CENTRO UNIVERSITÁRIO UNA

DIRETORIA DE EDUCAÇÃO CONTINUADA, PESQUISA E EXTENSÃO

CURSO DE PÓS GRADUAÇÃO EM ENGENHARIA DE SOFTWARE
COM ÊNFASE EM MÉTODOS ÁGEIS

IDENTIFICANDO OPORTUNIDADES DE REFATORAÇÃO ATRAVÉS DE MÉTRICAS

ALUNO: Alexandre Aquiles Sipriano da Silva

PROFESSOR ORIENTADOR: Edgard Davidson Costa Cardoso

BELO HORIZONTE

2011 / 01

Identificando Oportunidades de Refatoração através de Métricas

Resumo

A prática de refatoração é uma maneira efetiva de realizar a melhoria contínua do código fonte de um software, de maneira a permitir que mudanças nas necessidades de negócio dos clientes sejam implementadas com menos esforço. Porém, a detecção de qual código deve ser refatorado depende muitas vezes da intuição de especialistas. Este artigo estuda o uso de métricas de código fonte na identificação de oportunidades de refatoração, de maneira a detectar pontos de melhoria de maneira quantitativa. Foi realizado um estudo de caso de dois projetos comerciais em que métricas de código fonte foram levantadas e analisadas, de maneira a identificar pontos de melhoria e relacioná-los a possíveis refatorações. Os resultados mostram o uso de métricas como uma ferramenta interessante para identificação de possíveis refatorações em uma base de código. Porém, como uma mesma métrica pode estar relacionada a várias refatorações, a decisão de qual refatoração deve ser efetuada ainda depende de intuição.

Palavras-chave: Refatoração; Métricas; Design de Software

1 Introdução

Uma das premissas básicas das metodologias ágeis de software é a aceitação de mudanças tardias. Isso traz vantagens competitivas aos patrocinadores dos projetos e é uma condição para atender às alterações das necessidades de negócio dos clientes.

Em termos arquiteturais, no entanto, pode ser difícil acompanhar essas mudanças. É preciso que haja um esforço de melhoria contínua do design e da arquitetura do software, de modo a conter o custo das mudanças.

Esse esforço de melhoria contínua pode ser realizado por meio de refatorações. Fowler *et al.* (1999) definem uma refatoração como uma modificação feita na estrutura interna de um software de maneira a facilitar seu entendimento e a diminuir os custos de futuras alterações, sem afetar seu comportamento externo.

Fowler *et al.* (1999) indicam que a intuição de um especialista é a melhor maneira de identificar código que necessita de refatorações. Porém, muitas equipes não dispõe de um especialista à sua disposição. Nesse tipo de equipe, pode ser interessante o uso de técnicas quantitativas para permitir a melhoria contínua do design do software. A identificação quantitativa de oportunidades de refatoração pode ser realizada pela utilização de métricas de código fonte.

O objetivo desse artigo é estudar o uso de métricas para identificar pontos do código que necessitam de melhoria e para decidir quais refatorações devem ser usadas nesses pontos.

Para isso, são identificadas métricas interessantes para a avaliação do design e da arquitetura de um projeto de software. Então, são estudadas ferramentas que auxiliam na geração, a partir de código, das métricas identificadas e é escolhida uma ferramenta. Depois, a ferramenta escolhida é aplicada em um estudo de caso, em dois projetos comerciais, de maneira a encontrar pontos de melhoria. Em seguida,

são examinadas quais refatorações podem ser feitas nos pontos de melhoria identificados. Finalmente, são discutidos os resultados obtidos e a metodologia adotada.

2 Referencial Teórico

2.1 Refatoração

De acordo com Fowler *et al.* (1999), uma refatoração pode ser definida como uma modificação feita na estrutura interna de um software de maneira a facilitar seu entendimento e a diminuir os custos de futuras alterações, sem afetar seu comportamento externo. São tomados pequenos passos (mover atributos entre classes, extrair métodos, etc) que, cumulativamente, melhoram a estrutura geral do código de um sistema.

Segundo Fowler *et al.* (1999), não se deve separar um tempo específico para refatorações. Refatorações devem ser pequenas e constantes e devem ser feitas enquanto se realiza alguma outra tarefa no código como ao implementar novas funcionalidades, ao resolver bugs no sistema ou ao participar de uma revisão de código.

Martin e Martin (2007) afirma que as refatorações devem ser feitas continuamente, sempre que for identificada um oportunidade de tornar o código mais legível, simples e expressivo.

Fowler *et al.* (1999) ressaltam a importância de testes unitários ao realizar uma refatoração, de maneira a garantir que o comportamento do sistema não foi alterado.

Beck (2002) explica que aplicar refatorações é o passo final do ciclo do Desenvolvimento Orientado a Testes (TDD). Após guiar a implementação por testes que falham, fazendo-os passar da maneira mais simples possível, deve-se melhorar o código através de refatorações.

2.1.1 Catálogo de refatorações

Entre as refatorações detalhadas por Fowler *et al.* (1999), estão as seguintes:

2.1.1.1 Extract Method

Extract Method é uma refatoração que deve ser usada para tornar um bloco de código em um novo método cujo nome explica o propósito do método. Fowler *et al.* (1999) relatam que essa é uma das refatorações mais utilizadas. A extração de blocos de código em métodos promove o reuso e a legibilidade.

2.1.1.2 Extract Class

Extract Class deve ser utilizada quando uma classe possui métodos ou atributos que devem pertencer a uma classe que ainda não existe. Um nova classe é criada e os métodos e atributos são movidos para a nova classe.

2.1.1.3 Move Method

Move Method é uma refatoração que deve ser utilizada quando um método usa mais atributos de outra classe do que atributos da própria classe a qual pertence. Deve ser criado um novo método na classe alvo e o método antigo pode ser removido ou deixado acrescido de uma deleção para a nova classe. Fowler *et al.* (1999) relatam que esta refatoração corrige classes que estão altamente acopladas, tornando-as mais simples, de maneira a desacoplá-las e corrigir a distribuição de responsabilidades.

2.1.1.4 Move Field

Move Field é uma refatoração que deve ser utilizada quando um atributo é utilizado mais por outra classe do que pela classe em que é definido.

2.1.1.5 Replace Conditional with Polymorphism

Replace Conditional with Polymorphism é uma refatoração em que é feita troca expressões condicionais (como if e switch) por uma interface comum. A implementação do comportamento pode ser feita através de herança ou pelos padrões State ou Strategy. O design resultante dessa refatoração permite que inserções/remoções de comportamentos sejam feitas apenas adicionando/removendo classes, sem a necessidade de alterar código existente.

2.2 Identificando oportunidades de Refatoração

Carneiro (2003) lista as seguintes abordagens de detecção de oportunidades de refatoração propostas na literatura:

- baseada em análise cognitiva
- baseada em Unified Modeling Language (UML)
- baseada em meta-programação declarativa
- baseada em invariantes
- baseada em métricas

2.2.1 Detecção baseada em análise cognitiva

A detecção de refatorações baseada em análise cognitiva é a abordagem recomendada por Fowler *et al.* (1999), que indica a intuição humana como melhor maneira de identificar código que possivelmente necessita de refatorações. Fowler *et al.* (1999) identificou *Code Smells*, que são heurísticas que ajudam na identificação de estruturas de código que necessitam de refatorações.

2.2.2 Detecção baseada em UML

Astels (2002) analisou a visualização de diagramas de classe e diagramas de sequencia do UML como meio de detectar oportunidades de refatoração, além de descrever uso de ferramentas de UML como auxílio para refatorações.

2.2.2 Detecção baseada em meta programação declarativa

Wuyts (2001) e Tourwé e Mens (2002) propuseram a detecção de oportunidades de refatoração através da consulta a regras definidas por meio de um ambiente de meta-programação lógica. Nesse ambiente, a declaração das regras de *design* a serem seguidas são feitas por meio da linguagem de programação lógica SOUL, uma variante de Prolog. Depois de feitas as consultas, são listadas as violações às regras definidas.

2.2.2 Detecção baseada em invariantes

Kataoka (2001) descreve uma abordagem que utiliza análise dinâmica do comportamento de um programa, através da ferramenta Daikon, de maneira a detectar automaticamente invariantes e relacioná-las a refatorações.

2.2.2 Detecção baseada em métricas

Simon, Steinbrückner e Lewerentz (2001) criaram ferramentas de visualização que mostram métricas relacionadas à coesão das classes e as relacionam com quatro possíveis refatorações: Move Method, Move Attribute, Extract Class e Inline Class. As métricas de coesão são mostradas graficamente como uma medida de distância entre os membros (métodos e atributos) de uma classe. Quando há uma grande distância entre um membro e os demais membros de uma classe e uma pequena distância entre o membro e membros de outras classes, há um problema de coesão.

Carneiro (2003) fez um estudo extensivo sobre a utilização de métricas de código fonte para a detecção de oportunidades de refatoração. Foram desenvolvidas duas abordagens: uma abordagem analítica, ou top-down, que foi chamada de Meta Pergunta Métrica e uma abordagem empírica, ou bottom-up, em que foram criados coeficientes de associação entre métricas e refatorações: Coeficiente de Associação entre Métrica e Refactoring (CAMR) e Coeficiente de Associação Forte entre Métrica e Refactoring (CAFMR).

Zhao e Hayes (2006) relataram a criação de uma ferramenta que cria uma lista priorizada de várias refatorações possíveis.

2.3 Métricas

Carneiro (2003) agrupas métricas orientadas a objetos em quatro categorias: métricas de tamanho, métricas de herança, métricas de complexidade e métricas de comunicação.

2.3.1 Métricas de tamanho

Métricas de tamanho partem do pressuposto de que estruturas de código muito grandes tendem a ser difíceis de manter. Por isso, é medido o tamanho de classes e métodos. Entre as medidas de tamanho, estão:

- Lines of Code (LOC): métrica aplicada a classes e métodos, proposta por Lorenz e Kidd (1994). O objetivo dessa métrica é obter a quantia de

linhas de código de uma classe ou método. A métrica Method Lines of Code (MLOC) trata apenas das linhas de código de métodos.

- Number of Methods (NOM): métrica aplicada a classes, proposta por Lorenz e Kidd (1994). Mede o número de métodos de cada classe.
- Number of Statements (NOS): métrica aplicada a métodos, proposta por Lorenz e Kidd (1994). Mede o número de sentenças em um método.
- Number of Attributes ou Number of Instance Variables (NIV): métrica aplicada a classes, proposta por Lorenz e Kidd (1994). Mede o número de atributos de instância de uma classe.
- Number of Static Attributes ou Number of Classe Variables (NCV): métrica aplicada a classes, proposta por Lorenz e Kidd (1994). Mede o número de atributos estáticos de uma classe.

2.3.2 Métricas de herança

Métricas de herança medem a implementação da hierarquia no software. Entre as métricas de herança, estão:

- Depth of Inheritance Tree (DIT): aplicada a classes e interfaces, proposta por Chidamber e Kemerer (1994). Mede a profundidade da hierarquia de herança, visando reuso, legibilidade e testabilidade.
- Number of Children (NOC): aplicada a classes e interfaces, proposta por Chidamber e Kemerer (1994). Mede o número de subclasses de uma determinada classe ou interface, visando reuso.
- Number of Inherited Methods (NMI): aplicada a classes, proposta por Lorenz e Kidd (1994). Mede o número de métodos herdados e definidos em uma superclasse.
- Number of Overriden Methods (NMO): aplicada a classes, proposta por Lorenz e Kidd (1994). Mede o número de métodos definidos em uma superclasse e redefinidos na subclasse.

2.3.3 Métricas de complexidade

Métricas de complexidade são medem de maneira heurística a legibilidade do código. Entre as métricas de complexidade, estão:

- McCabe's Cyclomatic Complexity (CC): métrica aplicada em classes e métodos, proposta por McCabe (1976). Através da medição das alternativas possíveis no controle de fluxo de um método ou classes, é feita uma estimativa da legibilidade do código. A cada divisão de fluxo do código (if, for, while, do, case, catch, operador ternários e operadores condicionais lógicos como && e ||) a métrica é acrescida de 1.
- Weighted Methods per Class (WMC): métrica aplicada em classes, proposta por Chidamber e Kemerer (1994). Mede complexidade, tamanho, esforço para manutenção e reuso. É a soma da complexidade de McCabe para todos os métodos de uma classe.

2.3.4 Métricas de comunicação

Métricas de comunicação medem o acoplamento e coesão entre as classes. Entre as métricas de comunicação, estão:

- Response for Class (RFC): aplicada a classes, proposta por Chidamber e Kemerer (1994). Mede o acoplamento, complexidade e pré-requisitos para teste.
- Coupling Between Objects (CBO): aplicada a classes, proposta por Chidamber e Kemerer (1994). Mede coesão e reuso.
- Lack of Cohesion in Methods (LCOM): aplicada a classes, proposta por Chidamber e Kemerer (1994). Mede coesão, complexidade, encapsulamento e uso de variáveis. É calculada pela média do número de métodos que acessam cada atributo da classe subtraída do número total de métodos e divida pelo número de métodos que não acessam o atributo em questão. Um valor baixo indica uma classe coesa e um valor próximo a 1 sugere que a classe pode ser dividida em algumas subclasses.

2.4 Relação entre métricas e refatorações

Carneiro (2003) relacionou a aplicação de refatorações à variação em várias métricas, de maneira a encontrar coeficientes de associação entre métricas e refatorações.

- Um método tem muitas linhas de código (a métrica MLOC é muito grande): deve ser utilizada a refatoração Extract Method.
- Uma classe tem muitos métodos (valor de NOM alto): devem ser utilizadas as refatorações Move Method ou até Extract Class.
- Uma classe com muitas variáveis de instância (valor de NIV alto): deve ser utilizada a refatoração Move Field.
- Uma classe com muitas variáveis estáticas (valor de NCV alto): deve ser utilizada a refatoração Move Field.
- Um método possui código com muitos desvios de fluxo (alto valor de CC): deve ser utilizada a refatoração Extract Method ou Replace Conditional with Polymorphism.
- Uma classe possui muitos métodos complexos (alto valor de WMC): devem ser utilizadas as refatorações Extract Method, Move Method, Extract Class ou Replace Conditional with Polymorphism.
- Uma classe utiliza muitos atributos de outra classe: devem ser utilizadas as refatorações Extract Method, Move Method ou até Extract Class.

3 Procedimentos Metodológicos

A pesquisa aqui apresentada pode ser classificada como um pesquisa descritiva, que tem por objetivo estudar qualitativamente a aplicação de métricas na identificação de oportunidades de refatoração, de maneira a estabelecer relações entre as métricas escolhidas e refatorações aplicadas, medindo o efeito das refatorações na evolução do software por meio das métricas.

Como método da pesquisa apresentada, foi utilizado um estudo de caso de dois projetos comerciais de uma empresa da região metropolitana de Belo Horizonte, cujo nome será mantido em sigilo. Os projetos estão sendo ativamente desenvolvidos

atualmente, através da linguagem de programação Java, com a Interface de Desenvolvimento Integrado (IDE) Eclipse.

O Projeto 1 está sendo desenvolvido há 5 meses e está em fase de finalização. É um projeto menor, que possui ao todo por volta de 20 mil linhas de código, sendo 20% de testes unitários, e possui 35 entidades persistidas.

O Projeto 2 já está em produção há mais de 1 ano, tendo levado 2 anos para ser finalizado. Trata-se de um produto da empresa e está em fase de manutenção. Possui ao todo por volta de 156 mil linhas de código, sendo 11% de testes unitários, e 231 entidades persistidas.

Como instrumento para coleta de dados, depois de analisadas diversas alternativas, foi escolhida a ferramenta Eclipse Metrics[1], que possibilita a geração e verificação de métricas de maneira bastante rápida.

Foram analisadas as seguintes métricas:

- MLOC (Method Lines of Code)
- NOM (Number of Methods)
- NIV (Number of Instance Variables)
- NCV (Number of Class Variables)
- CC (McCabe's Cyclomatic Complexity)
- WMC (Weighted Methods per Class) e
- LCOM (Lack of Cohesion in Methods).

Também foram estudadas outras ferramentas, além da Eclipse Metrics, para métricas e análise estática de código da linguagem Java como JDepend, PMD, FindBugs, Checkstyle, CRAP, JavaNCSS, Sonar, entre outras.

Algumas possuem métricas não medidas pela ferramenta Eclipse Metrics, como a ferramenta CRAP e a ferramenta JavaNCSS.

[1] http://eclipse-metrics.sourceforge.net/

Ferramentas como PMD, FindBugs e Checkstyle são ferramentas de análise estática de código que não necessariamente efetuam métricas a partir do código fonte, mas aplicam regras de aderência a padrões de código ou tentam identificar padrões de código que possivelmente possuem bugs. Esse tipo de ferramenta é importante especialmente como plugin de ferramentas de integração contínua, pois podem falhar o build de código não aderente aos padrões ou com possíveis bugs assim que o código for inserido no controle de versão.

A ferramenta Sonar também é uma ferramenta bastante interessante, que gera um website com um histórico do resultado das ferramentas citadas acima. Esse site pode ser gerado periodicamente, a partir de um servidor de integração contínua e permite que sejam analisadas as tendências de qualidade de código do software que está sendo desenvolvido.

A ferramenta Eclipse Metrics é especialmente interessante porque é integrada com a IDE Eclipse. Isso permite a análise contínua das métricas enquanto estão sendo feitas as refatorações, tornando mais curto o ciclo de feedback entre a alteração do código e a verificação do efeito nas métricas.

4 Análise dos dados

Foram medidas as métricas selecionadas nos dois projetos do estudo de caso, através da ferramentas Eclipse Metrics da IDE Eclipse.

4.1 MLOC (Lines of Code)

Para o Projeto 1, a maior medida de linhas de código de um método é 104, enquanto no Projeto 2 é 508.

É interessante notar que, tanto para o Projeto 1 como para o Projeto 2, os métodos com mais linhas de código são parte da configuração da infra-estrutura de Injeção de Dependência. Para melhorar esse código, diminuindo as linhas de código, deveriam ser utilizados módulos, de maneira a isolar a configuração de injeção de dependência em pacotes menores. Já era um ponto de melhoria conhecido, porém

não foi possível realizar essa separação em métodos em decorrência de limitações técnicas.

No Projeto 1, o segundo maior método é responsável pela geração de relatórios, com 73 linhas de código. O código foi feito às pressas, para a entrega de uma funcionalidade, que conscientemente foi deixado como ponto de melhoria. É interessante notar que é o método com maior complexidade ciclomática, 20. É um ponto interessante para focar na refatoração. Outro ponto de refatoração identificado através do número de linhas de código, é um método de importação de dados, com 65 linhas de código. O terceiro ponto de melhoria, um método de cálculo matemático em planilhas, tem 37 linhas de código, e também foi uma escolha consciente entre velocidade de entrega e legibilidade, sendo uma ótima oportunidade para refatoração.

No Projeto 2, desconsiderando código de infra-estrutura, os método de geração de relatórios de várias classes do sistema são os mais extensos com 200, 101, 74 e 71 linhas de código, respectivamente.

Em todos os casos mencionados acima, seria interessante utilizar a refatoração Extract Method, para diminuir as linhas de códigos dos métodos, quebrando o código em métodos menores e possivelmente inspirando a criação de novas classes ou movendo métodos para outras classes.

4.2 NOM (Number of Methods)

Para o Projeto 1, havia uma entidade com 32 métodos de acesso (getters e setters). Tratava-se de uma entidade que mapeia dados de uma simulação. Na verdade, essa classe poderia ser separada em outras classes, de maneira a tornar o código mais organizado e legível. Havia outra classe com 32 métodos e era responsável por gerar relatórios.

Para o Projeto 2, a classe com maior número de métodos é uma entidade com 116 métodos. Os métodos dessa entidade são métodos de acesso (getters e setters). Os métodos foram criados para fazer o mapeamento objeto-relacional de entidades que

estavam relacionadas à classe em questão, de maneira a disparar a remoção das entidades. Foi um ponto de melhoria conhecido pela equipe. A segunda classe com maior número de métodos é uma classe genérica usada por todos os serviços do sistema que são acessíveis externamente e possui 78 métodos.

Nos casos acima, é interessante utilizar a refatoração Move Method, para distribuir comportamentos para classes que possuem a responsabilidade. Se não houverem classes com as responsabilidades, é interessante utilizar a refatoração Extract Class.

4.3 NIV (Number of Instance Variables)

Para o Projeto 1, a classe com maior número de atributos é a mesma classe com maior número de métodos. Esta classe possui 16 atributos. O número de métodos está relacionado porque a maioria dos métodos era de acesso (getters e setters). A segunda classe com maior número de atributos é uma classe com 14 atributos que faz parte do núcleo do sistema.

Para o Projeto 2, a classe com maior número de atributos também coincidiu com a classe com maior número de métodos, com 53 atributos. Foi interessante notar uma classe com 37 atributos que serve como ponte entre dois frameworks utilizados para infra-estrutura.

Nos dois projetos, uma refatoração que seria interessante para diminuir essa métrica seria Extract Class.

4.4 NCV (Number of Class Variables)

Para o Projeto 1, as classes com maior número de atributos estáticos, variando de 2 a 4, são classes utilitárias nas quais são definidas constantes.

Para o Projeto 2, há uma classe com 21 atributos estáticos, que é responsável pela exportação de dados para o formato .mdb, do Microsoft Access. Esses atributos estáticos são constantes que são utilizadas na configuração do formato, mas poderiam ser separados em uma classe responsável pela configuração.

Para o Projeto 1, não há necessidade de realizar nenhuma refatoração inspirada pela métrica NCV. Já para o Projeto 2, seria interessante usar Extract Class seguido de Move Field para os métodos estáticos, de maneira a deixar o código mais organizado.

4.5 CC (Cyclomatic Complexity)

Para o Projeto 1, o método com maior complexidade ciclomática tem 20 como valor e é o mesmo método que possui maior número de linhas de código. É um alvo interessante para refatorações. O método equals() das classes que modelam uma planilha são outros pontos de melhoria, já que possuem de 11 a 13 de complexidade ciclomática. Outros métodos passíveis de melhoria quanto à métrica CC são métodos que implementam consultas genéricas ao Banco de Dados e métodos de validação de coordenadas espaciais.

Para o Projeto 2, há um método com 53 como valor de complexidade ciclomática, que possui 130 linhas de código. A legabilidade pode ser bastante melhorada. A classe que serve de ponte entre frameworks e que possui alto número de atributos, também possui alta complexidade ciclomática (38). No Projeto 2, há mais de 35 classes com métodos cuja complexidade ciclomática é maior que 10, um nível alarmante.

Nos casos acima, a refatoração que faria com que a complexidade fosse diminuída seria Extract Method. Dependendo do tamanho da classe e da relação entre os métodos extraídos, talvez fosse interessante utilizar a refatoração Extract Class. Para o caso do Projeto 1, que possui uma sequência de várias cláusulas switch, seria interessante utilizar a refatoração Replace Conditional with Polymorphism.

4.6 WMC (Weighted Method per Class)

Para o Projeto 1, a classe com maior WMC, com 69 como valor medido, foi uma classe utilitária.

Para o Projeto 2, desconsiderando classes do framework de interface com o usuário (UI), o maior WMC medido foi 161, de uma classe responsável pelo comportamento comum dos serviços do sistema. Essa mesma classe foi uma das classes com maior valor para a métrica NOM, que mede número de métodos. As classes de exportação de dados, que possuem alto valor para NCV (medida de variáveis estáticas), também possuem alto valor de WMC.

Podem ser utilizadas as refatorações Extract Method, Move Method ou Extract Class para diminuir essa métrica, melhorando assim a legibilidade do código.

4.7 LCOM (Lack of Cohesion in Methods)

Como a métrica LCOM varia de 0 a 1, foi plotada multiplicada por 100 na Figura 1.

Para o Projeto 1, houve uma medida de 1 (que é uma medida ruim) para a LCOM em duas classes: uma das classes gera avisos e a outra oferece o serviço de CRUD para uma funcionalidade de configuração de simulações.

Para o Projeto 2, foram 17 classes com medida de LCOM de 1. São classes dos mais variados pacotes, como validação, persistência, entre outros.

Para diminuir a métrica LCOM, é interessante utilizar as refatorações Extract Method, Extract Class e/ou Move Method.

5. Considerações Finais

A utilização de métricas de código fonte para a detecção de oportunidades de refatoração é realmente efetiva. Além da detecção de pontos de melhoria que apenas a intuição de um especialista seria capaz de identificar, o uso de métricas mostrou-se especialmente importante ao analisar bases de código pouco familiares. Métricas de código fonte são uma maneira interessante de enxergar o todo e medir quantitativamente a qualidade do código fonte.

Nos dois projetos comerciais analisados, foram identificados vários pontos de melhoria. Em ambos os projetos foi acumulado débito técnico, ou seja, algumas funcionalidades foram entregues com possíveis pontos de melhoria na qualidade de código para atender o cronograma do projeto.

Porém, a relação entre as métricas medidas e as possíveis refatorações precisam ainda da intuição de um especialista. Muitas das métricas indicam várias possíveis refatorações e há a necessidade da análise de uma pessoa experiente em orientação por objetos, design de software e refatorações para escolher qual caminho deve ser tomado.

Outro ponto a ser destacado é a necessidade de testes unitários no momento de se efetivar uma refatoração. É necessária uma suíte de testes extensa para garantir que uma refatoração, quando efetuada, não modificou o comportamento externo do sistema. Nos dois projetos estudados, alguns pontos do código não possuíam uma suíte de testes adequada. Isso pode ser medido com ferramentas de cobertura de código, como EclEmma[1], que permitir fazer uma análise de quais linhas de código foram exercitadas pela suíte de testes unitários.

Não foram analisadas nesse artigo métricas de acoplamento entre pacotes, como Afferent/Efferent Coupling e Instability. Essas métricas são interessantes para identificar quais as classes mais usadas no sistema, de maneira a oferecer um insumo para a priorização dos vários pontos de melhoria possível. Estudar a relação entre as métricas de acoplamento entre pacotes e as outras métricas, de maneira a priorizar refatorações é uma linha de pesquisa interessante.

[1] http://www.eclemma.org/

6. Referências

ASTELS, D. Refactoring with UML. *XP 2002*. Disponível em: http://cf.agilealliance.org/articles/system/article/file/915/file.pdf. Acesso em 23 de julho de 2011.

BECK, K. *Test Driven Development*: by example. Boston, Estados Unidos da América: Addison Wesley Professional, 2002.

CARNEIRO, G. F. *Usando Medição de Código Fonte Para Refactoring*. 2003. 129 f. Dissertação (Mestrado em Redes de Computadores) – Universidade Salvador, Salvador, 2003.

CHIDAMBER, S.; KEMERER, C. Metrics Suite for Object Oriented Dedsign. *IEEE Transactions on Software Engineering*. [S. l.], vol. 20, n. 6, junho de 1994.

FOWLER, M. *et al*. *Refactoring:* improving the design of existing code. Boston, Estados Unidos da América: Addison Wesley, 1999.

KATAOKA, Y. *et al*. Automated support for program refactoring using invariants. *Proceedings of the International Conference on Software Maintenance*, p. 736-743, Florença, Itália, 2001.

LORENZ, M.; KIDD, J. *Object-Oriented Software Metrics*. Nova Jersey, Estados Unidos da América: Prentice Hall, 1994.

MARTIN, R.; MARTIN, M. *Agile Principles, Patterns and Practices in C#*. 2. ed. Nova Jersey, Estados Unidos da América: Prentice Hall, 2007.

McCABE, T. A software complexity measure. *IEEE Transactions on Software Engineering.* [S. l.], vol. 2, n. 4, p. 308-320, dezembro de 1976.

SIMON, F.; STEINBRÜCKNER, F.; LEWERENTZ, C. Metrics Based Refactoring. *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering,* Lisboa, Portugal, março de 2001.

TOURWÉ, T.; MENS T. Automatically identifying refactoring opportunities using logic meta programming. *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering,* Benevento, Itália, março de 2003.

WUYTS, R. *A logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. 2001. 138 f. (Tese, Ciência da Computação) – Vrije Universiteit Brussel, Bruxelas, Bélgica, 2001.

ZHAO, L; HAYES, J. F. Predicting Classes in Need of Refactoring: An Application of Static Metrics. *Proceedings of the Workshop on Predictive Models of Software Engineering*, Filadélfia, Estados Unidos da América, setembro de 2006.