dependências ou apenas classes concretas e com dependências.

Fonte: BOISSIER; CASSELL, 2014.

Tabela 1 – Lista de Métricas

É importante ressaltar que a definição de uma métrica não compreende uma interpretação para o resultado obtido, conforme a característica "Explicável" que diz que os resultados não devem ter dupla interpretação, ou seja, devem ser apenas mensuráveis. A interpretação e uso do resultado obtido deve ser conduzido pela equipe de projeto de *software* ou analistas de qualidade para auxiliar a tomada de decisões.

No próximo item serão discutidos os conceitos de odores de *software*.

2.2. Odores

Odores de *software* são indícios que se manifestam no código fonte conforme o *design* da aplicação se deteriora (FOWLER et al., 1999). Durante o desenvolvimento de uma aplicação, várias modificações no código acontecem, porém estas podem não ser comportadas pelo *design* incialmente concebido e nem sempre os desenvolvedores possuem o conhecimento do projeto ou tempo disponível para revisar este *design* a fim de corrigir os pontos de deterioração.

Segundo FOWLER et al., 1999, com a degradação da aplicação pode-se observar alguns problemas, tais como:

- Rigidez: dificuldade de modificar a aplicação, pois uma modificação pontual força diversas modificações em outras classes;
- Fragilidade: modificações na aplicação causam retrabalho em outras classes que não possuem relação conceitual com a modificação;
- Imobilidade: dificuldade em reaproveitar partes de uma aplicação devido ao esforço e risco em modularizá-lo;
- Viscosidade: dificuldade em preservar o *design* da aplicação durante a modificação do código;
- Complexidade desnecessária: existência de estruturas complexas que não adicionam reais benefícios à aplicação;
- Repetição desnecessária: repetição de estruturas que poderiam ser unificadas com abstrações;
- Opacidade: dificuldade em ler e entender o código fonte, pois suas variáveis e métodos não expressam de forma clara a real intenção da aplicação;
- Entre outros.

Na Tabela 2 são definidos um conjunto de odores comuns encontrados em códigos fontes em geral.

Duplicated Code	O mesmo trecho de código pode ser encontrado em várias classes do projeto.
Long Method	São métodos muito longos e que realizam diversas atividades distintas. São de difícil compreensão e seu nome não explicita todas as operações que o método realiza.
Large Class	A classe possui muitas atribuições (que deveriam estar divididas entre duas ou mais classes), consequentemente muitas variáveis de instância e métodos.
Long Parameter List	A lista de parâmetros de um método é muito longa e contêm apenas tipos primitivos, como consequência é difícil de compreender e muito suscetível a mudanças em sua assinatura quando houver a necessidade de mais dados.
Divergent Change	Ocorre em classes que concentram mais de uma responsabilidade dentro da aplicação. Uma classe sofre modificações a partir de requisitos não relacionados, como, por exemplo, a troca do banco de dados e a adição de um novo tipo de imposto.
Shotgun Surgery	Classes devem concentrar responsabilidades sobre um aspecto da aplicação. Este odor ocorre quando essas responsabilidades estão espalhadas em diversas classes. Quando é necessário realizar uma modificação, que deveria ser pontual, os desenvolvedores acabam codificando pequenas modificações em várias classes da aplicação.

Feature Envy	Objetos são criados para agrupar dados e processos que utilizam estes dados. Este odor é encontrado em métodos que utilizam mais dados de outro objeto do que seus próprios dados.
Data Clumps	É um conjunto de variáveis que é encontrado em diversas partes do código; essas variáveis não fazem sentido separadamente na aplicação, porém não foram devidamente agrupadas em um objeto.
Primitive Obsession	É caracterizado pela relutância do desenvolvedor em criar pequenos objetos para agrupar dados ou realizar pequenas tarefas, como, por exemplo, uma classe Dinheiro que combine números e moeda.
Switch Statements	Sua utilização no código evidencia claramente a falta de polimorfismo no *design* da aplicação e o maior problema que esta declaração acarreta é a duplicação de código, pois dificilmente essa decisão será tomada em apenas um ponto do projeto.
Parallel Inheritance Hierarchies	É um caso especial de *Shotgun Surgery*, onde a criação de uma subclasse em uma hierarquia obriga a criação de outra subclasse em outra hierarquia.
Lazy Class	São classes que tiveram seu escopo muito reduzido, não demonstrando valor de negócio suficiente para existirem isoladamente. A redução do escopo geralmente ocorre devido à refatorações ou eliminação de variáveis e/ou métodos que não são mais necessários na aplicação.
Speculative Generality	É caracterizado por funcionalidades que os desenvolvedores acreditam que serão necessárias no futuro, porém não existe nenhuma demanda imediata para elas, resultando em projetos maiores do que o esperado, complexos e de difícil manutenção.

Temporary Field	São variáveis de instância que apenas auxiliam a execução de um método da classe, não pertencem ao contexto do objeto e causam problemas de entendimento aos desenvolvedores.
Message Chains	São sequências de chamadas que geram uma cadeia de navegação que deveria ser omitida para seus clientes. Seu maior problema é alterar seus clientes quando partes intermediárias da estrutura de navegação são modificadas.
Middle Man	É uma classe que existe apenas para realizar delegações simplistas, sem adicionar nenhuma funcionalidade à aplicação, gerando apenas mais uma camada para ser acessada.
Inappropriate Intimacy	É caracterizado por classes que conhecem muito sobre o funcionamento interno dos atributos e métodos de outra classe. Também é observado nos casos de herança, onde as subclasses acabam tendo acesso a comportamentos internos da superclasse.
Alternative Classes With Different Interfaces	É caracterizado por métodos que realizam a mesma atividade, porém possuem assinaturas diferentes.
Incomplete Library Class	São bibliotecas que não permitem que os desenvolvedores adicionem ou modifiquem comportamentos que serão utilizados na aplicação, devido à uma *design* mal planejado.
Data Class	São classes que possuem apenas variáveis de instância, métodos *get* e *set* e que são manipuladas por outras classes, separando os dados de seu processamento.
Refused Bequest	É encontrado em casos onde a superclasse não foi generalizada de forma correta, obrigando suas subclasses a herdarem comportamentos que não precisam ou que não estão relacionadas às suas responsabilidades.

Comments	Comentários não são considerados más práticas, porém quando o desenvolvedor sente a necessidade de comentar o que determinado trecho de código faz existe um indício de que este código está no lugar errado. Se há, por exemplo, a necessidade de comentar um trecho de código dentro de um método, há uma grande chance de que este trecho deva pertencer a um novo método, cujo nome seja autoexplicativo.

Fonte: FOWLER; BECK; BRANT et al, 1999, p.75-88.

Tabela 2 – Lista de Odores.

No próximo item serão discutidos os conceitos de *anti-patterns*.

2.3. *Anti-Patterns*

Um *anti-pattern* é uma forma literária que descreve uma solução comum e recorrente para um problema de desenvolvimento de *software* e que gera consequências negativas para o *design* da aplicação (BROWN, 1998). Pode ser causado pela falta de conhecimento ou inexperiência do desenvolvedor para resolver determinado problema ou pela aplicação de *patterns* em contextos errados.

Na Tabela 3 são definidos um conjunto de *anti-patterns* comuns encontrados em códigos fontes em geral.

The Blob	É encontrado em projetos onde uma única classe monopoliza o processamento e outras classes são usadas apenas para encapsular dados, ou seja, mesmo sendo desenvolvido em linguagens orientadas a objeto o projeto apresenta características de programação procedimental.
Lava Flow	É comumente encontrado em projetos que nasceram com intuito de serem meramente exploratórios porém acabaram sendo utilizados em produção. Quando este projeto passa a ser utilizado em produção, seus desenvolvedores não realizam a limpeza desse projeto, resultando em uma massa de linhas de código com complexidade tão elevada que a equipe de desenvolvimento sempre relutará em realizar manutenções futuras no código.
Functional Decomposition	É o resultado do trabalho de programadores experientes em linguagens procedimentais atuando em aplicações orientadas a objeto. Como resultado pode-se encontrar classes que se comportam como sub-rotinas, o que torna a compreensão da arquitetura impossível e testes são impraticáveis.
Poltergeists	São classes com papéis muito limitados e com ciclo de vida muito curto, que são instanciados apenas para invocar métodos de outras classes do projeto. São construídos sob um péssimo *design* pois são desnecessários, consomem recursos ao serem instanciados e não respeitam regras de orientação a objetos.

Spaghetti Code	É encontrado em projetos pouco estruturados, onde a codificação e adição de novas funcionalidades comprometem de forma tão profunda a pouca estrutura existente que o código passa a ser incompressível até mesmo para o desenvolvedor que o criou.
Cut-and-Paste Programming	É identificado pela presença de vários trechos de códigos similares espalhados pelo projeto, que degenera a reutilização do código e torna sua manutenção impraticável, causando, por exemplo, o mesmo retrabalho em diversas partes do projeto.

Fonte: BROWN, 1998, p. 42-78.

Tabela 3 – Lista de *Anti-Patterns*.

Os *anti-patterns Continuous Obsolescence*, *Ambiguous Viewpoint*, *Boat Anchor*, *Golden Hammer*, *Dead End*, *Input Kludge*, *Walking Through a Minefield* e *Mushroom Management* encontrados na bibliografia não serão abordados neste trabalho, pois são *anti-patterns* de gestão de projetos, não sendo possível evidenciá-los através da aplicação de métricas no código fonte.

3. As Métricas e o Odor *Middle Man*

Inicialmente foi realizado o estudo bibliográfico sobre as métricas, os odores e os *anti-patterns* de *software*.

Deste estudo inicial foi escolhido o odor *Middle Man* para realizar a prova de conceito, pois foi considerado um odor comum nas aplicações escolhidas para o desenvolvimento desta pesquisa.

Para iniciar esta pesquisa foi escolhido um projeto feito em linguagem Java, onde a localização dos quatro *Middle Mans* era conhecida. Foi utilizado um *plugin* do Eclipse chamado Eclipse Metrics Plugin 1.3.8 que possibilitou a obtenção das métricas de classe e pacote dos *Middle Man* do projeto conhecido.

Com estes dados foi feito o relacionamento entre as métricas e intervalos de valores que se repetiram para os odores no projeto e criado um conjunto inicial de métricas para identifica-lo.

Foi gerada uma lista com as métricas de todas as classes e pacotes de classe do projeto. Essa lista foi filtrada de acordo com o conjunto das métricas e intervalos de valores de identificação do *Middle Man*, porém surgiram muitos falsos positivos[2] na filtragem.

O conjunto de métricas do *Middle Man* foi comparado com o conjunto de métricas de cada falso positivo, a fim de encontrar outras métricas e intervalos de valores que pudessem remover o falso positivo da listagem.

O conjunto inicial de métricas foi refinado e aplicado novamente no projeto conhecido, listando apenas os *Middle Man* esperados inicialmente.

Foi escolhido um projeto desconhecido onde essas métricas foram aferidas a fim de listar as classes que sugerem a presença do odor. Após a filtragem

[2] Falsos Positivos são classes que não possuem a estrutura de delegação de um *Middle Man,* porém possuem os mesmos valores para o conjunto de métricas.

foi realizada a inspeção do código a fim de validar se as métricas apontaram todos os *Middle Man* do projeto.

4. Identificando o Odor *Middle Man* no Código Fonte

A prova de conceito utilizará o odor *Middle Man* que é caracterizado por delegações simplistas. Considerando esse aspecto, pode-se concluir que um *Middle Man* é uma classe de baixa complexidade e que caracteriza uma camada extra de uma classe que nem sempre agrega valor ao sistema.

Nos próximos itens serão discutidos os passos utilizados para encontrar um conjunto de métricas para identificar o *Middle Man* em um projeto conhecido. Na sequência o conjunto de métricas será aplicado em um projeto desconhecido para avaliar sua eficácia.

Por fim discutiremos casos de classes que agregam valor ao sistema mas que podem ser confundidas com *Middle Mans*.

4.1. Definição do Conjunto de Métricas

Devido às características do *Middle Man* iniciou-se uma filtragem no projeto conhecido, na qual foram listadas apenas as classes que possuíam Complexidade Ciclomática com valor máximo de um, ou seja, classes cujo método mais complexo possui valor um. Ao final foram listadas 48 classes do projeto.

Na Tabela 4 pode-se observar que foram encontrados diversos falsos positivos, sendo estes 44 das 48 classes listadas.

Tipo	Total
Beans	1
Enum	9
Exceções Personalizadas	13
Interface	21
Middle Man	4

Tabela 4 – Total de classes com baixa complexidade ciclomática.

Na Tabela 5 pode-se observar o conjunto final de métricas que exclui os falsos positivos e que será utilizado para encontrar o odor *Middle Man* nos projetos.

Métrica	Valor
Complexidade Ciclomática	Igual a 1
Linhas de Código dos Métodos	Maior que 0
Número de Atributos	Igual a 1

Tabela 5 – Conjunto de métricas para encontrar um *Middle Man.*

A métrica **Linhas de Código dos Métodos** foi utilizada para excluir as interfaces da listagem, pois interfaces possuem apenas a assinatura do método enquanto o *Middle Man* possui métodos implementados. Porém essa métrica não é capaz de excluir da listagem os *Beans*, *Enums* e Exceções Personalizadas já que estes também possuem métodos implementados.

A métrica **Número de Atributos** com valor igual a um foi utilizada pois o *Middle Man* é uma delegação simplista, ou seja, ele apenas encapsula os métodos de outro objeto, sendo assim ele obrigatoriamente possui um atributo. Essa métrica também auxilia na exclusão dos *Beans* e *Enums*, já que estes geralmente

encapsulam mais de um atributo; e das Exceções Personalizadas pois geralmente estendem de classes de exceção padrão e implementam apenas os métodos básicos sem utilizar atributos, servindo somente para tratamento refinado de erros em um sistema.

4.2. Aplicação das Métricas em um Projeto Desconhecido

Para testar a eficácia do conjunto de métricas definido para o *Middle Man* foi escolhido o projeto JUnit 4[3], que é um *framework* desenvolvido para facilitar o criação de testes automatizados e que possui código fonte aberto disponível no GitHub.

Na Tabela 6 pode-se observar todas as classes que foram evidenciadas pelo conjunto de métricas definido para o *Middle Man*.

[3] JUnit 4 inspecionado no dia 01/11/2014, portanto não há garantias que a inspeção gere os mesmo resultados no futuro.

Classe	Complexidade Ciclomática	Linhas de Código dos Métodos	Número de Atributos
TestDecorator	1	6	1
JUnit4TestCaseFacade	1	5	1
PrintableResult	1	8	1
IgnoredClassRunner	1	3	1
StacktracePrintingMatcher	1	11	1
ThrowableCauseMatcher	1	7	1
ThrowableMessageMatcher	1	7	1
InitializationError	1	4	1
Fail	1	2	1
TestName	1	2	1
JunitCore	1	53	1

Tabela 6 – Possíveis *Middle Mans* do projeto JUnit.

Ao realizar a inspeção do código para essas classes pode-se observar que cinco delas (PrintableResult, StacktracePrintingMatcher, ThrowableCauseMatcher, ThrowableMessageMatcher e JunitCore) não possuiam estruturas parecidas com o *Middle Man*, pois continham métodos com várias linhas de código e diversas operações.

Na Tabela 7 pode-se observar o conjunto refinado de métricas que exclui os falsos positivos encontrados no projeto JUnit.

Métrica	Valor
Complexidade Ciclomática	Igual a 1
Linhas de Código dos Métodos	Maior que 0
Número de Atributos	Igual a 1
Número de Métodos	Igual ao total de Linhas de Código dos Métodos

Tabela 7 – Conjunto refinado de métricas para encontrar um *Middle Man.*

A métrica **Número de Métodos** foi adicionada ao conjunto para ser comparada com o **Linhas de Código dos Métodos**, pois como visto anteriormente o *Middle Man* é caracterizado por delegações simplista, ou seja, espera-se que cada método da classe possua apenas uma linha de código que delegará a ação ao atributo da classe.

Na Tabela 8 pode-se observar todas as classes que foram evidenciadas pelo conjunto refinado de métricas definido para o *Middle Man*.

Classe	Complexidade Ciclomática	Linhas de Código dos Métodos	Número de Atributos	Número de Métodos
TestDecorator	1	6	1	6
JUnit4TestCaseFacade	1	5	1	5
IgnoredClassRunner	1	3	1	3
InitializationError	1	4	1	4
Fail	1	2	1	2
TestName	1	2	1	2

Tabela 8 – *Middle Mans* do projeto JUnit.

Por meio da inspeção de código das seis classes listadas pode-se observar que as classes TestDecorator, Fail e TestName apresentaram as delegações simplistas discutidas no odor *Middle Man*. Já as classes JUnit4TestCaseFacade, IgnoredClassRunner e InitializationError foram descartadas como possíveis odores, pois continham operações que não são caracterizadas como delegações.

Cabe ressaltar que não houve uma análise de todo o projeto para verificar se as classes listadas na Tabela 8 possuem ou não algum valor arquitetural para o projeto JUnit, somente observou-se cada classe individualmente.

O código do projeto JUnit foi inspecionado e comprovou-se que o conjunto de métricas evidenciou corretamente os três *Middle Man* que existiam no projeto, portanto pode-se concluir que é um conjunto eficaz, mas que ainda depende do julgamento humano para eliminar falsos positivos .

4.3. Julgamento Humano

Em alguns casos o julgamento humano é fundamental para definir se determinada classe pode ou não ser considerado um *Middle Man*.

Na Figura 1 temos um exemplo de uma classe que, de acordo com o conjunto de métricas, é apontada como um odor.

```
public class StringProperties {

	private Properties properties;

	public String get(String key) {
		return (String) properties.get(key);
	}

	public void put(String key, String value) {
		properties.put(key, value);
	}

}
```

Figura 1 – Classe StringProperties.

Na Tabela 9 pode-se observar os valores das métricas coletados para a classe StringProperties.

Métrica	Valor
Complexidade Ciclomática	1
Linhas de Código dos Métodos	2
Número de Atributos	1
Número de Métodos	2

Tabela 9 – Métricas da classe StringProperties.

A classe StringProperties possui uma funcionalidade sutil que é garantir que Properties aceite apenas Strings, ou seja, essa classe força restrições na utilização de Properties. Este fato a descaracteriza como odor, mesmo apresentando medidas de métricas dentro dos padrões definidos na Tabela 5.

Na Figura 2 é apresentado outro exemplo de classe que é apontado como odor de acordo com os padrões definidos na Tabela 5.

```
public class ProductService {

	private ProductDAO productDAO;

	public List<Product> getAllProducts() {
		return productDAO.getAllProducts();
	}

	public void createProduct(Product product) {
		productDAO.createProduct(product);
	}

	public void updateProduct(Product product) {
		productDAO.updateProduct(product);
	}

	public void deleteProduct(Product product) {
		productDAO.deleteProduct(product);
	}

}
```

Figura 2 – Classe ProductService.

Na Tabela 10 pode-se observar os valores das métricas coletados para a classe ProductService.

Métrica	Valor
Complexidade Ciclomática	1
Linhas de Código dos Métodos	4
Número de Atributos	1
Número de Métodos	4

Tabela 10 – Métricas da classe ProductService.

A função da classe ProductService é expor os métodos de acesso e modificação da tabela Produto como um serviço para outras aplicações, ou seja, essa classe adiciona um valor arquitetural a aplicação.

Com os exemplos apresentados neste capítulo pode-se concluir que nem toda a classe de delegação simples é um odor, portanto é crucial que haja um julgamento humano sobre cada classe apontada pelo conjunto de métricas.

5. Conclusão

Este trabalho realizou um estudo sobre a possibilidade de evidenciar odores de *software* e *anti-patterns* em projetos de *software* utilizando métricas como ferramenta para busca.

Foram discutidos os aspectos objetivos das métricas de *software*, que possibilitam extrair facilmente uma medição de um código fonte, porém existe a dificuldade em evidenciar informações relevantes à partir dos número obtidos.

Os odores de *software* e *anti-patterns*, em contra partida, são características subjetivas do código fonte, sendo encontrados por meio da inspeção de código, mas apresentam um ganho real de manutenabilidade e legibidade de código quando corrigidos. Sua identificação é feita por meio da inspeção de código e é influenciada pela *expertise* do desenvolvedor, ou seja, quanto menos experiente o profissional menor será a chance de encontrar indícios de deterioração do código.

O objetivo proposto para este trabalho foi identificar um conjunto de métricas que evidenciasse a existência de odores e *anti-patterns* em um projeto, com o intuito de ajudar o desenvolvedor a inspecionar o código de forma pontual. Para atingir esse objetivo foi realizada uma prova de conceito com intuito de evidenciar o odor *Middle Man*, foram aplicados em dois projeto: o primeiro controlado, no qual a ocorrência do odor era conhecida, e outro projeto cujo código fonte era desconhecido.

O conjunto inicial de métricas coletado no projeto controlado se mostrou eficaz e identificou corretamente apenas as classes que apresentavam o odor. O mesmo conjunto de métricas foi aplicado em um projeto desconhecido, porém percebeu-se que quase metade das classes listadas não eram *Middle Mans* e não possuíam sequer delegações. O conjunto de métricas foi refinado por meio da

inspeção de código das classes listadas, listando apenas seis classes, das quais três delas eram os *Middle Mans* presentes no projeto.

Por meio dos resultados obtidos com o projeto desconhecido pode-se concluir que é possível definir conjuntos de métricas para evidenciar possíveis odores de *software* com um certo grau de eficácia, mas que ainda dependem do julgamento humano e da inspeção do código fonte para confirmar ou não a presença do odor.

A seguir são listados os possíveis trabalhos futuros que poderão ser desenvolvidos com base nesta pesquisa.

- *Plugin* para o Eclipse: criar um *plugin* que, ao clicar sobre o projeto, evidencie todas as classes que possuem métricas compatíveis com o *Middle Man*. Esse *plugin* deve ser flexível para receber novos conjuntos de métricas e valores para evidenciar outros odores e *anti*-patterns e deve permitir que o desenvolvedor abra a classe evidenciada através de um duplo clique;
- Continuação da pesquisa: neste trabalho foram listados outros 21 odores e 6 *anti-patterns* que não foram pesquisados, portanto pode-se dar continuidade ao trabalho pesquisando métricas que os evidenciem.

Referências

BOISSIER, G.; CASSELL K. **Eclipse Metrics Plugin 1.3.8 – Getting Started**. Disponível em: <http://metrics2.sourceforge.net/>. Acesso em: 17 de Junho de 2014.

BROWN, W. J. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. Wiley, 1998.

DEMARCO, T. **Controlling Software Projects: Management, Measurement and Estimation**. Yourdon Press, 1982.

FOWLER, M.; BECK, K; BRANT, J.; OPDYKE W.; ROBERTS D. **Refactoring: Inproving the Design of Existing Code**. Addison-Wesley, 1999.

MARTIN, R. C. **Agile Software Development: Principles, Patterns, and Practices**. Prentice Hall, 2002.