IDENTIFICAÇÃO DE BAD SMELLS EM SOFTWARE A PARTIR DE MODELOS UML

HENRIQUE GOMES NUNES

IDENTIFICAÇÃO DE BAD SMELLS EM SOFTWARE A PARTIR DE MODELOS UML

Proposta de dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Co-orientadora: Kecia Aline Marques Ferreira

Belo Horizonte
Novembro de 2012

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Segundo Sommerville [2007], métrica de software "é qualquer tipo de medição que se refere a um sistema de software, processo ou documentação relacionada". Historicamente, as métricas são utilizadas para estimar custo, esforço e tempo de desenvolvimento e manutenção. As métricas mais comuns são usadas para avaliar o tamanho de um software e estimar o tempo que se levará para desenvolvê-lo (Nuthakki et al. [2011]). A medição de softwares permite que uma organização melhore seu processo de desenvolvimento, auxilie no planejamento, acompanhe e controle o projeto e avalie a qualidade do software. Soliman et al. [2010] afirmam que muitas métricas de software têm sido propostas na literatura, como: linhas de códigos, porcentagem de comentários, número de módulos, contagem de elementos, abstração de dados acoplados. Ele ainda afirma que a maioria delas são aplicadas nas fases posteriores do projeto, porém que é necessário entender o projeto desde seu início.

As métricas de software também auxiliam na identificação dos desvios de projeto, conhecidos na literatura como *bad smells* Fowler [1999]. Alguns trabalhos têm sido desenvolvidos para identificar *bad smells* em software com o uso de métricas a partir de código-fonte (Marinescu [2002], Bertran [2009]). Todavia, os trabalhos realizados até então são restritos nos *bad smells* por utilizar apenas a os códigos-fontes. O trabalho de dissertação proposto nesse texto visa contribuir nesse aspecto, propondo método e ferramentas para identificação de *bad smells* em *software* orientado por objetos a partir de modelos UML, aplicando métricas de software.

A Unified Modeling Language, também conhecida como UML, é uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de softwares orientados por objetos (Fowler [2005]). Cada vez mais, com a inserção da UML desde o início dos projetos, é necessário entender como suas métricas podem ser utilizadas na prática. Ser capaz de obter métricas da UML

é importante porque ela está estritamente relacionada ao paradigma orientado por objetos, isso porque a UML fornece uma visão de toda estrutura do sistema e independe de uma linguagem de programação. Além disso, com a avaliação de modelos UML, a qualidade do software pode ser medida nos estágios iniciais, onde as alterações são mais baratas. No entanto, poucas são as discussões de como aplicar as métricas descobertas via UML (Soliman et al. [2010]).

Nesse sentido, Yi et al. [2004] cita em seu trabalho que durante os últimos anos diversos pesquisadores definiram métricas para softwares orientados por objetos e que um grupo específico de estudiosos se concentrou na definição de métricas para modelos da UML. Yi et al. [2004] ainda afirma que não há um consenso sobre as métricas a serem aplicadas nesses modelos. Alguns atribuem maior importância às métricas baseadas em métodos e atributos, outros acreditam que as classes e seus relacionamentos são os mais importantes.

Assim como nos códigos-fonte, as métricas na UML permitem analisar confiabilidade, manutenibilidade e complexidade de sistemas, porém neste caso, logo nas fases iniciais do ciclo de vida. A UML surgiu com a proposta de padronizar questões relacionadas ao projeto orientado por objetos, permitindo assim que um sistema fosse representado independente da linguagem de programação utilizada. Dentre os diagramas, destaca-se o Diagrama de Classes que representa classes de objetos e suas relações (Yi et al. [2004]). A análise desse diagrama pode nos fornecer diversas métricas relacionadas à manutenibilidade e complexidade do software logo no início do projeto. Porém, em um projeto real é inviável calcular tais métricas manualmente (Girgis et al. [2009]).

1.1 Objetivo

Diante dos problemas apresentados na Seção 1, entendemos a importância de identificar desvios de projeto logo nas fases iniciais do projeto de software. Nesse sentido, o objetivo principal de nossa proposta consiste em definir um método de identificação de *bad smells*[1] em softwares a partir de modelos UML, aplicando métricas de software. Para tal, definem-se os seguintes objetivos específicos no trabalho proposto: identificar um conjunto de métricas para detectar tais *bad smells*; definir um método e desenvolver uma ferramenta que permita automatizar a coleta destas métricas e a aplicação delas na identificação dos *bad smells* a partir de modelos UML.

[1]Desenhos incorretos e más práticas de programação.

1.2 Metodologia

Para alcançar os objetivos definidos nesse trabalho, primeiramente será realizado um levantamento de trabalhos mais recentes relacionados a utilização de métricas em modelos UML, para identificar quais os diagramas da UML são mais utilizados e quais são as métricas de software aplicáveis a modelos UML. Será realizado também um estudo sobre as ferramentas que já foram desenvolvidas e que possuem objetivos em comum com este trabalho. Em seguida, estudaremos as possíveis arquiteturas que possibilitarão o desenvolvimento da ferramenta que automatizará a identificação de anomalias, assim como quais métricas ela coletará, quais os filtros utilizaremos e quais os valores limites das métricas utilizadas serão aplicados para a identificação dos *bad smells*. A avaliação do método e da ferramenta propostos será realizada por meio de estudos de casos experimentais em softwares abertos.

1.3 Contribuições

As principais contribuições desse projeto incluem:

1. Identificar quais métricas melhor representam os aspectos relacionados a *bad smells* em modelos UML.

2. Disponibilizar uma ferramenta que dê suporte para engenheiros de software na identificação de desvios de projeto logo no início do projeto.

3. Validar a ferramenta, via um estudos de casos experimentais.

O restante desta proposta de dissertação está organizada da seguinte forma: Capítulo 2 apresenta a revisão da literatura, Capítulo 3 descreve a solução proposta e o Capítulo 4 apresenta o sumário e cronograma do trabalho a ser desenvolvido. O Capítulo 4 também conclui esta proposta.

Capítulo 2

Extração de Métricas a partir de Modelos UML

Neste capítulo serão abordados alguns trabalhos relacionados a métricas e *bad smells*, focando principalmente em modelos UML. Primeiramente, serão discutidos os conceitos básicos de métricas orientadas por objetos. Em seguida são apresentadas algumas ferramentas desenvolvidas para coleta de métricas orientadas por objetos em modelos UML. Depois será discutido o trabalho de Marinescu [2002], que é um dos primeiros trabalhos que utiliza métricas de software para identificação de *bad smells*. Por fim, é destacado o trabalho de Bertran [2009], que realizou experimentos de identificação de *bad smells* em diagramas de classes.

2.1 Métricas de Software

Sommerville [2007] define métrica como um valor numérico relacionado a algum atributo de software, permitindo a comparação entre atributos da mesma natureza. Ele argumenta que a medição nos permite realizar previsões gerais de um sistema e identificar componentes com anomalias. Em relação as métricas, o mesmo autor a define como qualquer tipo de medição que faça referência a um sistema, processo ou documento, por exemplo, a quantidade de linhas de códigos de um programa. Sommerville [2007] argumenta que por meio das métricas podemos controlar um software. Podemos também, via atributos internos de um software, definir atributos externos como facilidade de manutenção, confiabilidade, portabilidade e facilidade de uso, por exemplo.

Yi et al. [2004] fazem, em seu trabalho, uma comparação de diversas métricas para diagramas UML encontrados na literatura: métricas de Marchesi, métricas de Genero, métricas de In, métricas de Rufai e métricas de Zhou. Estas métricas, as-

sim como as métricas orientadas por objetos, tratam de métodos, atributos, herança, relacionamentos, dentre outros elementos orientados por objetos.

2.2 Ferramentas de Medição Aplicadas a Modelos UML

Na literatura há ferramentas propostas que extraem métricas de modelos da UML, na maioria dos casos, do diagrama de classes. Girgis et al. [2009] apresentam uma ferramenta capaz de calcular automaticamente diversas métricas relacionadas ao diagrama de classes. O objetivo dos autores é que a partir delas seja possível tomar decisões acerca de um projeto de software. Eles dividem as métricas utilizadas em quatro grupos: métricas de complexidade, métricas de tamanho de software, métricas de acoplamento e métrica de herança, todas extraídas de diagramas de classes. Na ferramenta construída, o sistema recebe como entrada um diagrama de classe, porém em um formato *extensible markup language*, ou XML, chamado XMI, que é a sigla de XML *Metadata Interchange*[1]. A ferramenta coleta os seguintes dados das classes do modelo UML, nome, visibilidade e para cada classe as seguintes informações são coletadas: atributos (nome, tipo e visibilidade), métodos (nome, tipo, parâmetros e visibilidade) e relacionamentos (quantidades de agregações, composições, associações e heranças). As métricas são geradas a partir desses dados e os resultados são exibidos para o usuário. Para validar o trabalho, os autores apresentam um estudo de caso em que são coletadas métricas de três versões de um sistema em Java feito em código aberto. Os autores apresentam as alterações do software em dados quantitativos e concluem que o comportamento das métricas permite avaliar a evolução do software em diversos aspectos: avaliação da qualidade do projeto, melhor visualização do sistema, detecção de falhas dos padrões de projetos e detecção da necessidade de refatorações.

Girgis et al. [2009] detalham bem como seu sistema foi construído tecnicamente e quais as fontes das métricas utilizadas, porém a proposta possui o problema de que apenas as variações das métricas são apresentadas, cabendo ao usuário definir manualmente se esta variação é um problema ou não para o seu projeto.

Soliman et al. [2010] apresentam uma ferramenta que coleta seis métricas CK. Essas métricas são extraídas de diagramas de classe, diagramas de sequência e diagramas de atividades, que são representados em arquivos do tipo XMI. A justificação dos auto-

[1] A justificação do autor para utilizar este formato é que este é o padrão adotado pela Object Management Group, também conhecida como OMG (OMG [2012]), a maior organização internacional que aprova padrões abertos para aplicações orientadas por objetos.

res para escolherem as métricas CK é de que além destas cobrirem todos os aspectos de modelos orientados por objetos, elas são amplamente aceitas e são adotadas pela *Object Constraint Language*, ou OCL (Fowler [2005]), linguagem que define parte do padrão da UML. Para validar a ferramenta desenvolvida, os autores realizam dois estudos de caso. No primeiro caso, eles escolhem um modelo de um software fictício de matrícula em cursos *online*, fornecido no sítio da Microsoft, que disponibiliza os diagramas UML com o objetivo de fazer uma pequena demonstração desta linguagem. Os três diagramas utilizados foram: diagrama de classes, diagrama de sequência e diagrama de atividades. Os autores coletam as seis métricas de cada classe e fornecem o valor médio de cada uma delas, comparando com o maior e menor valor obtido para cada métrica. O segundo estudo de caso é realizado em um sistema de código aberto chamado Midas, que é apresentado no próprio trabalho de Soliman et al. [2010]. Neste caso, os diagramas não são fornecidos pelo criador do sistema. Então, os autores utilizam a ferramenta de modelagem Argo UML (ArgoUML [2012]) e realizam uma engenharia reversa no código-fonte para obter os diagramas. A análise ocorre da mesma forma feita no estudo anterior.

Soliman et al. [2010] não aprofundam muito na análise dos resultados dos dois estudos, apenas concluem se determinada métrica está alta ou baixa para cada caso, baseando-se nas médias. Algumas sugestões para trabalhos futuros são apresentadas como a análise de sistemas como um todo, visto que no trabalho proposto por eles as classes são analisadas individualmente, e, definição de valores limites para verificar se as métricas dos modelos UML estão dentro de valores aceitáveis.

O trabalho de coletar métricas feito por Nuthakki et al. [2011] se diferencia dos outros trabalhos, porque a proposta é a de criar uma ferramenta capaz de extrair métricas em diferentes formatos de arquivos gerados pelas ferramentas de modelagem. Para isso, uma ferramenta denominada UXSOM é desenvolvida pelos autores. Esta ferramenta suporta diagramas de classes exportados para os formatos XML, UXF (UML exchange format) e XMI. A ferramenta proposta utiliza os arquivos de representação de diagramas gerados pelas seguintes ferramentas: ArgoUML, UMLet, ESS-Model, MagicDraw UML, Sparx System Enterprise Archtect. Para cada arquivo, são extraídas as seguintes informações: formato do arquivo do diagrama, quantidade de métodos totais e públicos, quantidade de atributos totais e públicos, quantidade de associações, quantidade de filhos diretos, quantidade de classificadores e pacotes e quantidade de relações entre classificadores e pacotes. O mesmo diagrama de classes é gerado para cada ferramenta e ele é exportado para algum dos formatos citados - XML, UXF ou XMI. Os autores então comparam os valores obtidos e observam, que apesar de algumas diferenças, os valores são bem próximos, concluindo então que a ferramenta cumpriu

perfeitamente os objetivos definidos.

Nuthakki et al. [2011] mostram que existe diferença na utilização de uma ferramenta para outra, inclusive aquelas que utilizam um mesmo formato de arquivo. A pesquisa realizada, porém, não explora as métricas coletadas para avaliar a qualidade do software ou identificar possíveis desvios de projeto.

As ferramentas apresentadas nesses três trababalhos apenas coletam métricas. Essas métricas não fornecem informaçãos relavantes relacionadas a questões de qualidade de software, como, por exemplo, a facilidade de manutenção de um sistema. Considerando tal problema, o trabalho proposto para essa dissertação tem o objetivo de desenvolver uma ferramenta que permita identificar desvios de projeto de software, utilizando métricas coletadas a partir de modelos UML.

2.3 Detecção de *Bad Smell* em Software

O processo de concepção de softwares está sujeito a erros. Falhas na estrutura de projetos orientados por objetos têm um forte impacto negativo nos atributos relacionados a qualidade, tais como a flexibilidade ou capacidade de manutenção. Estas falhas são conhecidas como *bad smells* e é de extrema importância sua descoberta desde o início do projeto.

Bertran [2009] afirma que para existir um controle de qualidade efetivo, é necessário uma quantificação adequada, questionando se "é possível expressar boas regras de um projeto de um modo quantificável ?"

As métricas de softwares são poderosos mecanismos para avaliar e controlar qualidade do projeto de software, porém, elas apresentam algumas limitações, como por exemplo, a difícil interpretação dos dados que ela gera e a indicação de quais pontos específicos de um software geraram anomalias nos resultados das métricas.

Alguns pesquisadores propõem mecanismos de estratégias de detecção de desvios de projetos em software orientado por objetos. Eles se propõem a detectar e localizar problemas de projeto antes da implementação de um sistema. A estratégia de detecção é uma condição lógica composta por métricas e pode localizar classes e métodos problemáticos, em vez de apenas registrar os valores das métricas. Bertran [2009] definiu em seu trabalho *bad smell* como "desenhos incorretos e más práticas de programação".

Algumas das principais referências para detecção de *Bad-smell* em software orientado por objetos são os trabalhos de Marinescu [2002] e Marinescu [2004], cujo objetivo é desenvolver métodos e técnicas que forneçam informações relevantes, qualitativa e quantitativamente, resultante da coleta de métricas de códigos orientados por objetos.

O autor utilizou estratégias de detecção (*detection strategies*). Ele então desenvolveu uma ferramenta que coleta métricas e as aplicam em fórmulas relacionadas a estratégias de detecção e gera informações relevantes relacionadas ao código-fonte. Estas informações apontam as falhas de especificação do projeto. Ele dividiu as falhas em quatro grupos: análise, mutabilidade, estabilidade e testabilidade. O autor realizou um estudo de caso no qual, para cada estratégia destes quatro grupos foram definidos valores limites, e para o modelo de um sistema o autor avaliou os resultados obtidos. Este trabalho é um dos pioneiros sobre detecção de *bad smells* em softwares orientados por objetos.

A identificação e utilização de métricas para softwares orientados por objetos é muito incentivada, porém há alguns problemas na aplicação prática de métricas de software, por exemplo:

- A definição de métricas são algumas vezes imprecisas ou incompletas.
- A interpretação dos resultados não é muito precisa, pouco empírica, o que gera pouca confiabilidade na aplicação das métricas.
- Uma métrica isolada pode apontar uma anomalia no software, mas nem sempre indica a causa da anomalia.

Visando contornar esses problemas, o objetivo dos trabalhos de Marinescu [2002] e Marinescu [2004] se resumem em propor mecanismos que permitam, por meio de estratégias de detecção, definir resultados mais palpáveis extraídos das métricas coletadas. A seguir são apresentadas as estratégias utilizadas por Marinescu [2004] para identificar um *Bad Smell*.

2.3.1 Estratégias de Detecção

Esta seção apresenta as estratégias de detecção de *bad smell* definidas por Marinescu [2002] e Marinescu [2004]. Estratégia de detecção nesse contexto é definida como "uma expressão quantificável de uma regra, aonde podemos verificar se fragmentos do código-fonte estão em conformidade com esta regra definida"(Bertran [2009]). Serão abordados os seguintes aspectos sobre essas estratégias: mecanismos de filtragem, mecanismos de composição, definição da estratégia para definir um *bad smell*.

2.3.1.1 Mecanismos de Filtragem de Métricas

A filtragem possibilita a redução do conjunto inicial de dados, de modo que se possa verificar uma característica especial de fragmentos que têm suas métricas capturadas.

Os filtros permitem definir limites de valores para determinados aspectos. Na Figura 2.1, Marinescu [2004] define tipos de filtros e os divide em dois grupos:

- Filtro marginal: é definida uma margem que divide os valores do conjunto de dados. Esse pode ser dividido em dois sub-filtros denominados semânticos e estáticos.

- Filtro de intervalo: é definido um limite inferior e outro superior para o conjunto de dados.

Na Figura 2.1 os filtros são detalhados.

Type of Data Filter	Limit Specifiers		Filter Example
Marginal	Semantical	*Relative*	▪ `TopValues(10)`, ▪ `BottomValues(5%)`
		Absolute	▪ `HigherThan(20)` ▪ `LowerThan(6)`
	Statistical		▪ `Box-Plot`
Type of Data Filter	**Specification**		**Filter Example**
Interval	Composition of two marginal filters, with semantical limit specifiers of opposite polarities		`Between(20,30) := HigherThan(20) ∧ LowerThan(30)`

Figura 2.1. Filtros de dados. Fonte: Marinescu [2004]

Marinescu [2002] e Marinescu [2004] apresentam um conjunto de regras que devem orientar o engenheiro para decidir qual tipo de filtro de dados deve ser aplicado sobre os resultados de uma métrica particular:

- Regra 1: escolher um filtro absoluto quando for quantificar regras de projeto que especificam limite explicitamente concreto dos valores.

- Regra 2: escolher um filtro semântico quando a regra do projeto for definida em termos de valores difusos marginais, como "acima ou abaixo de valores ou em valores extremos".

- Regra 3: para grandes sistemas, utilizar os filtros do tipo marginal, mais especificamente, os semânticos. Para sistemas menores, utilizar intervalos.

- Regra 4: escolher de um filtro estatístico para casos em que as regras de projeto façam referência a valores extremamente alto / baixo, sem especificar qualquer limite preciso.

2.3.2 Mecanismos de Composição

Os mecanismos de composição permitem utilizar as métricas em operações matemáticas, possibilitando assim a mescla de métricas para obter informações relevantes. Marinescu [2002] e Marinescu [2004] definem dois tipos de mecanismos de composição:

- ponto de vista lógico: permite aplicar operadores lógicos às métricas
- ponto de vista de conjuntos: permite agrupar as métricas em conjuntos

Marinescu [2002] e Marinescu [2004] utilizam do *bad smell* denominado *God Class*[2], para exemplificar o processo de formulação das estratégias de detecção de uma falha de projeto concreto. O ponto de partida é dado por uma ou mais regras informais. Na Seção 2.3.2.1, a definição destas regras será mais detalhada.

2.3.2.1 Definição de Estratégias de Detecção de *Bad Smell*

Primeiramente, as regras definidas para *God Class* tratam de encontrar uma classe com alta complexidade, baixa coesão e um alto nível de acoplamento. No trabalho de Marinescu [2002] e Marinescu [2004], foram usadas as seguintes métricas, extraídas de Lanza et al. [2005], para identificar God Class:

- *Weighted Method Count* (WMC): soma de complexidade dos métodos.
- *Tight Class Cohesion* (TCC): é o número de métodos conectados diretamente, ou seja, conta o número relativo de métodos de uma classe que acessam pelo menos um atributo em comum.
- *Access to Foreign Data* (ATFD): número de acessos a atributos de classes externas.

O próximo passo é definir o mecanismo de filtragem. Neste caso o autor definiu TopValues(25 porcento) para algumas métricas e HigherThan(1) para outras. Após isso, esses elementos devem ser relacionados utilizando os mecanismos de composição.

[2]Problema que ocorre quando uma classe centraliza grande parte das responsabilidades de um sistema.

2.4 Avaliação da Qualidade de Software com Base em Modelos UML

A maioria dos trabalhos desenvolvidos até hoje estão relacionados à detecção de anomalias extraídas exclusivamente do código-fonte. Embora detectar desvios de projeto em código fonte seja útil para evitar a degradação do software, é necessário também que existam meios para se identificar problemas desde o início do ciclo de vida do sistema.

Nesse sentido, o trabalho de Bertran [2009] define estratégias de detecção que podem ser aplicadas a diagramas UML. O propósito do trabalho é definir técnicas que encontrem os mesmos elementos problemáticos do projeto, quando extraídos do código fonte, mas que sejam aplicado a diagrama de classes. Nesse trabalho foram considerados seis *bad smells*:

- *God Class*: identifica classes com alta responsabilidade no sistema.
- *Data Class*: identifica classes que não definem funcionalidade própria.
- *Long Parameter List*: identifica métodos com uma longa lista de parâmetros.
- *Shotgun Surgery*: identifica classes que ao serem alteradas causam impacto em outras classes.
- *Misplaced Class*: identifica classes que se relacionam pouco com classes de seu pacote, porém se relacionam muito com classes de outros pacotes.
- *God Package*: identifica pacotes grandes, que possuem muitos dependentes.

Foi desenvolvida uma ferramenta com o objetivo de automatizar o controle de qualidade em modelos UML, ou seja, a aplicação automatizada das estratégias propostas e dos modelos de qualidade. Tal ferramenta foi utilizada em dois estudos experimentais. O primeiro estudo tem por objetivo verificar se os valores das métricas obtidos em diagramas de classes são correspondentes àqueles obtidos em código fonte. Para isso, foram extraídos dados do código fonte e do diagrama de classes de 9 sistemas, e os valores obtidos foram comparados. O segundo estudo propõe a aplicação do modelo de qualidade QMOOD em uma sequencia de versões do software JHotdraw com o objetivo de avaliar a mudança das métricas internas e externas de um sistema ao longo de sua vida, utilizando os diagramas de classe para extrair as métricas. Dada a importância do trabalho de Bertran [2009] em relação ao tema de pesquisa proposto para essa proposta de dissertação, a Seção 2.4.1 o apresenta em mais detalhes.

2.4.1 O Modelo de Qualidade QMOOD

A qualidade de um software deve ser definida em termos de atributos de produtos de software que são de interesse do usuário. Utilizar os atributos internos de um software a fim de predizer os atributos externos que são definidos pelo usuário é uma tarefa fundamental. Por causa disso, engenheiros de software definiram na literatura modelos de qualidade para a estimativa dos atributos externos em termos de atributos internos (por Bertran [2009]).

Bertran [2009] utilizou o modelo de qualidade QMOOD (*Quality Model for Object-Oriented Design*), que propõe estimar os atributos de qualidade que são de interesse para os usuários. Esse modelo tem um formato hierárquico, no qual os níveis superiores fazem referências aos atributos externos de qualidade de um software, como flexibilidade e facilidade de uso e os atributos externos são compostos por um ou mais atributos internos de um software. Sendo assim, o QMOOD é decomposto em quatro níveis como mostra a Figura 2.2.

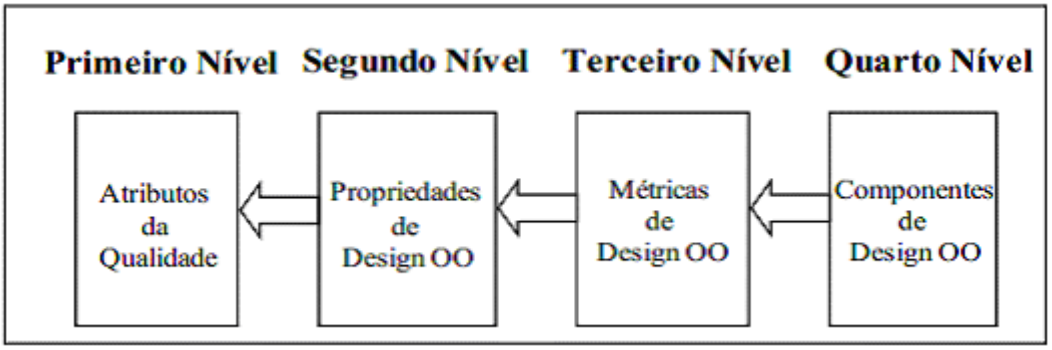

Figura 2.2. Níveis do modelo de qualidade QMOOD. Fonte: Bertran [2009]

Os atributos de qualidade estão relacionados a atributos externos de qualidade, que se baseiam em um conjunto definido pela norma ISO9126. Os atributos utilizados no QMOOD foram: funcionalidade, eficácia, facilidade de compreensão, facilidade de extensão, usabilidade e flexibilidade.

As propriedades do projeto orientado por objetos são avaliadas examinando as classes, métodos e atributos de um modelo, ou seja, são atributos internos. Abordam tamanho, hierarquia, coesão, abstração, acoplamento, herança, polimorfismo, composição e troca de mensagens entre classes.

As métricas de projeto Orientadas por Objetos utilizadas no trabalho de Bertran [2009] são mostradas na Figura 2.3.

Os componentes de projeto Orientados por Objetos são os elementos do diagrama: classes, atributos, métodos, relações como composição e herança. Os componentes

Métrica	Nome	Descrição
DSC	Número de classes no *design*	Conta o número total de classes no *design*
NOH	Numero de Hierarquias	Conta a quantidade de hierarquias de classes no *design*
ANA	Média de Ancestrais	Representa a média de classes das quais a classe avaliada herda informação. É calculado utilizando o número total de classes na estrutura de herança.
DAM	Métrica de acesso a dados	Calcula a proporção que representam os atributos privados (protegidos) do total de atributos da classe (faixa 0-1)
DCC	Acoplamento direto entre classes	Conta a quantidade de classes com as quais a classe avaliada está direitamente relacionada. Na contagem, são incluídas as declarações de atributos e parâmetros.
CAM	Coesão entre os métodos da classe	Representa o grau de relação entre os métodos de uma classe baseada nos parâmetros dos métodos da classe. Primeiramente é calculado o conjunto máximo independente dos tipos dos parâmetros de todos os métodos. Depois é computada a soma da interseção dos tipos dos parâmetros de cada um dos métodos com o conjunto independente. Finalmente, a soma é divida pelo número total de parâmetros multiplicado pela quantidade de métodos.
MOA	Medida de agregação	Representa a composição de uma classe utilizando os atributos declarados. É computada contando o número de atributos cujo tipo é definido pelo usuário.
MFA	Abstração funcional	Calcula a proporção que representam os métodos herdados do total de métodos da classe (faixa 0-1)
NOP	Numero de métodos polimórficos	Conta a quantidade de métodos que podem ter comportamento polimórfico (e.x. em Java, os métodos não declarados como final).
CIS	Tamanho da interface de uma classe	Conta a quantidade de métodos públicos que a classe possui. Não são considerados nesta métrica os atributos públicos.
NOM	Quantidade de métodos	Conta a quantidade de métodos declarados na classe.

Figura 2.3. Métricas utilizadas no modelo QMOOD. Fonte: Bertran [2009]

de projeto são os responsáveis por fornecer informações para gerar as métricas do modelo. Por sua vez, o conjunto de métricas forma as propriedades de projeto, como por exemplo, os valores de acoplamento e coesão de um modelo. Já as propriedades,

quando aplicadas em fórmulas, geram os valores dos atributos de qualidade, ou seja, os atributos externos de um modelo. Um exemplo de uma fórmula de uma qualidade externa utilizada no trabalho é:

Facilidade de extensão $= 0,25 * coesao - 0,25 * acoplamento + 0,25 * trocademensagens + 0,5 * tamanho$

2.4.2 Estratégias para a Detecção de Problemas de Desenho

No trabalho de Bertran [2009] o modelo QMOOD não foi integrado ao mecanismo de detecção de *bad smells*, mas a intenção de trabalho futuro da autora é de que utilizando os dois mecanismos, se terá a estimativa dos atributos externos da qualidade e a detecção/localização de possíveis entidades do projeto causadora dos valores não desejados dos atributos estimados.

A aplicação de estratégicas de detecção possui problemas. O primeiro deles é que tal técnica não garante que o problema identificado realmente exista e que o projeto precise ser alterado. Portanto, mesmo que as estratégias gerem alerta, o projetista deve analisar o projeto para verificar se a notificação realmente é um problema no projeto.

Ferramentas como InCode [2012] fazem a detecção automática de *bad smells* como *Data Class*, *God Class*, *Feature Envy* e *Duplication Code* extraindo informações direto do código fonte. Valores gerados por essa ferramenta foram utilizados como parâmetro para comparação dos resultados obtidos por Bertran [2009].

2.4.3 Estudos Experimentais

A ferramenta QCDTool desenvolvida por Bertran [2009], é utilizada para realização dos estudos experimentais. A ferramenta é responsável por realizar a coleta de dados para os seis *bad smells* considerados no trabalho e por aplicar o modelo QMOOD, ambos para diagramas de classe.

No primeiro estudo a ferramenta QCDTool buscou avaliar se os problemas de projeto encontrados em códigos fonte são os mesmos identificados em seu diagrama de classes. O objetivo foi avaliar a acurácia, precisão e o *recall* entre os valores obtidos para código fonte e o diagrama para os seis *bad smells*. Para isso, foram utilizados nove sistemas, sendo alguns softwares de código livre e outros sistemas desenvolvidos em trabalhos de cursos acadêmicos. Todos os sistemas foram desenvolvidos na linguagem Java.

Os resultados do estudo mostraram que as estratégias de detecção dos *bad smells Data Class*, *God Class* e *God Package* estavam correlacionadas significativamente com

suas correspondentes versões para o código. As estratégias Long Parameter List e Data Class apresentaram 100% nos graus de *recall*. A God Package apresentou 100% de precisão, o que indica que a Shotgun Surgery apresentou alto grau de similaridade dos valores entre código fonte e o modelo. Isto significa que todas as entidades que tiveram estes seis problemas de projeto no código fonte foram detectadas pela estratégia proposta para o modelo. Porém, algumas das entidades identificadas como problemáticas no modelo não foram identificadas como tal no código.

O segundo estudo teve como objetivo avaliar a usabilidade do modelo de qualidade QMOOD em diagramas de classe. Este modelo foi aplicado a uma sequência de versões do framework JHotDraw [2012], foram avaliadas ao total 4 versões do aplicativo. Os atributos de qualidade externos avaliados dos modelos foram: facilidade de compreensão, flexibilidade e reusabilidade. A avaliação de estudo consistiu em avaliar valores manualmente de uma versão para outra e verificar se a alteração desses valores correspondem a diferenças significativas na estrutua do software. As métricas consideradas avaliam os seguintes aspectos: abstração, acoplamento, coesão, complexidade, composição, encapsulamento, polimorfismo, tamanho e troca de mensagens.

Os resultados da aplicação do modelo QMOOD nos diagramas de classe mostraram que os valores das métricas que avaliam os atributos de qualidade reusabilidade e flexibilidade aumentaram ao longo das quatro versões. Isto aconteceu pelo aumento das funcionalidades ao longo das versões do software. Além disso, o atributo da qualidade facilidade de compreensão degradou, isto foi motivado pelo fato que novas funcionalidades e características foram inseridas para aumentar a capacidade do sistema por meio da implementação de novas classes e métodos a classes já existentes.

Bertran [2009] realizou os dois experimentos, mas não relacionou diretamente o estudo do modelo de qualidade QMOOD com os resultados dos *bad smells*. Seu trabalho realizou dois estudos separados, evidenciado uma questão em aberto que é verificar a viabilidade de relacionar um modelo de qualidade com a detecção de *bad smells*.

O trabalho de Bertran [2009] está relacionado diretamento ao trabalho que será desenvolvido na dissertação de mestrado proposta. As principais contribuições do trabalho de Bertran [2009] são:

1. Definição de técnicas para detecção de 6 *bad smells* utilizando diagramas de classe.

2. Criação de uma ferramenta para automatizar a análise da qualidade de métricas, detecção de problemas de projeto e modelos de qualidade em diagramas UML.

3. Validação da ferramenta QCDTool, através da comparação dos resultados obtidos de diagramas UML e código-fonte.

2.5 Conclusão

Os trabalhos de Girgis et al. [2009], Soliman et al. [2010] e Nuthakki et al. [2011] discutem acerca de quais métricas são extraídas a paritr de modelos UML. Por meio do estudo deles, fica claro que o diagrama de classes é o modelo predominante quando o foco é medição. Esses trabalhos, também apresentam um modelo de arquivo denominado XMI, que é o responsável por fornecer todas as métricas do diagrama de classes. Porém, nenhum desses trabalhos relacionam as métricas com atributos externos de um sofware, ou seja, não nos fornece informações relevantes acerca do projeto a ser desenvolvido.

Os trabalhos de Marinescu [2002] e Marinescu [2004] apresentam uma solução para transformar métricas em informações relevantes para detectar falhas de projetos. Essas falhas são denominadas *bad smells*. Porém, esses trabalhos lidam apenas com informações coletadas em códigos-fonte, que passa a existir apenas no fim do projeto.

O trabalho de Bertran [2009] é o mais próximo do trabalho proposto nesse texto. Nele, a autora associa métricas extraídas de diagramas de classes a *bad smells*. Para automatizar tal tarefa, a autora desenvolveu uma ferramenta para coleta desses dados. Contudo, os coeficientes foram definidos arbitrariamente, não havendo uma justificativa do porque desses valores serem adotados, senão simplesmente o contexto do problema. O QMOOD também possui algumas limitações. Ela não se preocupa em apontar os elementos do projeto responsáveis por valores ruins eventualmente obtidos para os atributos. O desenvolvedor é informado apenas do problema, devendo diagnosticar a verdadeira causa por conta própria. Portanto, um dos pontos falhos do trabalho de Bertran [2009] é a discussão limitada a respeito de valores limites para cada *bad smell*, sendo esse um ponto importante a se observar nessa proposta.

Além desses pontos negativos observados, a própria autora, cita algumas limitações de seu trabalho:

1. A utilização de poucos sistemas para validar os resultados.

2. A utilização de apenas o diagrama de classes da UML.

3. A ausência de outros *bad smells* existentes.

4. A realização de um estudo de caso real, antes do sistema ser implementado. Pois, Bertran [2009] utilizou diagramas de classes criados via engenharia reversa de sistemas já implementados.

5. A limitação do QCDTool, por não exibir estatísticas e diversos formatos de exportação.

6. A não realização de estudos para fins didáticos da ferramenta QCDTool.

Capítulo 3

Identificação de Bad Smells em Software a Partir de Modelos UML

Este capítulo apresenta o problema identificado para essa proposta de dissertação de mestrado e a metodologia que será utilizada para solucioná-lo.

3.1 O Problema

Os problemas de projeto de software surgem desde a concepção deste e, à medida em que sua evolução ocorre, as soluções para estes problemas tornam-se cada vez mais caras, visto que as pequenas mudanças podem causar grandes impactos em todo o sistema. Descobrir anomalias em um sistema finalizado é importante, pois sua manutenção é necessária, porém em alguns casos, pode-se tornar inviável.

Sendo assim, este trabalho propõe definir um método baseado em métricas de software para identificar falhas de estrutura de um projeto logo em suas fases inicias, permitindo que questões como flexibilidade e facilidade de manutenção sejam trabalhadas antes mesmo da programação do software iniciar, mais especificamente, na fase de desenho do software. Serão desenvolvidas as ferramentas para aplicação do método proposto, bem como serão realizados estudos experimentais para avaliar o método proposto.

3.2 Metodologia para a Solução Proposta

O trabalho proposto será baseado na utilização de métricas de modelos UML para identificar desvios de projeto em software orientado por objetos. Portanto, o primeiro

passo é identificar quais diagramas da UML são relevantes na construção de um sistema e quais as métricas que podem ser extraídas a partir desses.

As métricas são atributos internos de um sistema, elas por si só não dizem ao engenheiro de software nada a respeito do sistema que esta sendo construído. Portanto, o segundo passo é transformar esses atributos internos, ou seja as métricas, em atributos externos como, por exemplo, a facilidade de manutenção de um sistema. Para que isso seja possível, utilizaremos as técnicas denominadas por Marinescu [2002] como *bad smells*, "designs incorretos e más práticas de programação". Os *bad smells* nos fornecem valores referência das classes de um sistema, o que nos permitirá avaliar se um determinado atributo externo está acima ou abaixo do esperado.

Estes dados são importantes para os engenheiros de software que são responsáveis por elaborar os diagramas do sistema a serem construídos, mas as tarefas de coleta de métricas e sua transformação em atributos externos não podem consumir muito tempo do projeto, pois senão elas tornariam inviáveis suas utilizações. Sendo assim, este trabalho propõe, por meio da criação de uma ferramenta, automatizar a coleta das métricas e sua transformação em atributos externos, assim como fornecer alertas sobre possíveis anomalias no projeto determinando valores limites para cada atributo.

3.3 Avaliação

Para avaliar o trabalho utilizaremos ferramentas já consolidadas como o InCode para obter valores de *bad smells* de sistemas já construídos, ou seja, de seu código-fonte. Em seguida realizaremos uma engenharia reversa desses sistemas para obter os diagramas da UML. Nossa ferramenta irá coletar dados desses diagramas. Então compararemos nossos resultados com os obtidos do código-fonte para verificar a confiabilidade de nossos dados. Também, avaliaremos manualmente, classe por classe se os alertas para cada *bad smell* emitido por nossa ferramenta é verdadeiro ou falso.

Capítulo 4

Desenvolvimento do Trabalho

Nesse capítulo é descrito o que se espera com o desenvolvimento dessa dissertação. Para isso, o organizamos da seguinte forma: Seção 4.1 delineia uma proposta de sumário para a dissertação, Seção 4.2 apresenta o cronograma de trabalho baseado no sumário proposto, e Seção 4.3 mostra as contribuições esperadas.

4.1 Sumário da Dissertação

1. Introdução

    a) Contextualização

    b) Motivação

    c) Objetivo

    d) Problema a ser resolvido

    e) Contribuição

    f) Estrutura da Dissertação

2. Métricas de Software

    a) Métricas de Software

    b) Métricas de Software Orientadas por Objetos

    c) Métricas de Software Orientadas por Objetos de Modelos UML

    d) Ferramentas de Medição Aplicadas a Modelos UML

    e) Conclusão

3. Bad Smells
    a) Atributos Externos de Qualidade de Software
    b) Detecção de *Bad Smell* em Software
    c) Avaliação da Qualidade de Software com Base em Modelos UML
    d) Conclusão
4. Ferramenta de Coleta de Bad Smells em Modelos UML
    a) Especificação da Ferramenta
    b) Casos de Uso
    c) Interface do usuário
    d) Arquitetura
    e) Implementação
    f) Conclusão
5. Estudo Experimental
    a) Planejamento do Estudo Experimental
    b) Escolha e Adaptação das Métricas e *Bad Smells*
    c) Sistemas Avaliados
    d) Coleta dos Dados do Código-Fonte
    e) Coleta dos Dados do Modelo UML
    f) Conclusão
6. Análise dos Resultados
    a) Análise Comparativa: Código-Fonte x UML
    b) Análise Manual dos Resultados
    c) Conclusão
7. Limitações do Trabalho
8. Conclusão e Trabalhos Futuros

4.2 Cronograma

Cronologicamente, como estão nas Figuras 4.1, 4.2 e 4.3, a organização dese trabalho ocorrerá da seguinte forma: entre os meses de Novembro até o fim de Janeiro será dada continuidade ao trabalho já iniciado nessa proposta, será refinada a introdução e o referencial teórico. Entre Fevereiro até o fim de Junho, a ferramenta proposta será proposta e implementada. De Julho a Outubro os experimentos serão realizados. De Novembro de 2013 a Fevereiro de 2014 serão analisados os resultados, avaliadas as limitações, elaboradas as conclusões e durante todo esse tempo. Paralelamente, dois artigos, serão escritos e submetidos para conferências da área.

Atividade	nov/12	dez/12	jan/13
1. Introdução			
a) Contextualização			
b) Motivação			
c) Objetivo			
d) Contribuição			
e) Estrutura da Dissertação			
2. Métricas de Software			
a) Métricas de Software			
b) Métricas de Software Orientadas por Objetos			
c) Métricas de Software Orientadas por Objetos de Modelos UML			
d) Ferramentas de Medição Aplicadas a Modelos UML			
e) Conclusão			
3. Bad Smells			
a) Atributos Externos de Qualidade de Software			
b) Detecção de Bad Smell em Software			
c) Avaliação da Qualidade de Software com Base em Modelos UML			
d) Conclusão			

Figura 4.1. Cronograma do desenvolvimento da dissertação - parte 1

Atividade	fev/13	mar/13	abr/13	mai/13	jun/13	jul/13	ago/13	set/13	out/13
4. Ferramenta de Coleta de Bad Smells em Modelos UML									
a) Especificação da Ferramenta									
b) Casos de Uso									
c) Interface do usuário									
d) Arquitetura									
e) Implementação									
5. Estudo Experimental									
a) Planejamento do Estudo Experimental									
b) Escolha e Adaptação das Métricas e Bad Smells									
c) Definição dos Valores Limites para os Bad Smells									
d) Sistemas Avaliados									
e) Coleta dos Dados do Código-Fonte									
f) Coleta dos Dados do Modelo UML									

Figura 4.2. Cronograma do desenvolvimento da dissertação - parte 2

Atividade	nov/13	dez/13	jan/14	fev/14
6. Análise dos Resultados				
a) Análise Comparativa: Código-Fonte x UML				
b) Análise Manual dos Resultados				
c) Conclusão				
7. Limitações do Trabalho				
8. Conclusão e Trabalhos Futuros				
9. Escrita dos artigos				

Figura 4.3. Cronograma do desenvolvimento da dissertação - parte 3

4.3 Contribuições

No fim desse trabalho esperamos contribuir primeiramente identificando um conjunto de métricas que melhor representem os *bad smells* que iremos coletar e de quais diagramas UML estes podem ser extraídos. Essas métricas orientadas por objetos sofrerão ajustes em sua forma de coleta, pois primeiramente sua extração foi pensada em código-fonte. Outra contribuição será a disponibilização de uma ferramenta que dê suporte, de forma automatizada, para engenheiros de software na identificação de desvios de projeto.

A ferramenta construída será avaliada experimentalmente, de tal forma que futuros usuários possam entender seu funcionamento e suas diferenças se comparada com ferramentas que realizam este trabalho em códigos-fonte.

Temos a intenção de publicar pelo menos dois artigos em conferências de ponta na àrea de nossa pesquisa. Esses artigos serão escritos e submetidos durante o desenvolvimento desse trabalho de dissertação.

Referências Bibliográficas

ArgoUML (2012). http://argouml.tigris.org/, acessado em 25/11/2012.

Bertran, I. M. (2009). Avaliação da qualidade de software com base em modelos uml.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2005). Uml essencial: Um breve guia para a linguagem-padrÃ£o de modelagem de objetos. 3a ed.

Girgis, M.; Mahmoud, T. & Nour, R. (2009). Uml class diagram metrics tool. In *Computer Engineering Systems, 2009. ICCES 2009. International Conference on*, pp. 423 –428.

InCode (2012). http://www.intooitus.com/products/incode, acessado em 25/11/2012.

JHotDraw (2012). http://www.jhotdraw.org/, acessado em 25/11/2012.

Lanza, M.; Marinescu, R. & Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Marinescu, R. (2002). In *Measurement and Quality in Object-Oriented Design*.

Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 350 – 359.

Nuthakki, M. K.; Mete, M.; Varol, C. & Suh, S. C. (2011). Uxsom: Uml generated xml to software metrics. *SIGSOFT Softw. Eng. Notes*, 36(3):1--6.

OMG (2012). http://www.omg.org/, acessado em 25/11/2012.

Soliman, T.; El-Swesy, A. & Ahmed, S. (2010). Utilizing ck metrics suite to uml models: A case study of microarray midas software. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pp. 1 –6.

Sommerville, I. (2007). Engenharia de software 8a ed.

Yi, T.; Wu, F. & Gan, C. (2004). A comparison of metrics for uml class diagrams. *SIGSOFT Softw. Eng. Notes*, 29(5):1--6.