Qualidade de Produto de *Software*: uma abordagem baseada no controle da complexidade

Marcelo Criscuolo

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: ____/____/_____

Assinatura:_____

Qualidade de Produto de *Software*: uma abordagem baseada no controle da complexidade

Marcelo Criscuolo

Orientador: ***Profa. Dra. Rosely Sanches***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

USP – São Carlos
Fevereiro/2008

Agradecimentos

Agradeço a Deus por tudo o quanto tem me feito e por sua misericórdia infinita.

À minha esposa Ana, pelo seu constante apoio manifesto de inúmeras maneiras e pela sua preocupação com o desenvolvimento deste trabalho. Também ao Pedro, nosso pequeno filho, por contribuir mantendo a minha cabeça fria.

Aos meus pais, José Eduardo e Claudete, por todo esforço que já fizeram em meu favor. Agradeço particularmente ao meu pai por me incentivar, por intermédio da minha mãe, como de costume, me fazendo saber que ele não ficaria nada contente se eu não terminasse o mestrado.

À professora Rosely Sanches, por ter verdadeiramente me orientado e por suas atitudes sempre humanas (no melhor sentido que esse adjetivo possa ter) e compreensivas.

Aos professores Adenilso e João Batista, pelo código-fonte cedido.

À professora Rosana Vaccare Braga, pela oportunidade de trabalho que me permitiu dedicar mais tempo à escrita deste texto.

A Glauco Carneiro, da Universidade Salvador, por ter me enviado sua dissertação a respeito de refatoração.

Ao Professor Bill Opdyke, autor da primeira tese sobre refatoração, e ao seu orientador de doutorado, Professor Ralph Johnson, por responderem os *e-mails* a respeito de refatoração que lhes enviei no início deste trabalho.

Ao mais leve comedor de alface mineiro-paulistano-araraquarense, Sidney Leal, e à sua inseparável fornecedora de origamis, Ana Nakamura, por terem dedicado parte do seu escasso tempo à realização do estudo de caso apresentado neste trabalho.

Ao André Scandaroli, pela sua mania esquisita e contagiosa de conversar em voz passiva que acabou sendo muito útil para a escrita deste texto acadêmico.

A todos que já aplicaram seus esforços no desenvolvimento de *software* de código aberto, por terem viabilizado a execução deste trabalho sobre uma plataforma totalmente livre. Em particular, à empresa Red Hat pelas fontes livres *Liberation*, metricamente equivalentes a outras fontes proprietárias tomadas como padrão em diversos contextos.

A todos que de alguma maneira contribuíram, ainda que inconscientemente, para a realização deste trabalho.

“Qualquer tonto é capaz de escrever um código que um computador consiga entender. Os bons programadores escrevem códigos que outros humanos consigam entender.”

Martin Fowler

Resumo

É rara a preocupação com a qualidade de implementação de *software*. Pode-se observar que as estruturas internas dos *software*s são freqüentemente complexas e desorganizadas, especialmente no caso dos *software*s que são ditos orientados a objetos. Essa complexidade afeta diretamente a manutenibilidade e a susceptibilidade a erros, dificultando a alteração e a adição de novas funcionalidades aos *software*s. As próprias alterações, inerentes aos *software*s, os tornam mais complexos, o que agrava o problema. Neste contexto, acredita-se que o controle da complexidade pode levar a produtos de *software* de melhor qualidade. Assim, trata-se neste trabalho da manutenção preventiva, implementada por meio de inspeções, refatorações e análise de métricas. São estudadas falhas de manutenibilidade em uma amostra de programas orientados a objetos e, a partir dos resultados, são propostos artefatos de apoio para um processo de inspeção de *software* e modelos para os produtos de trabalho gerados nesse processo. Propõe-se o uso da técnica de leitura PBR (Leitura Baseada em Perspectivas) como uma maneira de se melhorar a detecção de falhas de manutenibilidade. Finalmente, a proposta deste trabalho foi validada por meio de um estudo de caso.

Palavras-chave: *manutenibilidade, complexidade de software, qualidade de software, inspeção, PBR, refatoração, métricas.*

Abstract

The commitment with the quality of software implemetation is rare. It's possible to observe that the software internal structures are frequently complex and disorganized, especially when talking about software that is said to be object-oriented. This complexity directly affects maintainability and error proneness, making it difficult to change and to add new functionalities to software. Changes themselves, that are inherent in software, make it more complex, and that makes the problem more serious. In this context, it's believed that the control of complexity can lead to better quality software products. Thus, the subject of this work is the preventive maintenance, implemented by means of inspections, refactoring and metric analysis. Maintenance flaws were studied in a sample of object-oriented programs and, based on the results, support artifacts for an inspection process were proposed, along with models of work products for this process. The use of PBR (Perspective-Based Reading) technique is proposed as a means of improving the detection of maintenance flaws. Finally, the proposal of this work is validated through a case study.

Keywords: *maintainability, software complexity, software quality, inspection, PBR, refactoring, metrics.*

Lista de Figuras

Sumário

Capítulo 1 - Introdução

1.1 Contexto e Motivação

*Software*s mal implementados são abundantes. Implementações complexas e de má qualidade são parte do cotidiano dos desenvolvedores de *software*. Abordando o paradigma da orientação a objetos (OO), pode-se encontrar, por exemplo, problemas de acoplamento – em níveis de abstração mais altos – até implementações de métodos desnecessariamente complexas – no nível de codificação.

Foi demonstrado por meio de estudos que a complexidade do *software* compromete a sua compreensibilidade, afeta negativamente a manutenibilidade e aumenta a susceptibilidade a erros [Subramanyam e Krishnan, 2003]. Num outro trabalho [Binkley e Schach, 1998], medidas de acoplamento entre artefatos de *software* foram relacionadas à taxa de erros. Outros autores [Cartwright e Shepperd, 2000] ainda questionaram o uso da herança, no que diz respeito a ter sido ou não aplicada adequadamente, após constatarem que as classes de níveis mais baixos de uma árvore hierárquica eram as que apresentavam as maiores taxas de erros. Esses mesmos autores também apontaram no sistema que estudaram, como uma possível causa de erros, a presença de classes muito longas e a necessidade de realizar alterações comuns em grupos de classes – sendo que essa necessidade indica potencial de generalização não explorado.

Pode-se perceber, portanto, que, no que concerne aos erros, há um ciclo do qual fazem parte a má implementação (complexidade), os erros e a manutenção do *software*: a má implementação torna o *software* mais susceptível ao surgimento de erros, e a correção de

erros se dá pela manutenção, que é diretamente prejudicada pela má implementação.

No que concerne, por sua vez, à adição de novas funcionalidades (também chamada de evolução [Figueiredo, 2005]), conclui-se que o nível de complexidade da implementação do *software*, ao afetar particularmente a compreensibilidade, é de importância crítica visto que, para adicionar funcionalidades a um *software* existente, é preciso fundamentalmente compreender como estão organizadas suas estruturas. Outros problemas gerados pela complexidade de implementação, como o acoplamento entre entidades, também afetam negativamente a evolução do *software*.

Por vários motivos – incluindo a correção de erros e a adição de novas funcionalidades – a entidade *software* sofre mudanças durante o seu ciclo de vida [Sommerville, 2001] [Brooks, 1987], e a execução dessas mudanças está sujeita ao obstáculo que é a complexidade de implementação. Ainda que tenha havido a preocupação em minimizar a complexidade de implementação durante a fase de desenvolvimento, as sucessivas mudanças tendem a tornar as estruturas do *software* mais complexas [Mens e Tourwé, 2004], de maneira que, sem nenhuma medida preventiva, o *software* tende a se deteriorar [Pressman, 2005].

Nesse contexto em que as mudanças ocorrem inevitavelmente, controlar a complexidade de implementação do *software* ao longo de todo o seu ciclo de vida é uma medida essencial para se produzir *software* de melhor qualidade e para tornar a execução das mudanças menos árduas e dispendiosas. Sommerville [2001] argumenta que, devido à manutenção (as mudanças) representar uma parte significativa do custo de um *software*, o custo total tende a diminuir quando mais esforços são investidos para melhorar a manutenibilidade.

O controle da complexidade se dá pela chamada **manutenção preventiva**, que consiste justamente em reorganizar as estruturas do *software* com o objetivo de melhorar a sua manutenibilidade [Pressman, 2005]. O primeiro passo para a realização da manutenção

preventiva é identificar as estruturas do *software* que apresentam problemas, e isso pode ser feito por meio de inspeções do código-fonte.

As inspeções são um processo para a revisão de um artefato de *software* em busca de defeitos [Fagan, 1976]. Uma etapa importante da inspeção é a leitura dos artefatos visando a encontrar defeitos. Dependendo de como for definido o processo de inspeção, o inspetor pode realizar a leitura baseando-se unicamente em seu julgamento pessoal, ou pode contar com alguma orientação, sendo a Leitura Baseada em Perspectivas (PBR[1]) [Shull et al., 2000] [Basili et. al, 1996] uma técnica de leitura. A PBR é baseada na premissa de que o trabalho de um inspetor é mais eficaz se a leitura dos artefatos for feita de apenas uma perspectiva, isto é, buscando por apenas uma categoria de defeitos (falhas estruturais, por exemplo). As perspectivas são determinadas por cenários que são, por sua vez, compostos por instruções e perguntas que esclarecem ao inspetor qual é o seu papel e o induzem a encontrar defeitos no artefato examinado.

Após a identificação das falhas estruturais é preciso corrigi-las, o que implica em reestruturar partes do *software*. Uma técnica útil para esse propósito, aplicada no âmbito da programação orientada a objetos, é a **refatoração** [Opdyke, 1992]. Refatorações são transformações que preservam o comportamento de um programa enquanto melhoram a sua estrutura interna [Fowler, 1999a]. Tais como os padrões de projeto de *software* [GoF, 1994], as refatorações são catalogadas no formato de linguagens de padrões.

Finalmente, na última etapa de uma inspeção, as correções devem ser validadas. Normalmente, um participante do processo de inspeção valida as correções e determina se são ou não satisfatórias. No entanto, acredita-se que as métricas de produto de *software* podem ser utilizadas para detectar, em particular, as correções de falhas estruturais, visto que aplicações semelhantes podem ser encontradas na literatura [Demeyer et al., 2000] [Kataoka et al.,

1 Do inglês, *Perspective-Based Reading*.

2002].

Defende-se, neste trabalho, que a prática da manutenção preventiva, implementada por meio de inspeções, refatorações e análise de métricas, contribui para a produção de *software*s de melhor qualidade ao servir como uma medida de controle da complexidade inerente às mudanças pelas quais passa o *software* durante todo o seu ciclo de vida. Assim, são estabelecidos os objetivos descritos a seguir.

1.2 Objetivos

É proposta deste trabalho realizar um estudo das falhas de modelagem de *software* OO – implementados na linguagem Java [Java, 2007] – com o objetivo de identificar as falhas mais freqüentes e de buscar evidências que permitam reconhecê-las. Além disso, pretende-se estudar o comportamento de métricas de produto de *software* OO diante de eliminação das falhas.

A partir desses estudos, busca-se propor artefatos de apoio a um processo de inspeção de *software* e modelos para os produtos de trabalho gerados. Tem-se como objetivo definir as atividades do processo de tal maneira que seja prevista a melhoria contínua dos artefatos de apoio.

Propõe-se desenvolver os seguintes artefatos de apoio ao processo de inspeção:

- cenário PBR para auxiliar na leitura dos artefatos;
- relação de falhas de manutenibilidade conhecidas;
- documento de recomendações de correções de falhas de manutenibilidade baseadas na aplicação de refatorações; e
- documento de métricas para validação das correções.

Propõe-se criar um modelo de relatórios de falhas encontradas nas inspeções e de relatórios de correção.

1.3 Organização

O restante deste texto está distribuído em mais 6 capítulos. No Capítulo 2 é feita uma revisão sobre garantia de qualidade de *software* e manutenção preventiva, abordando também a refatoração. No capítulo 3, trata-se de inspeção de *software*, Leitura Baseada em Perspectivas (PBR) e de métricas de produto de *software* orientados a objetos.

A seguir, no capítulo 4, é descrito o estudo realizado a respeito das falhas de manutenibilidade e de sua relação com métricas de *software*. Nesse capítulo também são apresentados os resultados diretos desse estudo.

No capítulo 5 são apresentados o processo de inspeção (detalhando cada atividade), os artefatos de apoio gerados a partir dos resultados do estudo realizado e os modelos dos relatórios de falhas encontradas e de correção, os quais são os produtos de trabalho das principais atividades do processo.

No capítulo 6 discorre-se sobre um estudo de caso de aplicação do processo de inspeção proposto numa pequena empresa de desenvolvimento de *software*. O principal objetivo do estudo de caso é a verificação da aplicabilidade do processo.

No capítulo 7 são apresentadas as conclusões a respeito deste trabalho e as propostas de trabalhos futuros.

Finalmente, encontram-se as referências bibliográficas para os trabalhos mencionados ao longo deste texto e a bibliografia referente a outros materiais não mencionados diretamente.

Capítulo 2 - Garantia de Qualidade de *Software* e Manutenção Preventiva

2.1 Considerações Iniciais

Neste capítulo é feita uma revisão sobre qualidade de *software*, abordando a Garantia de Qualidade de *Software* (GQS) com foco particular na manutenibilidade.

2.2 Garantia de Qualidade de *Software*

Software é algo essencialmente complexo [Brooks, 1987], tanto é que, de acordo com Sommerville [2001], a noção clássica de qualidade aplicada a produtos manufaturados, segundo a qual um produto deve estar em conformidade com suas especificações, não pode ser aplicada diretamente a produtos de *software*. Sommerville argumenta que essa definição não prevê problemas como requisitos implícitos (o cliente pode estar interessado na manutenibilidade do *software*, mas não ter explicitado isso nos requisitos), a dificuldade de se especificar claramente certos requisitos (manutenibilidade, por exemplo) e, ainda, a dificuldade de se escrever especificações completas.

Pressman [2005] define qualidade de *software* como a

> conformidade com os requisitos funcionais e de desempenho explicitamente declarados, com padrões de desenvolvimento explicitamente documentados e com características implícitas esperadas de todo *software* desenvolvido profissionalmente.

Percebe-se, pela sua subjetividade, que a parte final da definição adotada por Pressman ("*... características implícitas esperadas de todo* software *desenvolvido profissionalmente*")

acaba por ir ao encontro do argumento de Sommerville [2001]. Pressman [2005] ainda aponta que duas categorias de qualidade podem ser encontradas: qualidade de projeto[2] e qualidade de conformidade (ou de adequação). A qualidade de projeto refere-se às características – não relacionadas às funcionalidades – especificadas pelos projetistas para um produto, como, por exemplo, nível de tolerância a falhas, níveis de desempenho que devem ser satisfeitos e grau de manutenibilidade do código-fonte. Quanto mais rigorosas essas especificações, mais qualidade terá o produto, uma vez que essas sejam seguidas. A qualidade de conformidade diz respeito ao grau em que as especificações do projeto foram seguidas durante o processo de produção. Quanto mais estritamente forem seguidas as especificações do projeto, mais alto será o nível de qualidade de conformidade [Pressman, 2005].

Os esforços para a obtenção da qualidade baseiam-se, normalmente, na definição de padrões e no estabelecimento de procedimentos que garantam a sua adoção. Esses esforços são apresentados na norma internacional ISO/IEC 12207 [Walker, 1999] [Singh, 1995] – uma referência para processos de ciclo de vida de *software* – como um processo de apoio: o processo de garantia (ou gerência) de qualidade. A garantia de qualidade de *software* é normalmente estruturada em três atividades [Runeson e Isacsson, 1998] [Boegh et al., 1999] [Sommerville, 2001]: garantia de qualidade, planejamento de qualidade e controle de qualidade.

2.2.1 Garantia de Qualidade

Nessa atividade o objetivo é definir um arcabouço[3] de procedimentos e padrões de qualidade que serão aplicados na produção de *software* [Boegh et al., 1999]. Considera-se tanto o estabelecimento de padrões próprios baseados em experiências bem sucedidas da organização como a adoção de padrões já existentes.

2 A palavra “projeto” é usada, neste caso, como uma tradução para a palavra inglesa *design*.
3 Neste trabalho, o termo “arcabouço” é usado como uma tradução para a palavra inglesa *framework*.

São dois os tipos de padrões que podem ser estabelecidos [Runeson e Isacsson, 1998] [Pressman, 2005]: padrões de produto e padrões de processo. Padrões de produto são aplicados diretamente ao *software* que se desenvolve. Convenções de nomenclatura de variáveis, estilos de formatação de código-fonte e uso de comentários são exemplos padrão de produto. Os padrões de processo definem processos que devem ser seguidos durante o desenvolvimento de *software*. Exemplos desse tipo de padrão são os processos de especificação, projeto (ou elaboração) e validação de *software*.

As organizações normalmente mantêm os padrões em um repositório, que pode ser desde um documento impresso a um sistema disponível na sua rede interna. Uma vez que os padrões próprios são definidos com base em experiências bem sucedidas, aconselha-se usar como ponto de partida para a criação do repositório – enquanto essas experiências ainda não tiverem sido acumuladas – as normas e padrões de organizações nacionais (exemplo: ABNT – Associação Brasileira de Normas Técnicas) e internacionais (exemplo: ISO – *International Organization for Standardization*) [Sommerville, 2001].

2.2.2 Planejamento de Qualidade

A simples existência de um repositório de padrões não é suficiente para que *software* de qualidade seja produzido. Os desenvolvedores podem ignorar o repositório, ou ainda adotar os padrões desorganizadamente de acordo com critérios individuais, de maneira que poderiam acabar se atrapalhando mutuamente e o objetivo não seria atingido (por exemplo, um pode concentrar seus esforços na eficiência enquanto outro os concentra na reusabilidade, e sabe-se que esses atributos de qualidade podem ser conflitantes). É necessário, portanto, que os atributos de qualidade desejados para cada projeto sejam determinados logo no início e que, com isso, seja montado um plano de qualidade [Boegh et al., 1999]. Além disso, o plano deve conter outras informações como a forma de se avaliar a qualidade, quais padrões serão

adotados visando a alcançar a qualidade de acordo com os atributos escolhidos, e mais: quais padrões serão adotados tal como são descritos no repositório, quais sofrerão adaptações e quais padrões serão estabelecidos especificamente para o projeto, caso isso seja necessário. Sommerville [2001] ainda aponta que o plano de qualidade deve ser um documento sucinto, já que os envolvidos no projeto tendem a oferecer resistência a documentos longos.

2.2.3 Controle de Qualidade

Na atividade de controle de qualidade trata-se do problema de como garantir que os procedimentos e padrões listados no plano de qualidade sejam seguidos pelos envolvidos no projeto. Há duas abordagens complementares para o controle de qualidade [Boegh et al., 1999] [Sommerville, 2001]: as revisões e as análises automáticas.

Nas revisões, os artefatos de *software* são examinados – por pessoas diferentes de quem os produziu – com o objetivo de encontrar problemas que são enumerados e enviados ao autor do produto para que os resolva . Revisões podem ser feitas em qualquer artefato de *software* como documento de requisitos, projetos de interface com o usuário, código-fonte e até mesmo em processos de trabalho; sempre produzindo como resultado uma enumeração dos problemas encontrados para que sejam tratados posteriormente [Rai et al., 1998] [Mafra e Travassos, 2005].

As análises automáticas são baseadas fundamentalmente em métricas [Rai et al., 1998] [Boegh et al., 1999]: medidas são coletadas dos produtos de *software* e de processos, os valores são então analisados de acordo com padrões pré-estabelecidos de maneira que é possível tirar conclusões a respeito da qualidade dos artefatos medidos. Contudo, não costuma haver uma relação direta entre atributos de qualidade e métricas. Não é possível medir diretamente a manutenibilidade, por exemplo. Assim, assume-se que há um relacionamento entre os atributos de qualidade como a manutenibilidade e aspectos mensuráveis dos produtos

de *software* [Mafra e Travassos, 2005], tais como o tamanho de programas em linhas de código, número de blocos de código aninhados[4] e complexidade ciclomática. Dessa maneira, para que conclusões a respeito de um atributo de qualidade possam ser tiradas, analisa-se o comportamento das métricas relacionadas a esse atributo e, se não estiverem em conformidade com os padrões, conclui-se que o artefato medido pode apresentar defeitos a serem corrigidos.

Tendo sido feita uma revisão dos conceitos de qualidade de *software*, trata-se a seguir, na seção 2.3, da manutenção preventiva como uma prática para a obtenção da qualidade.

2.3 Manutenção Preventiva

É fato que a entidade *software* está sempre sujeita a mudanças [Sommerville, 2001] [Brooks, 1987]. Problemas são descobertos e precisam ser corrigidos, novos requisitos surgem, leis e regras são alteradas, enfim, diversos são os fatores que levam à necessidade de se alterar um *software* que já esteja em uso. Há ainda os que defendem que o *software* deve ser cultivado[5] [Figueiredo, 2005] [Bryant, 2000] [Brooks, 1987], evoluir aos poucos, que deve começar como um sistema simples, que contenha apenas as funcionalidades básicas e evoluir até um estágio completo, de maneira semelhante ao que acontece com os seres vivos. É proposta a mudança da metáfora da engenharia – pela qual outrora deixou-se de **escrever** para **construir** *software* [Brooks, 1987] – pela metáfora do viver. Fala-se sobre "TI[6] orgânica", sobre encarar o ciclo de vida de um *software* como o de um ser vivo que nasce, cresce, evolui e morre [Figueiredo, 2005]. Nesse contexto, o *software* passa por contínuas adaptações (passa por mudanças) até atingir a "fase adulta" e mesmo depois (a fase de evolução nesse caso equivale à de manutenção). Assim, desde o clássico modelo cascata [Pressman, 2005] [Sommerville, 2001] até os mais recentes modelos de processo de desenvolvimento, os ágeis

4 Do inglês, *nested*.
5 Do inglês, *grow*.
6 TI – Tecnologia da Informação

[Figueiredo, 2005] [Beck, 2004] [Fowler, 1999a], a mudança aparece em algum momento. No primeiro, aparece apenas no final, na fase de suporte; e no último, aparece como um componente principal do próprio processo de desenvolvimento.

As estruturas do *software*, em particular o código-fonte, ao passarem por sucessivas alterações que corrigem, modificam e introduzem novas funcionalidades, tendem a se tornar mais complexas e a se desviar do projeto original [Mens e Tourwé, 2004] – a se deteriorar [Pressman, 2005] – de maneira que a qualidade do produto de *software* acaba sendo comprometida. Ou seja, após passar por um certo número de alterações, o *software* atinge um estado de deterioração que acaba por impedir que novas (e necessárias) alterações sejam realizadas sem que essas introduzam defeitos críticos no produto. Além disso, esse estado de deterioração e complexidade torna as alterações muito mais difíceis, trabalhosas e, consequentemente, dispendiosas.

Conclui-se, portanto, que se nenhuma medida de prevenção for adotada, as alterações tendem naturalmente a diminuir a manutenibilidade do *software*. Assim, a manutenção preventiva visa justamente a impedir a deterioração do *software* garantindo que esse esteja num estado em que possa receber alterações com uma certa segurança, isto é, minimizando os riscos da introdução de defeitos e diminuindo a complexidade de suas estruturas. A importância de tornar a alteração, isto é, a manutenção dos sistemas de *software*, menos dispendiosa por meio da prevenção de sua deterioração torna-se evidente se for considerado o fato – apontado por trabalhos científicos – de que os esforços de manutenção costumam representar uma parte significativa do custo dos sistemas de *software* [Sommerville, 2001].

Portanto, a manutenção preventiva é uma atividade de inegável importância, mais ainda se a manutenibilidade for um dos atributos de qualidade previstos no plano de qualidade[7]. O aumento da manutenibilidade pode se dar pela execução da manutenção preventiva em outros

7 Ver sub-seção 2.2.2.

artefatos além do código-fonte, como os de documentação, por exemplo; contudo, trata-se neste trabalho apenas do código-fonte e é considerada, nesse contexto, como manutenção preventiva, a aplicação de *técnicas de reestruturação de código-fonte*. Tratando-se do paradigma da orientação a objetos, no qual se encerra o escopo desse trabalho, a reestruturação recebe o nome de refatoração. Esse tema é abordado na seção a seguir.

2.4 Refatoração

O termo refatoração[8] foi originalmente introduzido por Opdyke em sua tese de doutorado [Opdyke, 1992] e pode ser definido como: "*o processo de alterar um software orientado a objetos de maneira que o seu comportamento externo não seja alterado, mas sua estrutura interna seja melhorada*" [Fowler, 1999a]. As refatorações são operações normalmente documentadas por meio de linguagens de padrões[9] (assim como os padrões de projeto [GoF, 1994]), sendo que o catálogo de Fowler [Fowler, 1999a] seja talvez o trabalho mais conhecido a esse respeito. Neste trabalho, trata-se particularmente da refatoração de código-fonte, mas refatorações podem ser aplicadas em níveis mais altos de abstração [Mens et al., 2003], tais como: modelos de projeto (diagramas UML[10]), padrões (*design patterns*) e arquiteturas de *software*.

As refatorações são transformações que preservam o comportamento externo de um programa (inclusive erros), ou seja, se o programa for executado com as mesmas entradas antes e depois de uma refatoração bem sucedida as saídas deverão ser as mesmas. Quando se pratica a refatoração, o objetivo é melhorar a manutenibilidade do *software*, reestruturando o código-fonte de tal maneira que esse se torne mais reusável, mais fácil de ser compreendido e alterado, sendo portanto uma atividade de apoio para o **projeto**[11] e a **evolução** de *software*

8 Do inglês, *refactoring*.
9 Do inglês, *pattern language*.
10 *Unified Modeling Language*
11 Refere-se, nesse contexto, à atividade de se projetar uma solução para um problema, definindo as classes, suas operações e relacionamentos.

[Opdyke, 1992] [Roberts et al., 1997]. *"A idéia principal é redistribuir classes, variáveis e métodos através da hierarquia de classes para facilitar futuras adaptações e extensões"* [Mens e Tourwé, 2004].

O conceito de refatoração pode ser melhor ilustrado por meio de um exemplo do que por longas descrições textuais. Assim, um exemplo simples é apresentado a seguir.

2.4.1 Exemplo de refatoração

Considera-se que uma empresa de transportes quer calcular a soma das passagens pagas a partir de uma lista dos passageiros de um ônibus. A empresa concede descontos a determinadas categorias de passageiros: idosos pagam 70% do valor da passagem, estudantes pagam 50% e os demais passageiros, cuja categoria é denominada normal, pagam 100%.

Uma solução[12] para o problema é modelada pelas classes apresentadas na Figura 1. A classe `Passageiro` representa os passageiros do ônibus e para cada objeto dessa classe um valor (as constantes `ESTUDANTE`, `IDOSO` e `NORMAL`) é atribuído ao atributo `categoria`, indicando a qual categoria pertence o objeto e permitindo assim que seja possível deduzir a porcentagem da passagem que o passageiro deve pagar. A classe `Onibus` mantém uma lista dos passageiros e define o método `somarPassagens()` que itera pela lista considerando a categoria de cada objeto da classe `Passageiro` e calcula a soma das passagens. A classe `Simulador` apenas cria uma instância de `Onibus` e adiciona a ela algumas instâncias de `Passageiro` de categorias diferentes, chamando então o método `Onibus.somarPassagens()` e imprimindo o valor calculado.

12 Esse é um exemplo muito simples cujo único propósito é demonstrar o uso das refatorações. Não há preocupação alguma com a eficiência computacional da implementação, preocupa-se apenas com a maneira como a solução foi modelada em termos de classes, associações e operações.

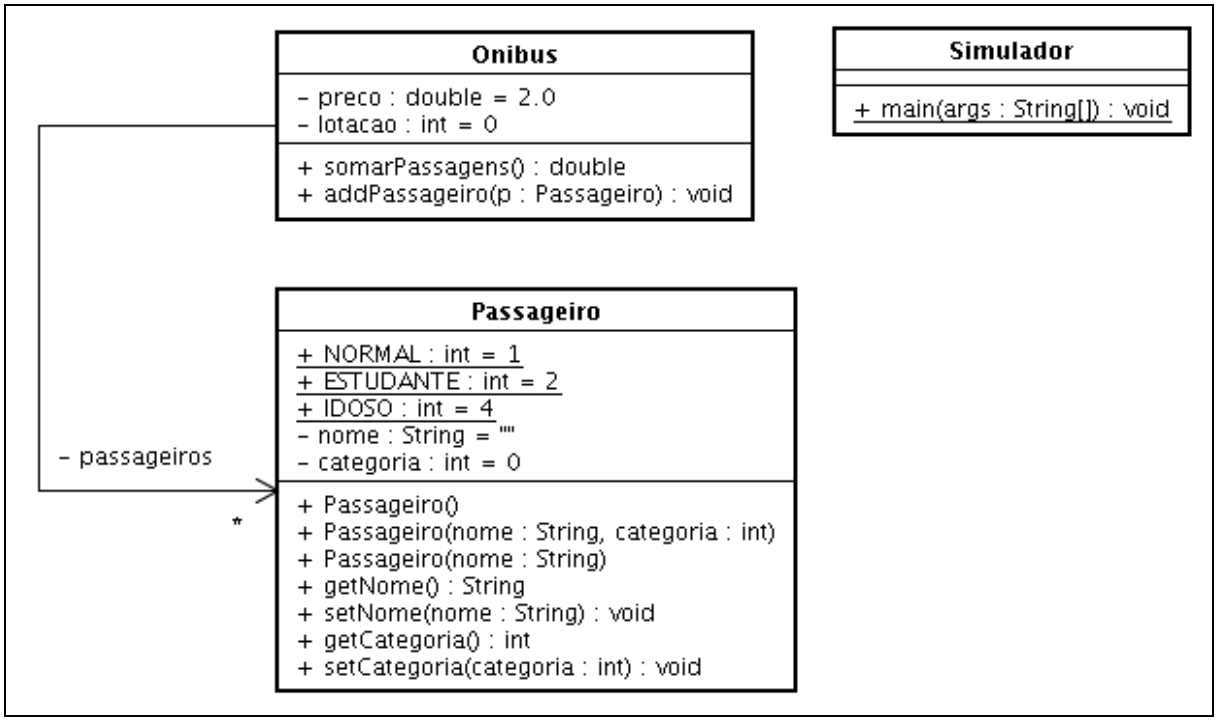

Figura 1: Diagrama de classes – solução para o problema da soma das passagens

Na Figura 2 é apresentado o código (na linguagem Java) do método `somarPassagens()`. Esse é um exemplo muito simples, mas não é difícil imaginar como ficaria esse método e a classe `Passageiro` se ao invés de 3 existissem 10 categorias ou mais: o bloco `switch` da linha 06 ficaria bem mais longo, e para cada nova categoria seria adicionada uma constante correspondente em `Passageiro`. A adição de uma nova categoria requer, portanto, a alteração de dois artefatos (esse número pode ser bem maior num sistema real), o que afeta diretamente a manutenibilidade. A classe `Onibus` acumula responsabilidades e detém praticamente toda a implementação do sistema no método `somarPassagens()`, enquanto `Passageiro` não assume nenhuma responsabilidade a não ser a de uma mera estrutura de dados. É preciso que `somarPassagens()` tenha conhecimento de cada categoria existente no sistema a fim de determinar a porcentagem correspondente e isso aumenta o acoplamento entre as classes `Onibus` e `Passageiro`.

```
01 public double somarPassagens() {
02   double total = 0;
03   double porcentagem = 0;
04
05   for (Passageiro p : passageiros) {
06     switch(p.getCategoria()) {
07       case Passageiro.ESTUDANTE:
08         porcentagem = 0.5;
09         break;
10
11       case Passageiro.IDOSO:
12         porcentagem = 0.7;
13         break;
14
15       case Passageiro.NORMAL:
16         porcentagem = 1;
17     }
18
19     total += (preco * porcentagem);
20   }
21
22   return total;
23 }
```

Figura 2: Método Onibus.somarPassagens()

Uma vez que a determinação da porcentagem a ser paga por uma dada categoria depende unicamente da própria categoria, é natural atribuir essa responsabilidade à classe `Passageiro` e fazer com que o método de `Onibus`, ao invés de determinar a categoria, solicite-a de um objeto do tipo `Passageiro`. Para realizar essa alteração, aplicam-se duas refatorações[13] [Fowler, 1999a]:

1. *Extract Method*: para isolar a determinação da porcentagem (linhas de 06 a 17 da listagem apresentada na Figura 2) em um método à parte; e

2. *Move Method*: para mover o método recém criado para a classe `Passageiro` e assim completar a alteração.

13 São usados os nomes originais das refatorações, em inglês, assim como aparecem no catálogo de Fowler.

```
01 public double somarPassagens() {
02   double total = 0;
03
04   for (Passageiro p : passageiros) {
05     total += (preco * getPorcentagem(p));
06   }
07
08   return total;
09 }
10
11 private double getPorcentagem(Passageiro p) {
12   double porcentagem = 0;
13
14   switch(p.getCategoria()) {
15     case Passageiro.ESTUDANTE:
16       porcentagem = 0.5;
17       break;
18
19     case Passageiro.IDOSO:
20       porcentagem = 0.7;
21       break;
22
23     case Passageiro.NORMAL:
24       porcentagem = 1;
25   }
26
27   return porcentagem;
28 }
```

Figura 3: Métodos somarPassagens() e o recém-extraído getPorcentagem(), ambos da classe Onibus

Observa-se na Figura 3 o resultado da primeira refatoração. Ao invés de determinar a porcentagem, o método `somarPassagens()` passou a invocar o recém-extraído `getPorcentagem()`, passando como parâmetro um objeto do tipo `Passageiro`. Essa primeira refatoração é uma preparação para mover a determinação da porcentagem para a classe `Passageiro`.

Nesse cenário, aplica-se a segunda refatoração, *Move Method*. O resultado pode ser observado na Figura 4. O método `somarPassagens()` passou a chamar `getPorcentagem()` diretamente a partir da referência do objeto do tipo `Passageiro`, não havendo mais a necessidade do parâmetro explícito, uma vez que `getPorcentagem()` passou a pertencer à classe `Passageiro`. Essa já é a versão final de `somarPassagens()`. Após a migração, `getPorcentagem()` teve sua visibilidade alterada para pública e perdeu o parâmetro explícito.

```
01 /* classe Onibus */
02 public double somarPassagens() {
03   double total = 0;
04
05   for (Passageiro p : passageiros) {
06     total += (preco * p.getPorcentagem());
07   }
08
09   return total;
10 }


01 /* classe Passageiro */
02 public double getPorcentagem() {
03   double porcentagem = 0;
04
05   switch(getCategoria()) {
06     case Passageiro.ESTUDANTE:
07       porcentagem = 0.5;
08       break;
09
10     case Passageiro.IDOSO:
11       porcentagem = 0.7;
12       break;
13
14     case Passageiro.NORMAL:
15       porcentagem = 1;
16   }
17
18   return porcentagem;
19 }
```

Figura 4: Método getPorcentagem() movido para a classe Passageiro

Após essas duas refatorações, foi resolvido o problema do acoplamento entre as classes `Onibus` e `Passageiro`. Contudo, a determinação da porcentagem ainda depende de uma sentença `switch` e de uma variável de tipo[14] em conjunto com uma lista de constantes. Essa abordagem é típica do paradigma da programação estruturada e apresenta os problemas mencionados no início desse exemplo: a complexidade da combinação `switch` mais constantes e a necessidade de se manter a lista de constantes. Linguagens que seguem o paradigma orientado a objetos oferecem recursos que simplificam a implementação de soluções como a implementada pelo método `getPorcentagem()` da Figura 4.

A uso do polimorfismo é uma solução natural para o problema de se executar uma operação ou outra dependendo do tipo (classe) de um objeto. A refatoração *Replace*

14 Refere-se por variável de tipo às variáveis que são usadas em conjunto com estruturas de decisão para escolher qual operação deve ser executada com base no valor (o tipo) da variável.

Conditional with Polymorphism (tradução livre: “Substituir estrutura condicional por polimorfismo”) trata, como o nome diz, da substituição de estruturas condicionais por polimorfismo (o que pode implicar na criação de novas classes). Essa refatoração foi aplicada ao método `getPorcentagem()` da classe `Passageiro` – que passou a representar a categoria normal – eliminando a sentença `switch` e tornando-o muito mais simples, como pode ser visto na Figura 5. A aplicação da refatoração resultou na substituição das constantes de tipo por novas classes (`Estudante` e `Idoso`), cada uma com a sua versão do `getPorcentagem()`, como pode ser visto na Figura 6.

```
1 public double getPorcentagem() {
2   return 1;
3 }
```

Figura 5: Método getPorcentagem() da classe Passageiro

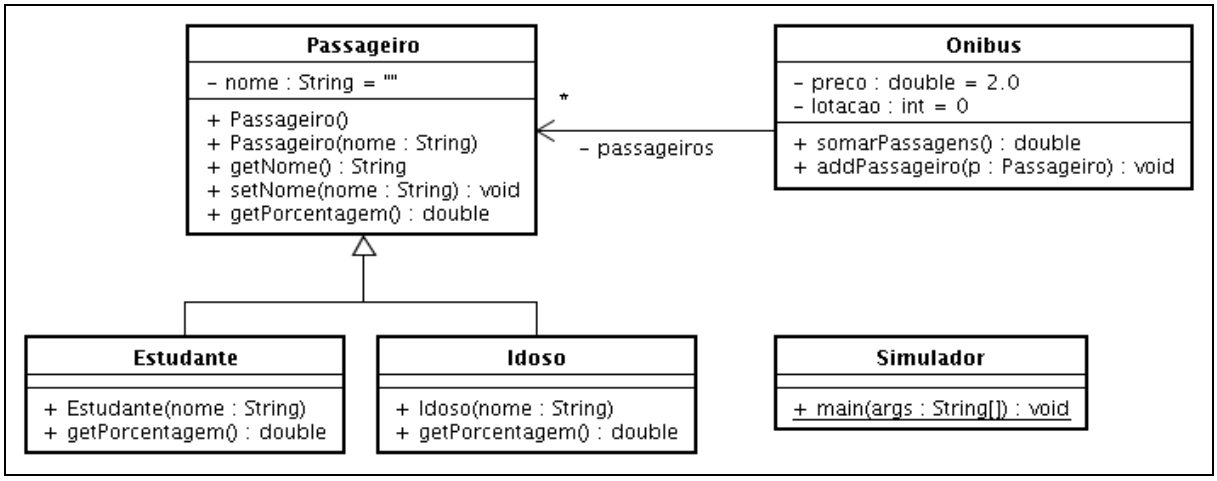

Figura 6: Versão final da solução para o problema da soma das passagens

Na Figura 6 pode ser observado o diagrama de classes após a aplicação das três refatorações (*Extract Method*, *Move Method* e *Replace Conditional with Polymorphism*) à primeira versão da solução para o problema das passagens. Após essa refatoração, a lista de passageiros da classe `Onibus` passou a armazenar instâncias das classes `Passageiro`, `Estudante` e `Idoso` e o polimorfismo passou a garantir que o método `somarPassagens()` receba os valores corretos ao invocar `getPorcentagem()` em instâncias de passageiro, visto que, para a classe `Estudante`, este retorna 0,5 e para `Idoso`, 0,7. Observa-se ainda que o

número de classes aumentou, mas que as classes tornaram-se bem mais simples do que antes. As constantes e a variável de tipo de `Passageiro` foram dispensadas. O método `somarPassagens()` passou a ser um simples laço `for` que repete uma multiplicação e uma soma, e a sentença `switch` desapareceu. A implementação do método `getPorcentagem()` passou a ser o simples retorno de um valor.

Após as refatorações a manutenibilidade melhorou consideravelmente. Para a introdução de uma nova categoria, por exemplo, basta apenas que uma nova classe que estenda a classe `Passageiro` seja criada e que essa sobrescreva o método `getPorcentagem()`, sem que nenhuma outra modificação seja necessária.

É importante salientar que após cada refatoração aplicada o sistema podia ser compilado e executado sem nenhuma alteração em seu comportamento externo. Ao final das três etapas, o resultado da execução do sistema continuou sendo o mesmo que era no início, de maneira que apenas sua estrutura interna foi melhorada.

Para garantir a consistência do sistema, ou seja, a preservação do comportamento externo, depois da aplicação das refatorações, essas devem ser executadas em pequenos passos intercalados com compilações e testes, de maneira que, se algum dos passos introduzir um erro, seja fácil isolar a alteração responsável e recuperar o sistema [Fowler, 1999a].

2.5 Considerações Finais

A qualidade de *software* mostra-se ser algo mais complexo do que a qualidade de produtos manufaturados. Para obtê-la faz-se necessário seguir um processo de desenvolvimento bem definido, normalmente garantindo que padrões de qualidade pré-selecionados e estabelecidos sejam seguidos.

A manutenção preventiva é uma atividade que proporciona, além de outros benefícios, a

garantia de que o produto de trabalho não se desvie dos padrões de qualidade. No âmbito dos sistemas orientados a objetos, a manutenção preventiva pode se dar pela aplicação das refatorações, que são reestruturações documentadas em linguagens de padrões e que proporcionam o aumento de compreensibilidade e a redução de complexidade de estruturas de *software*, de maneira que a manutenibilidade é elevada. Assim, a atividade de manutenção preventiva, implementada por meio das refatorações, deve ser empregada para corrigir desvios dos padrões de qualidade do produto de *software*.

Para que os desvios mencionados possam ser corrigidos é necessário que sejam antes detectados. Isso se dá pela execução da atividade de controle de qualidade[15] por meio de basicamente duas abordagens: as revisões e as análises automáticas. A **inspeção de *software*** é um tipo particular de revisão [Mafra e Travassos, 2005] e será mais detalhada no capítulo 3, assim como as métricas, as quais são fundamentais para a execução de análises automáticas.

15 Ver sub-seção 2.2.3.

Capítulo 3 - Inspeção e Métricas de *Software*

3.1 Considerações Iniciais

Este capítulo é composto por duas partes nas quais são abordados os temas inspeção e métricas de *software*. A primeira parte é iniciada com uma revisão sobre os conceitos gerais de inspeção de *software*, tratando-se, em seguida, da técnica de leitura PBR (*Perspective-Based Reading*, ou, Leitura Baseada em Perspectivas). Na segunda parte trata-se de métricas de produto de *software*, as quais são meio de se avaliar automaticamente a qualidade de um produto.

3.2 Inspeção de *Software*

Uma prática comum entre os que escrevem trabalhos acadêmicos (como este) é pedir a alguém – normalmente um orientador ou colega – que revise o texto, esperando que sejam encontrados pontos falhos, ou que mereçam alguma melhoria, tanto no trabalho desenvolvido como na própria escrita. O objetivo, ao se executar esse processo de revisão, não é outro senão melhorar a qualidade desses produtos (o trabalho e a escrita), ou, adotando termos mais específicos, executar o controle da qualidade desses produtos.

Revisar é, portanto, examinar algo com o objetivo de encontrar falhas[16], para então corrigi-las e assim melhorar a qualidade do produto examinado. Dessa maneira, uma conversa, durante uma pausa para o café, em que alguém expõe problemas técnicos que está enfrentando a um amigo e recebe suas opiniões é uma revisão [Pressman, 2005], apesar de informal.

16 Entenda-se por falha não somente defeitos que levem a resultados incorretos, mas também pontos que, apesar de corretos, possam ainda ser melhorados.

Revisões técnicas formais são um recurso largamente usado em desenvolvimento de *software* [Mafra e Travassos, 2005]. A diferença entre esse tipo de revisão e o citado no exemplo anterior é que as revisões técnicas formais devem ser planejadas e devem seguir um processo formal [Pressman, 2005]. As revisões podem ter diversos objetivos, tais como tratar do cronograma ou custos de um projeto, ou detectar defeitos em produtos de trabalho [Sommerville, 2001]. As revisões cujo objetivo é a detecção de defeitos são chamadas de **inspeções** [Fagan, 1976].

Tanto as inspeções quanto os testes de *software* são técnicas usadas no processo verificação e validação (V &V) – um processo que permeia o ciclo de vida de um *software* e cujo objetivo é garantir que o *software* esteja em conformidade com as especificações (verificação) e que satisfaça às necessidades do cliente (validação) [Sommerville, 2001]. Estudos empíricos comprovaram que as inspeções são eficazes e mais baratas do que os testes, sendo que mais de 60% dos defeitos de um artefato podem ser detectados por inspeções [Laitenberger e Atkinson, 1999] [Sommerville, 2001]. Contudo, isso não significa que inspeções e testes são técnicas concorrentes, ao contrário, são complementares. Por poderem ser executadas desde o início do processo de desenvolvimento, as inspeções provêem contribuições às próximas atividades do processo, uma vez que um artefato gerado a partir de outro (um documento de requisitos, por exemplo) inspecionado e corrigido tende a apresentar menos defeitos. Assim, a execução de inspeções contribui para que haja menos erros a serem detectados na fase de testes, o que, inclusive, acaba por afetar positivamente o cronograma do projeto [Fagan, 1976]. Inspeções podem ser aplicadas em diversos tipos de artefato [Mafra e Travassos, 2005] como documentos de requisitos, projetos de interface com o usuário, diagramas, código-fonte e, inclusive, planos de teste.

A inspeção é um processo rigoroso e bem definido para a detecção de defeitos em artefatos de *software* [Mafra e Travassos, 2005], que costuma ser conduzido por pequenos grupos de

pessoas assumindo papéis claramente definidos [Sommerville, 2001]. No trabalho em que foram propostas as inspeções de *software* [Fagan, 1976], aconselha-se que o grupo de inspeção seja formado por quatro pessoas assumindo os seguintes papéis:

- *Moderador*: conduz a reunião de inspeção, sendo responsável pelo seu agendamento, alocação de recursos, por garantir que o artefato inspecionado esteja completo e por manter o foco da reunião. É responsabilidade do moderador prover os resultados da reunião às partes interessadas e acompanhar os trabalhos de correção dos defeitos encontrados.
- *Projetista*: o responsável pelo projeto (ou modelagem) do sistema.
- *Programador*: responsável pela implementação do projeto criado pelo projetista.
- *Testador*: o responsável pelos casos de teste que serão aplicados ao artefato em questão.

Nesse mesmo trabalho [Fagan, 1976], são descritas as atividades do processo de inspeção:

1. *Apresentação (realizada em grupo)*: o contexto no qual se insere o artefato a ser inspecionado é apresentado ao grupo. No caso de um programa, por exemplo, o projetista descreve quais são os propósitos do mesmo e fornece aos participantes a documentação relevante para a inspeção (diagramas, pseudo-código, especificações, código-fonte).
2. *Preparação (individual)*: cada participante estuda a documentação que recebeu a respeito do artefato a ser inspecionado e também a documentação a respeito de erros comumente encontrados. Nessa fase defeitos já podem ser identificados.
3. *Reunião de inspeção*: na reunião, o moderador escolhe um leitor que descreve o artefato inspecionado. Enquanto o leitor apresenta o artefato, os outros participantes

levantam suas dúvidas, identificadas na fase 2. É responsabilidade do moderador permitir que as discussões prossigam apenas até o ponto em que um defeito é identificado, e então anotar o defeito. Para que a reunião seja mais produtiva, não são procuradas as soluções para os problemas encontrados, apenas os defeitos são anotados. Se a solução for óbvia, o moderador também a anota juntamente com o defeito. O autor [Fagan, 1976] aconselha que, de acordo com sua experiência, as reuniões não ultrapassem duas horas de duração. Após esse tempo a produtividade tende a cair.

4. *Correção de defeitos*: os defeitos listados na reunião de inspeção são, então, corrigidos pelo autor do artefato inspecionado.

5. *Acompanhamento*: cabe ao moderador fazer o acompanhamento das correções, podendo ou não organizar uma nova inspeção para validá-las. O autor [Fagan, 1976] sugere que uma nova inspeção deve ser organizada quando mais de 5% do trabalho inspecionado passar por correções.

Em seu trabalho, Fagan [1976] considera primordialmente a inspeção de código-fonte mas, como mencionado anteriormente, vários tipos de artefato podem ser inspecionados. Em trabalhos posteriores foram sugeridas várias alterações no processo de inspeção e nos papéis dos participantes [Mafra e Travassos, 2005] [Sommerville, 2001] [Shull et al., 2000] [Laitenberger e DeBaud, 1998], contudo, apesar das variações propostas, as atividades descritas por Fagan continuam valendo como um modelo geral do processo de inspeção.

Uma conseqüência imediata da adoção das inspeções é a disseminação de conhecimento (sobre o projeto e também conhecimento técnico), visto que profissionais menos experientes entram em contato com outros mais experientes nas reuniões de inspeção e assim o conhecimento é disseminado [Shull et al., 2000] [Fagan, 1976].

É possível notar que a detecção de defeitos concentra-se nas atividades 2 e 3 (listadas anteriormente) e que dependem fortemente dos inspetores. Se nenhum tipo de técnica de leitura for utilizado pelos inspetores na atividade 2, chamada em trabalhos mais atuais [Shull et al., 2000] de "fase de detecção de defeitos", a identificação dos defeitos dependerá unicamente de seu conhecimento e julgamento pessoal. Já em seu trabalho, Fagan [1976], aponta que é necessário induzir as pessoas a encontrarem erros para que melhores resultados sejam obtidos. O autor propõe que os defeitos encontrados em inspeções anteriores sejam classificados por freqüência e custo, e que então seja elaborada uma lista de perguntas (*checklist*) com aqueles que ocorrem com maior freqüência e com os que têm o maior custo de correção para então ser usada como guia pelos inspetores na leitura dos artefatos.

Durante vários anos (pelo menos até o início da década de 90) as abordagens de leitura mais utilizadas foram a *ad hoc*, na qual nenhum auxílio é fornecido ao inspetor e depende-se unicamente do seu julgamento pessoal, e a *checklist* que, apesar oferecer aos inspetores um apoio mais concreto do que a *ad hoc*, também apresenta deficiências como [Mafra e Travassos, 2005] [Laitenberger e DeBaud, 1998]:

- perguntas gerais e pouco adaptadas ao contexto, de maneira que a *checklist* não colabora para que o inspetor compreenda o artefato inspecionado; e
- não são fornecidas instruções concretas de como a *checklist* deve ser usada.

Para suprir essas necessidades foram propostas técnicas de leitura. "*Uma técnica de leitura é caracterizada por uma série de passos para a análise individual de um artefato de software de forma a permitir a extração do entendimento necessário para a realização de determinada tarefa*" [Mafra e Travassos, 2005]. Entre essas técnicas está a Leitura Baseada em Perspectivas (PBR – *Perspective-Based Reading*) [Basili et. al, 1996] [Shull et al., 2000], à qual é dedicada a sub-seção a seguir.

3.2.1 Leitura Baseada em Perspectivas (PBR)

A PBR foi inicialmente proposta como uma técnica de leitura para documentos de requisitos [Basili et. al, 1996], mas, devido ao fato de ser uma técnica extensível, é possível usá-la para a leitura de outros tipos de artefato. De fato, há relatos do uso da PBR para a inspeção de artefatos da UML e de código-fonte [Laitenberger e Atkinson, 1999]. Neste texto, o artefato documento de requisitos é usado como um exemplo para a apresentação dos conceitos da Leitura Baseada em Perspectivas (PBR), contudo, deve-se ter em mente que esses conceitos são aplicáveis a outros tipos de artefato de *software*.

O princípio da Leitura Baseada em Perspectivas é o de que o trabalho de um inspetor é mais eficaz se ele mantiver sua atenção em somente um aspecto do artefato sob inspeção – originalmente o documento de requisitos [Shull et al., 2000]. Ou seja, se um inspetor estiver interessado em detectar defeitos que causem impacto na elaboração de casos de testes – como a ausência de parâmetros de entrada, ou saída esperada para o programa mal especificada – seu trabalho será mais eficaz (encontrará mais defeitos) se ele realizar a leitura do documento de requisitos somente em busca de defeitos relacionados a testes do que se estivesse analisando o artefato em busca de defeitos de todos os tipos.

A PBR baseia-se em revisões individuais feitas pelos usuários do artefato inspecionado, cada um sob seu ponto de vista (ou perspectiva), de maneira que a união dos resultados de cada revisão individual resulte na cobertura de todos os aspectos de interesse do artefato inspecionado [Shull et al., 2000]. Para a aplicação da PBR é preciso, portanto, identificar quais são os usos do artefato inspecionado, para que assim possam ser determinadas as perspectivas sob quais será feita a leitura [Mafra e Travassos, 2005] [Shull et al., 2000]. Se um documento de requisitos é usado por um testador para criar o plano de testes do sistema, por um projetista como base para a elaboração do projeto do *software* e pelo usuário (ou

cliente) para garantir que as funcionalidades desejadas para o sistema estejam todas ali expressas, fica claro que esse deve ser inspecionado respectivamente sob as perspectivas de leitura do projetista, do testador e do usuário (Figura 7).

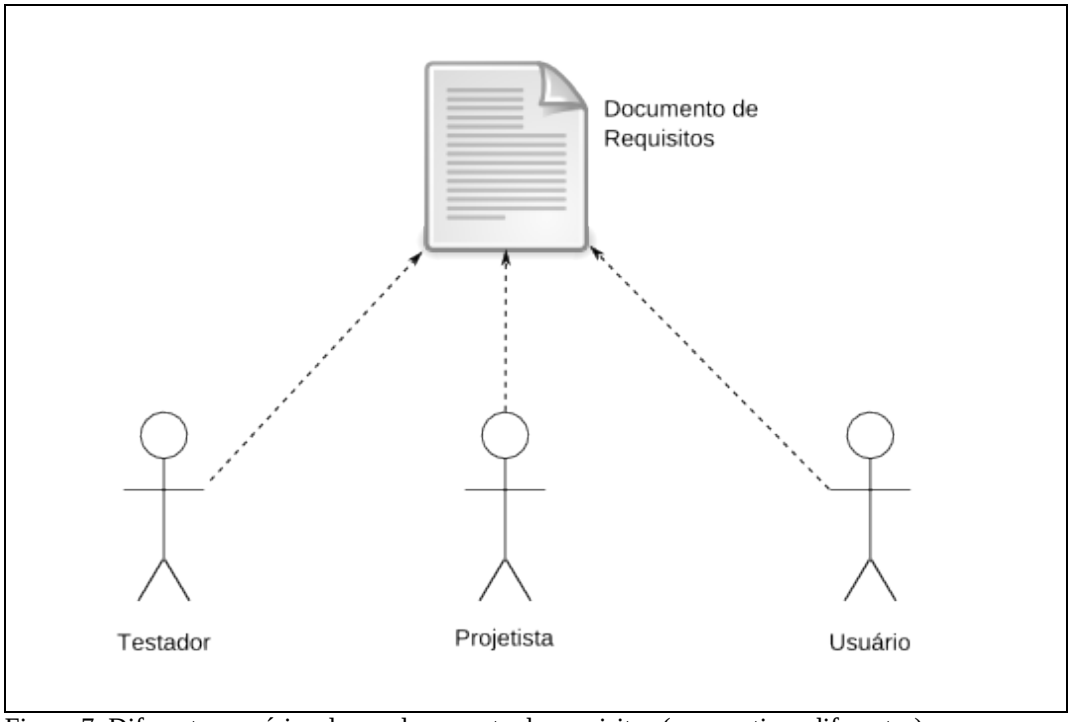

Figura 7: Diferentes usuários de um documento de requisitos (perspectivas diferentes)

Numa inspeção, cada inspetor assume uma perspectiva sob a qual fará a leitura do artefato em busca de defeitos. São fornecidos aos inspetores cenários – específicos para cada perspectiva – contendo instruções de como proceder com a leitura. O inspetor deve criar representações de alto nível dos produtos de trabalho que seriam criados pelos usuários do artefato. Enquanto cria as representações, o inspetor responde às perguntas que são listadas no cenário assim como as instruções de como e quais produtos de trabalho criar [Laitenberger e DeBaud, 1998]. Por exemplo, o cenário da perspectiva do testador orienta o inspetor a criar alguns planos de teste baseando-se no documento de requisitos e então responder às perguntas. A intenção é que, ao se criar os produtos de trabalho, dúvidas surjam e defeitos acabem sendo postos em evidência. O conjunto de perguntas é elaborado com base em tipos de defeitos já conhecidos, também de forma a evidenciá-los [Mafra e Travassos, 2005]. A

PBR prevê a melhoria do conjunto de perguntas por meio de alterações sugeridas pelos inspetores, que podem sugerir a reformulação, remoção ou adição de perguntas [Shull et al., 2000].

São apontados como benefícios proporcionados pela PBR [Shull et al., 2000]:

- permitir determinar se um determinado uso é ou não viável, ao identificar explicitamente os usos de um artefato;
- fazer com que o inspetor mantenha o foco em certos tipos de defeito, não desviando sua atenção ao tentar identificar todos os defeitos, de todos os tipos;
- ser orientada a objetivos e adaptável a situações específicas, permitindo que perspectivas sejam selecionadas de acordo com a necessidade em cada situação;
- poder ser ensinada por meio de treinamentos, uma vez que não depende unicamente dos conhecimentos do inspetor.

A análise de 14 estudos nos quais a PBR era comparada com *checklist* ou com leitura *ad hoc* demonstrou que em 57% desses a PBR mostrou-se mais eficaz [Mafra e Travassos, 2005]; os autores atribuem isso, assim como a diminuição da diferença de desempenho entre os inspetores mais experientes e os menos experientes, à “orientação ativa”, nome que dão às instruções providas por meio dos cenários.

Em outro trabalho [Laitenberger e Atkinson, 1999], é argumentado que, apesar do sucesso da abordagem por perspectivas da PBR, essa técnica ainda apresenta limitações quando se considera a inspeção no contexto do desenvolvimento de *software* orientado a objetos. Contudo, das quatro limitações apresentadas pelos autores, apenas uma é relevante para os propósitos deste trabalho e será aqui tratada.

Nas publicações originais [Basili et. al, 1996] [Shull et al., 2000] a técnica PBR pode ser

definida como: "*ler um artefato de software da perspectiva dos vários clientes desses artefatos com o propósito de identificar defeitos*" [Laitenberger e Atkinson, 1999]. Segundo os autores, uma das limitações é o uso do termo "defeito", que pode ser inapropriado em determinadas situações como no caso da inspeção de um artefato do ponto de vista de um mantenedor, em que o artefato apresente inconformidades com os padrões de qualidade. O termo defeito não é apropriado nessa situação, pois apesar da inconformidade, não há resultados incorretos sendo gerados. Assim, os autores [Laitenberger e Atkinson, 1999] sugerem a substituição de "defeito" pelo termo "falha", o qual definem como "*qualquer propriedade de um artefato que o impeça de satisfazer aos seus requisitos de qualidade*", definição mais abrangente, que contempla, inclusive, a definição de defeito.

Nesse mesmo trabalho [Laitenberger e Atkinson, 1999], também são propostas alterações para o modelo de cenários da PBR, sendo que uma alteração interessante é a inclusão de uma seção de introdução, descrevendo qual é o interesse do usuário (perspectiva) no artefato sob inspeção e quais são os atributos de qualidade relevantes para aquele artefato naquela perspectiva.

A seguir, na seção 3.3, trata-se das métricas de *software* que, assim como as inspeções, são úteis para a implementação do controle de qualidade.

3.3 Métricas de *Software*

Com exceção de avaliações subjetivas, baseadas em critérios como a opinião individual, a maioria dos processos de avaliação está fundamentada na quantificação, na medição. Isso é verdade quando se quer avaliar quanta água cabe em um determinado recipiente (capacidade de "*x litros*" e não de "*um pouco d'água*"), assim como também é verdade quando se quer medir o desempenho de um aluno que, apesar de se tratar de uma grandeza de natureza diferente do volume de um recipiente que é diretamente mensurável por números, cria-se uma

maneira de quantificar o seu aprendizado (as notas) e um instrumento para medi-lo (as provas).

No que concerne ao que se deseja medir, as métricas de *software* são semelhantes às notas: "*normalmente é impossível medir atributos de qualidade de software diretamente*" [Sommerville, 2001], assim como o é medir o quanto um aluno aprendeu sem lançar mão de uma métrica (normalmente exercícios que valem pontos cujos acertos indicam que o aluno domina determinado tópico). Uma vez que não é possível medir diretamente um atributo externo do *software*, como a manutenibilidade, é necessário relacioná-lo a um ou mais atributos internos mensuráveis e então analisar o comportamento dessas medidas para que conclusões possam ser tiradas a respeito do atributo de interesse. Essa idéia é ilustrada na Figura 8, na qual observa-se que a manutenibilidade, impossível de ser medida diretamente, está relacionada a atributos mensuráveis como o número de parâmetros dos métodos.

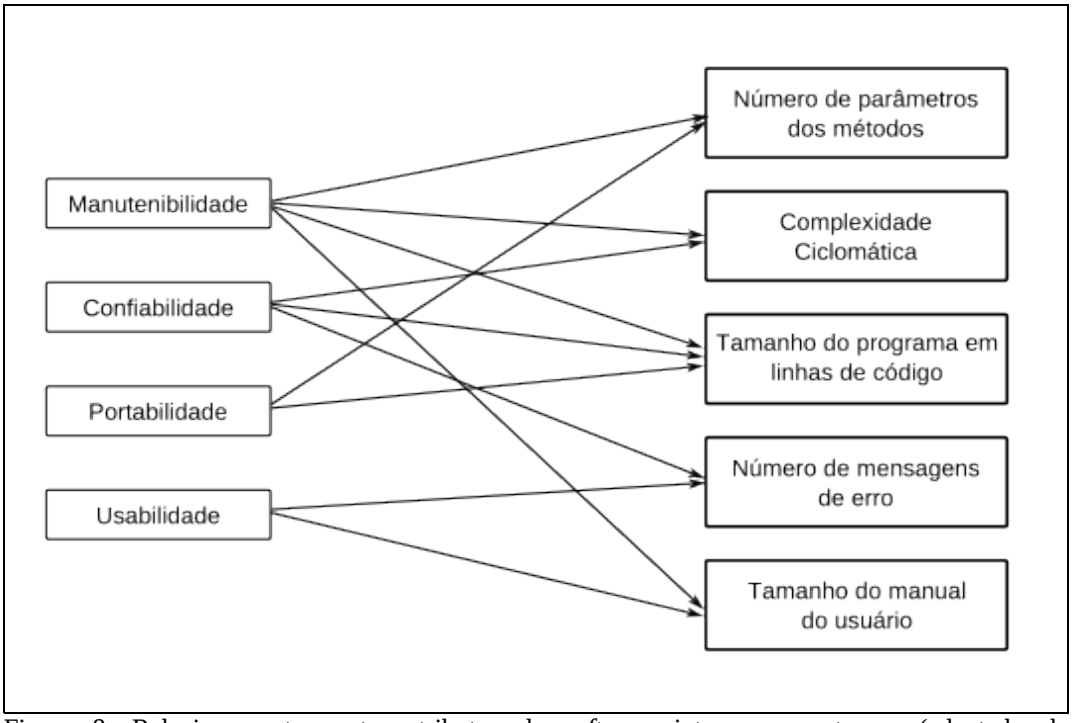

Figura 8: Relacionamentos entre atributos de *software* internos e externos (adaptado de [Sommerville, 2001])

As métricas de *software* fornecem uma base quantitativa para a avaliação da qualidade sob diversos pontos de vista; elas são usadas em diferentes escopos: processos empresariais,

projetos e produtos de *software*. Dentro da categoria produtos, as métricas ainda podem ser categorizadas como de análise, de modelagem, de código-fonte e de testes [Pressman, 2005]. Esse trabalho está focado nas métricas de produto, particularmente nas categorias modelagem e código-fonte no âmbito da orientação a objetos (OO).

3.3.1 Métricas Orientadas a Objetos

Uma vez que a classe é a unidade central dos sistemas OO, as métricas nesse paradigma a tomam como principal ponto de referência, procurando-se quantificar as relações entre classes.

O trabalho de Chidamber e Kemerer (CK) [Chidamber e Kemerer, 1994] é um dos mais citados no âmbito das métricas OO. Eles definiram 6 métricas baseadas no relacionamento entre classes fundamentadas em princípios teóricos da filosofia e as validaram com dois estudos de caso de aplicações por duas empresas diferentes. Num trabalho mais recente, a eficácia das métricas CK foi comprovada para aplicações de análise de defeitos de *software* [Subramanyam e Krishnan, 2003]. As métricas desenvolvidas foram:

- *Weighted Methods per Class* (WMC): somatório, ponderado pela complexidade, do número de métodos de uma classe. Essa métrica pode funcionar como um indicativo de tempo e esforço requerido para desenvolver e manter a classe, do impacto causado nas “classes filhas”, uma vez que um maior número de métodos será herdado, e de quanto a classe é dependente da aplicação, o que limita o reuso.

- *Depth of Inheritance Tree* (DIT): o comprimento máximo de uma classe (nó) até a super-classe mais generalizada (raiz). Essa métrica pode funcionar como um indicativo da complexidade de se predizer o comportamento da classe – uma vez que quanto mais super-classes ela tiver mais métodos terá herdado – e da complexidade da modelagem.

- *Number of Children* (NOC, neste trabalho chamada de NSC): número de subclasses imediatas. Essa métrica pode indicar tanto a presença adequada do reuso, uma vez que a herança é uma forma de reuso, quanto o uso impróprio do mecanismo de herança, pois um número muito grande de subclasses pode significar a presença do uso indevido de herança.

- *Coupling Between Objects* (CBO): contagem do número de outras classes as quais a classe está acoplada. Quanto maior o acoplamento, mais sensível a alterações em outras partes do sistema é a classe, o que torna complicada a manutenção.

- *Response For a Class* (RFC): número de métodos que podem ser potencialmente executados em resposta a uma mensagem (chamada a um método) recebida pela classe. Quanto maior o número de métodos disparados, maior a complexidade da classe.

- *Lack of Cohesion in Methods* (LCOM): contagem do número de pares de métodos cuja similaridade é zero menos a contagem do número de pares cuja similaridade não é zero. Valores altos indicam que a classe está fazendo muitas coisas diferentes e talvez seja necessário dividi-la em duas, uma vez que a baixa coesão aumenta a complexidade da classe.

Lorenz e Kidd também propuseram métricas OO [Pressman, 2005]; duas delas são:

- *Class Size* (CS): o tamanho da classe pode ser medido por contagem do número de métodos ou do número de variáveis de instância, tantos os locais como os herdados. Valores altos indicam que a classe está com responsabilidades demais, assim como a WMC (métricas CK).

- *Number of operations added by a subclass* (NOA): número de métodos adicionados

em uma subclasse. Valores altos indicam que a classe está fugindo da abstração imposta pela superclasse.

As métricas de *software* têm sido utilizadas com sucesso em várias aplicações [Carneiro, 2003] [Subramanyam e Krishnan, 2003] [Simon et al., 2001] [Kataoka et al., 2002] [Demeyer et al., 2000] [Cartwright e Shepperd, 2000] [Binkley e Schach, 1998] e, portanto, têm seu valor e aplicabilidade comprovados.

3.4 Considerações Finais

As inspeções são um meio eficaz de se detectar defeitos em artefatos de *software* que pode ser aplicado desde os estágios iniciais do processo de desenvolvimento. Como uma abordagem para o controle de qualidade, as inspeções apresentam a vantagem de prover às atividades seguintes do processo de desenvolvimento entradas de mais qualidade, fazendo com que menos defeitos sejam propagados de uma atividade para outra.

Para que a detecção de defeitos não dependa unicamente do julgamento pessoal dos inspetores e também para auxiliá-los nessa tarefa, foram desenvolvidas as técnicas de leitura, das quais a Leitura Baseada em Perspectivas (PBR) é uma representante. Duas características interessantes da PBR são a sua adaptabilidade a diferentes situações, o que permite que sejam usadas perspectivas diferentes de acordo com a necessidade; e a possibilidade de ser ensinada por meio de treinamentos, o que a torna pouco dependente da experiência dos inspetores.

Assim como as inspeções, as avaliações automáticas são uma abordagem para a implementação do controle de qualidade. Avaliações automáticas são implementadas por meio de métricas de *software*, pelas quais é possível avaliar numericamente atributos não diretamente mensuráveis do *software*, como a manutenibilidade.

A seguir, no capítulo 4, trata-se de um estudo a respeito de falhas de manutenibilidade em

software orientado a objetos e de seu relacionamento com métricas de *software*.

Capítulo 4 - Falhas de manutenibilidade em *software* orientado a objetos e métricas de produto de *software*

4.1 Considerações Iniciais

Trata-se neste capítulo de um estudo a respeito das falhas de manutenibilidade de *software* orientado a objetos, cujo propósito é a identificação das falhas mais freqüentes e de sua relação com métricas de *software*, além do reconhecimento de evidências que indiquem a ocorrência dessas falhas. Inicia-se por uma contextualização do estudo, seguida pelas descrições das análises das falhas e métricas, e seus respectivos resultados.

4.2 Visão Geral

Adotando-se a definição de falha apresentada na sub-seção 3.2.1, pode-se definir "falha de manutenibilidade de *software* orientado a objetos" – desse ponto em diante chamada apenas de "falha de manutenibilidade" ou "falha" – como: *qualquer propriedade de um artefato de software orientado a objetos que comprometa a sua manutenibilidade*.

Neste estudo, foram analisados *software*s desenvolvidos na linguagem Java [Java, 2007], segundo o paradigma orientado a objetos, cujas fontes foram repositórios de *software* livre – como *SourceForge*[17] e *Google Code*[18] – e trabalhos de alunos de graduação do Instituto de Ciências Matemáticas e de Computação (ICMC) da Universidade de São Paulo (USP). Os *software*s tinham diversos propósitos, entre eles a automação comercial, o gerenciamento de acervo bibliográfico e a transferência de arquivos via rede.

17 http://sourceforge.net
18 http://code.google.com

As falhas estudadas foram, principalmente, as indicadas por Fowler em seu livro sobre refatorações [Fowler, 1999a], as quais são originalmente chamadas pelo autor de *bad smells*[19]. Contudo, foram detectadas outras falhas além dessas. Também foram identificados casos particulares dessas falhas. Houve casos em que a definição original não foi abrangente o suficiente para descrever falhas semelhantes. Nesses casos, essas definições foram reformuladas de modo a se tornarem mais abrangentes.

A primeira etapa do estudo foi a seleção dos *software*s a serem analisados em busca das falhas de manutenibilidade. Uma vez que é necessário compreender o *software* para que as falhas possam ser identificadas, foram escolhidos *software*s de pequeno porte em detrimento de outras escolhas possíveis como servidores de aplicação e sistemas gerenciadores de bases de dados, que são *software*s de maior porte. Essa política permitiu que um número maior de *software*s fosse analisado, visto que o esforço necessário para compreender cada um era menor, o que privilegiou a diversidade de autores e propósitos dos programas.

Buscou-se identificar, além das falhas mais freqüentes e de suas evidências, uma possível relação entre os valores de métricas de *software* orientado a objetos e a eliminação de uma falha, de maneira que essa propriedade pudesse ser usada para validar a correção de falhas de manutenibilidade automaticamente. Dessa maneira, foi preciso manter o histórico das versões do programa a cada falha encontrada e eliminada, assim como documentar qual foi a falha encontrada e quais foram as transformações aplicadas para eliminá-la para posterior análise. Essa necessidade levou à sistematização de um processo de análise apoiado por um sistema de controle de versões[20].

4.3 Análise das Falhas de Manutenibilidade

19 Em português, “maus cheiros”. Esse termo se deve a uma metáfora utilizada pelo autor.
20 Como sistema de controle de versões foi utilizado o *software* Subversion ou SVN (http://subversion.tigris.org/).

O processo de análise das falhas é ilustrado pelo diagrama de atividades da Figura 9.

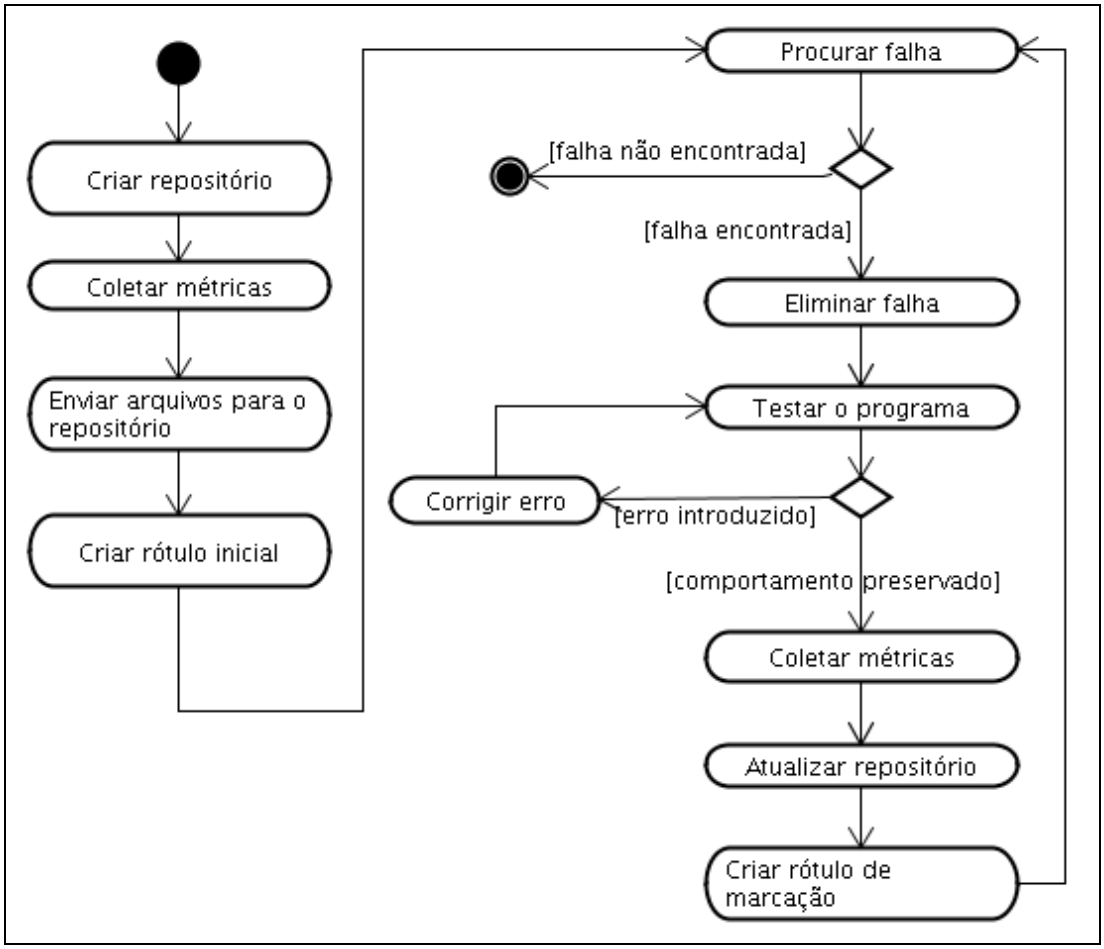

Figura 9: Processo de análise de falhas

Esse processo foi executado para cada um dos *software*s analisados. Inicia-se pela criação de um repositório de controle de versões para manter o histórico do programa ao longo das alterações. Coleta-se então as métricas do programa em seu estado inicial, as quais são gravadas num arquivo. Todos os arquivos do programa, incluindo o das métricas, são enviados para o repositório e é criado um rótulo[21] para marcar o estado inicial do programa.

Após esses passos iniciais de preparação do repositório inicia-se a busca por falhas (lado direito do diagrama de atividades da Figura 9). São buscadas, como mencionado anteriormente, principalmente falhas documentadas na literatura [Fowler, 1999a], todavia não se restringindo apenas a essas. Dois exemplos de falhas são a violação de encapsulamento e o

21 Em inglês, chama-se de *tag*, as marcações de determinados pontos no repositório.

mau uso da herança, na qual atributos idênticos são repetidos em várias classes semelhantes. As falhas encontradas serão discutidas detalhadamente mais adiante.

Para cada falha encontrada, são aplicadas transformações no código-fonte com o objetivo de eliminá-las. Essas transformações são as chamadas refatorações, que, como discutido na sub-seção 2.4, preservam o comportamento externo do programa. Podem ser necessárias várias refatorações consecutivas até que uma falha seja totalmente eliminada (ver exemplo na sub-seção 2.4.1). Em seguida, testes são realizados para garantir que o comportamento externo do programa foi realmente preservado após as refatorações. Se for detectada a introdução de algum erro, isto é, alguma mudança no comportamento externo, esse deve ser corrigido de maneira que o comportamento do programa continue sendo o mesmo que antes da eliminação da falha. Caso contrário, se após os testes for constatado que o comportamento foi preservado, métricas dessa nova versão do programa são coletadas e gravadas em arquivo (o mesmo arquivo de métricas do momento em que o repositório foi criado, o que gera uma nova versão desse arquivo). O repositório é então atualizado com as novas versões dos arquivos e um rótulo é criado para marcar esse ponto em que uma falha acabou de ser eliminada. Uma breve documentação a respeito da falha recém-eliminada contendo seu nome, artefatos envolvidos (classes, métodos, etc), evidências de sua ocorrência e refatorações aplicadas é armazenada no repositório como um comentário do rótulo (Figura 10).

Falha:
Mau uso de herança

Artefatos envolvidos:
classes Policial e HabitanteNormal

Evidências:
Conjunto de atributos iguais nas classes Policial e HabitanteNormal sem nenhuma relação de herança.

Refatorações aplicadas:
Move Field
Extract Class

O conjunto de atributos foi movido para uma classe base que Policial e HabitanteNormal tinham em comum.

Figura 10: Modelo de documentação de falhas

Volta-se então a procurar uma nova falha (canto direito superior da Figura 9) e outra iteração se inicia. A análise de um *software* termina quando não forem mais encontradas falhas. Ao término desse processo, tem-se um repositório com várias versões do programa analisado e com uma seqüência de rótulos delimitando os pontos entre uma eliminação de falha e outra, cada rótulo com sua respectiva documentação e com as métricas daquela versão, de maneira que seja possível analisar o comportamento das métricas de uma eliminação de falha para outra.

4.3.1 Resultados

Após a análise de 12 programas conforme o processo descrito anteriormente, foram detectadas 37 ocorrências de falhas distribuídas em 9 falhas diferentes, cuja relação é apresentada a seguir. Entre parênteses encontram-se os nomes originais, quando se tratar de uma falha prevista pela literatura [Fowler, 1999a].

1. **Acoplamento Forte** (*Feature Envy*): ocorre quando um método utiliza mais os atributos de outra classe (ainda que encapsulados) do que os de sua própria.

2. **Algoritmo Ruim**: trecho de código (algoritmo) implementado de maneira

complexa; normalmente caracterizada, entre outras coisas, pela presença de variáveis desnecessárias, de blocos de código aninhados em vários níveis, pelo uso de variáveis de sinalização (*flags*), pelo excesso de estruturas condicionais e de trechos de código duplicado.

3. **Mau Uso de Herança**: apesar de ter havido apenas uma ocorrência, essa é uma falha grave. Foram derivadas de uma classe base duas árvores hierárquicas cujos propósitos não tinham nenhum relacionamento semântico, apenas para que objetos de ambas as árvores pudessem ser adicionados à mesma coleção. A conseqüência imediata dessa falha é obrigar o programador, sempre que acessar um objeto de uma dessas coleções, a verificar qual é o seu tipo e a realizar uma conversão do tipo geral (da classe base) para o tipo específico que desejar utilizar. Essa conversão é sempre necessária uma vez que a classe base é "vazia" e nada, além do seu tipo, é herdado dela. Isso acaba por tornar o programa mais complexo e mais susceptível a erros (de conversão de tipos).

4. **Herança Negligenciada** (caso particular de *Data Clumps*): caracteriza-se pela repetição de grupos de atributos (ou mesmo de atributos únicos) em classes que fazem parte de uma mesma árvore hierárquica. Esse tipo de falha se dá devido à aplicação negligente do recurso de herança. Se os atributos são os mesmos, eles devem então ser concentrados na classe base, de maneira que a duplicação de código seja evitada, visto que métodos semelhantes que manipulam esses atributos repetem-se junto com eles.

5. **Lista de Parâmetros Longa** (*Long Parameter List*): consiste em métodos que recebem vários parâmetros. Listas de parâmetros longas são difíceis de se entender e tendem a mudar constantemente [Fowler, 1999a], além de normalmente prejudicarem o encapsulamento ao forçar a quebra de um objeto em seus vários atributos para que o

método possa ser chamado.

6. **Método Longo** (*Long Method*): é desejável que os métodos não sejam longos, uma vez que esses são mais difíceis de entender e de manter [Fowler, 1999a]. Não há um número exato de linhas de código (LoC – *Lines of Code*) que caracterize um método longo, mas neste estudo, todos os métodos considerados longos tinham mais de 40 LoC.

7. **Parâmetro desnecessário** (caso particular de *Speculative Generality*): é passado para um método um parâmetro que nunca é utilizado. Parâmetros inúteis apenas dificultam a utilização do método.

8. **Uso de técnicas procedimentais ao invés de polimorfismo** (generalizada a partir de *Switch Statements*): essa falha caracteriza-se pelo uso de estruturas de decisão (`if-else` e `switch`) combinadas com variáveis de sinalização (*flags*) e com outras maneiras de se determinar tipos de objetos (operador `instanceof`, por exemplo) para decidir qual operação executar. Selecionar qual operação deve ser executada com base no tipo de um objeto é exatamente o papel do polimorfismo, que permite que esse resultado seja conseguido de uma maneira mais simples e flexível do que por meio do uso de técnicas procedimentais (ver exemplo na sub-seção 2.4.1).

9. **Violação de encapsulamento**: caracteriza-se pela "quebra" do encapsulamento, um dos conceitos fundamentais da orientação a objetos [Horstmann e Cornell, 2004]. Se dá basicamente pelo uso indevido de modificadores de visibilidade (como `public` e `private`).

As ocorrências das falhas detectadas estão distribuídas conforme pode ser verificado no gráfico de freqüências da Figura 11. Pode-se verificar que das falhas identificadas, as três que

ocorreram com a maior freqüência foram, em ordem: “uso de técnicas procedimentais ao invés de polimorfismo”, “violação de encapsulamento” e “método longo”.

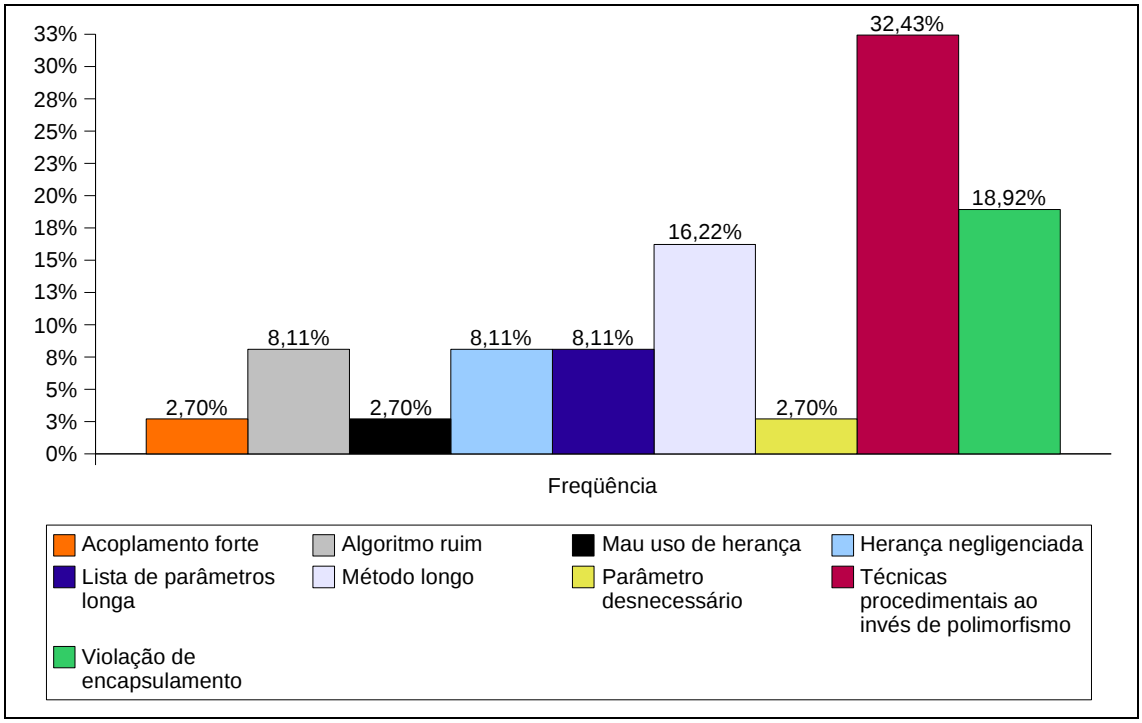

Figura 11: Freqüência de ocorrência das falhas

Antes da realização dessa análise, esperava-se encontrar – além das encontradas – falhas essencialmente ligadas ao paradigma OO [Baranauskas, 1995], como longas cadeias de mensagens enviadas de um objeto para outro (*Message Chains*) ou características herdadas de classes em níveis superiores que nunca são usadas (*Refused Bequest*) [Fowler, 1999a]. Essas são falhas que podem ser categorizadas como falhas de projeto[22] orientado a objetos. Contudo, é possível observar que as falhas identificadas neste estudo dividem-se em duas categorias: as que não são essencialmente ligadas ao paradigma OO e também podem se manifestar no paradigma procedimental (Algoritmo Ruim, Lista de Parâmetros Longa, Método Longo e Parâmetro Desnecessário); e as que indicam o mau uso da OO (Acoplamento Forte, Mau Uso de Herança, Herança Negligenciada, Uso de Técnicas Procedimentais ao Invés de Polimorfismo e Violação de Encapsulamento).

22 Usando a palavra projeto como uma tradução para o termo inglês *design*.

Essa natureza das falhas encontradas evidencia uma tendência que pôde ser percebida durante a análise: apesar dos programas terem sido implementados numa linguagem orientada a objetos (OO), boa parte de sua estrutura é estritamente procedimental e os recursos da OO são usados apenas superficialmente. Uma hipótese é que os programadores tenham adotado uma linguagem OO, mas que não tenham se adaptado ao paradigma de programação e continuam projetando suas implementações conforme o paradigma da programação estruturada.

4.4 Análise das Métricas

Como já mencionado, além de analisar as falhas de manutenibilidade mais freqüentes, também é objetivo deste trabalho estudar comportamento das métricas de produto de *software* após a eliminação dessas falhas. Por esse motivo, fez-se necessário não só identificar a ocorrência das falhas, mas também eliminá-las, para que as métricas pudessem ser coletadas antes e depois de cada eliminação.

As atividades de eliminação de falhas e de coleta de métricas são previstas no processo de análise discutido anteriormente (Figura 9), o qual foi elaborado de maneira que o histórico de cada *software* analisado fosse mantido por meio de um repositório de controle de versões.

Em cada coleta, os valores de um conjunto de métricas eram calculados para cada artefato aos quais as métricas fossem aplicáveis (tais como classes, pacotes e métodos) e armazenados na versão atual do arquivo de métricas. As métricas que compunham esse conjunto foram determinadas pelas capacidades da ferramenta[23] de cálculo utilizada. Das 19 métricas utilizadas neste estudo, quatro (WMC, DIT, LCOM e NSC) fazem parte do conjunto de métricas CK, apresentado na sub-seção 3.3.1. A relação das demais, a maior parte definida segundo a documentação da própria ferramenta, é apresenta a seguir.

23 *Plugin* Metrics (http://metrics.sourceforge.net) para o ambiente Eclipse (http://www.eclipse.org).

1. *Abstractness* (RMA): a soma do número de classes abstratas e interfaces dividida pelo total de tipos em um pacote.

2. *Afferent Coupling* (CA): número de classes fora de um pacote que dependem de classes de dentro do pacote.

3. *Efferent Coupling* (CE): número de classes de dentro de um pacote que dependem de classes de fora do pacote.

4. *Cyclomatic Complexity* (VG): a complexidade ciclomática é uma medida da complexidade de um bloco de código, baseada em teoria dos grafos [McCabe, 1976]. Quanto maior o seu valor, mais complexo é o programa medido.

5. *Method Lines of Code* (MLOC): número de linhas de código de um método.

6. *Nested Block Depth* (NBD): maior profundidade dos blocos de código aninhados dentro de um método. Por exemplo, se dentro de um método houver um bloco `while` contendo um `if`, a NBD será 3, uma vez que o bloco de código que delimita o método também é contado. Essa também é uma medida de complexidade.

7. *Number of Attributes (or Fields)* (NOF): número de atributos de uma classe.

8. *Number of Classes* (NOC): número de classes contidas num determinado escopo (pacote, sub-pacotes ou programa inteiro).

9. *Number of Interfaces* (NOI): análogo à NOC, porém conta o número de interfaces.

10. *Number of Methods* (NOM): número total de métodos dentro de um determinado escopo.

11. *Number of Overriden Methods* (NORM): número de métodos, dentro de um determinado escopo, que foram sobrescritos em classes derivadas de outras.

12. *Number of Parameters* (PAR): número de parâmetros que um método recebe.

13. *Number of Static Attributes* (NSF): número de atributos estáticos de uma classe.

14. *Number of Static Methods* (NSM): número de métodos estáticos de uma classe.

15. *Specialization Index* (SIX): "*o índice de especialização mede o quanto uma sub-classe sobrescreve (substitui) o comportamento de suas classes base*" [Schroeder, 1999]. A ferramenta utilizada define o índice de especialização como a multiplicação de NORM por DIT dividida por NOM. Valores próximos ou maiores que 1 indicam que a maior parte dos métodos da classe sobrescrevem outros métodos, o que indica problemas de projeto, visto que a classe está "rejeitando" os comportamentos herdados.

Após a análise de cada *software*, tem-se no repositório correspondente uma série de rótulos (*tags*) marcando os pontos em que houve eliminação de falhas, conforme ilustrado na Figura 12. O rótulo `tag-0` marca o estado inicial do *software*, antes de qualquer alteração, e os subseqüentes marcam as eliminações de falhas. Para cada rótulo há uma versão do arquivo de métricas coletadas, tal que, se o rótulo `tag-3` marcar a eliminação de uma ocorrência de uma certa falha, basta comparar as versões do arquivo de métricas marcadas pelo rótulo anterior (`tag-2`) e por esse (nessa ordem) para que se possa estudar as variações das métricas devidas à eliminação da falha.

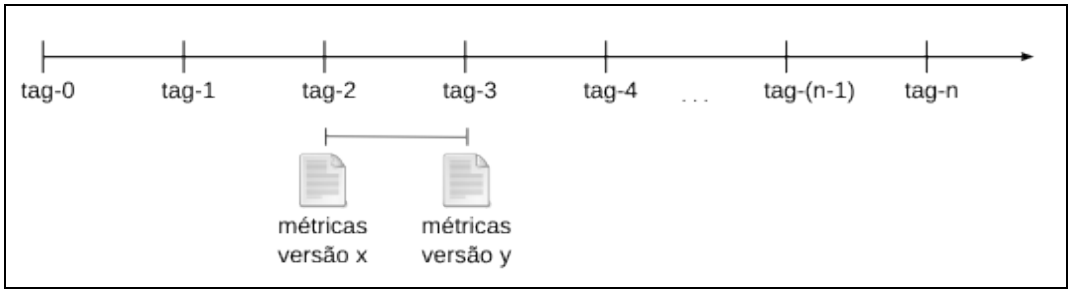

Figura 12: Rótulos em um repositório após análise do *software*

Os rótulos não marcam apenas as versões do arquivo de métricas, mas o estado de todos os

arquivos que compõem o *software* naquele instante. Assim, é possível também acompanhar todas as alterações que o *software* sofreu entre uma seqüência de refatorações (eliminação de falha) e outra. Para cada rótulo, há ainda a documentação a respeito da falha, armazenada como comentário do rótulo (segundo o modelo apresentado na Figura 10). Esses três recursos combinados (repositório marcado com rótulos, métricas coletadas e documentação) viabilizam a análise detalhada da evolução do *software* ao longo das transformações.

Desejava-se, a princípio, acompanhar o comportamento das métricas coletadas artefato por artefato, mas essa possibilidade foi logo considerada inviável visto que as transformações necessárias para a eliminação das falhas envolvem as operações de exclusão, adição e de mudança de nomes de artefatos, o que dificultaria muito – ou talvez até impossibilitaria – esse acompanhamento.

Diante dessa limitação, optou-se por analisar as médias aritméticas dos valores de cada métrica, considerando os valores de todos os artefatos do *software*. Assim, foi obtida a independência das operações citadas sem perder a capacidade de se detectar variações, uma vez que variações individuais afetam a média geral.

A quantificação das variações também não é trivial. As faixas de variação são dependentes do tamanho dos *software*s e por isso não é possível determinar valores absolutos. Por esse motivo, as variações das médias das métricas (entre dois rótulos) foi discretizada em três valores: `aumentou`, `diminuiu` e `constante`. Com isso, tinha-se como objetivo analisar não apenas a variação das métricas, mas também a maneira como variavam. Cogitou-se utilizar algoritmos usados na criação de árvores de decisão[24] [Russel e Norvig, 2002] para analisar a maneira como as métricas variam, porém, os resultados dos experimentos não foram satisfatórios uma vez que essa técnica requer um número de amostras maior do que o

24 Árvores de decisão são uma forma simples e eficaz de algoritmos de aprendizado usada em inteligência artificial. Elas são capazes de “tomar” uma decisão baseadas numa entrada descrita como uma série de atributos [Russel e Norvig, 2002].

disponível para que resultados razoáveis sejam alcançados. Estudou-se então quais foram as métricas afetadas pelas eliminações de falhas, desprezando-se a maneira como elas foram afetadas.

4.4.1 Resultados

Das falhas de manutenibilidade estudadas, analisaram-se as métricas relacionadas às três mais freqüentes (Figura 11), visto que, com o número de *software*s analisados, não foi possível acumular um número significativo de ocorrências para as outras falhas.

Para cada ocorrência de uma dessas três falhas verificou-se quais foram as métricas afetadas pela eliminação da mesma. Foram calculadas as médias das métricas e comparadas com as médias apresentadas no estado marcado pelo rótulo anterior, verificando-se quais métricas sofreram variações. Foram contadas as vezes em que cada métrica foi afetada, obtendo-se as três tabelas apresentadas na Figura 13 (as métricas que não aparecem não foram afetadas). Essas tabelas devem ser interpretadas da seguinte maneira: verifica-se que em 88% dos casos de eliminação de Método Longo, a métrica número de métodos (NOM) foi afetada. Optou-se por desprezar neste estudo as métricas que foram afetadas em menos de 75% dos casos; as linhas sombreadas indicam essas métricas.

Métrica	Freqüência
MLOC	100%
NBD	100%
NOM	100%
PAR	100%
SIX	100%
VG	100%
NOF	92%
NORM	92%
WMC	92%
LCOM	83%
DIT	75%
NOC	75%
NSC	75%
NSM	75%
RMA	75%
CE	33%
NSF	25%
CA	8%

Uso de técnicas Procedimentais ao invés de Polimorfismo

Métrica	Freqüência
LCOM	100%
MLOC	100%
NBD	100%
NOM	100%
PAR	100%
VG	100%
WMC	100%
SIX	29%
NOF	14%
RMA	14%

Violação de Encapsulamento

Métrica	Freqüência
MLOC	100%
NBD	100%
PAR	100%
VG	100%
WMC	100%
NOM	88%
LCOM	75%
NOF	50%
NSF	50%
NSM	50%
SIX	50%
DIT	38%
NORM	38%
NSC	25%
NOC	12%

Método Longo

Figura 13: Tabelas de freqüência das métricas afetadas por cada falha

Observa-se nestas tabelas que a complexidade ciclomática (VG) variou em 100% dos casos para as 3 falhas. Ao se fazer uma análise mais minuciosa, pôde-se verificar que a média da VG sempre diminuiu, ou seja, todas as transformações realizadas fizeram com que a complexidade do *software* diminuísse. Isso é uma evidência clara de que as refatorações realmente diminuem a complexidade dos *software*s, como exposto na sub-seção 2.4.

Embora o número de amostras analisadas para a constituição dessas tabelas seja pequeno do ponto de vista estatístico, pode-se considerá-las como modelos empíricos de probabilidade [Schefler, 1988], ainda que simples. Isso permite que sejam feitas interpretações das tabelas tais como: a probabilidade de que a profundidade da árvore de herança (DIT) seja afetada quando uma falha do tipo *Uso de Técnicas Procedimentais ao invés de Polimorfismo* é eliminada é de 75%.

Uma outra interpretação possível é a de que a probabilidade do número de métodos ponderados pela complexidade (WMC) ser afetado quando se elimina uma falha do tipo

Método Longo é de 100%. Em outras palavras, a WMC é sempre afetada pela eliminação de uma falha Método Longo. Isso pode ser facilmente verificado uma vez que a solução típica para Método Longo é dividir o método em outros menores (refatoração *Extract Method*), o que aumenta o número de métodos, afetando a WMC. Contudo, deve-se ressaltar que, devido ao tamanho das amostras analisadas[25], essas são probabilidades aproximadas e nem todas as probabilidades de 100% podem ser verificadas com a mesma facilidade, havendo inclusive a possibilidade desses valores serem menores.

4.5 Considerações Finais

A sistematização de um processo de análise (Figura 9) foi fundamental para a realização do estudo. Essa abordagem disciplinada propiciou a formação de bases de dados amostrais (os repositórios) organizadas e documentadas.

Por meio da análise desses dados, foram identificadas ocorrências de várias falhas de manutenibilidade. Algumas exatamente como descritas na literatura, outras que foram adaptadas e outras ainda que não foram encontradas na literatura examinada. Essas falhas identificadas na primeira etapa do estudo revelaram que a orientação a objetos tem sido aplicada de maneira superficial. Outras falhas, essencialmente ligadas ao paradigma OO, não foram encontradas.

Na segunda etapa do estudo, verificou-se que algumas métricas são afetadas pela eliminação de falhas com certa probabilidade. Uma constatação importante feita nessa etapa é a de que a complexidade ciclomática (VG) sempre diminui após uma seqüência de refatorações, o que comprova que essas transformações realmente diminuem a complexidade

25 É sugerido na literatura que para a formação de um modelo empírico razoável sejam analisados alguns milhares de amostras [Schefler, 1988]. Considerando as mesmas proporções verificadas neste trabalho, para a formação de uma amostra de 5.000 ocorrências de Método Longo, por exemplo, seriam necessárias aproximadamente 30.000 ocorrências de falhas, ou a análise de 10.000 *software*s, o que é inviável para um projeto de pesquisa executado por apenas uma pessoa dentro do prazo de um trabalho de mestrado.

dos programas.

No capítulo 5, é apresentado um processo de inspeção de *software* apoiado por artefatos criados a partir desses resultados.

Capítulo 5 - Processo de Inspeção

5.1 Considerações Iniciais

É apresentado neste capítulo um processo de inspeção de *software* apoiado por artefatos desenvolvidos a partir de resultados obtidos nas análises apresentadas no capítulo 4, a saber: um cenário PBR[26] para a perspectiva de um mantenedor de *software* orientado a objetos, uma relação de falhas de manutenibilidade freqüentes, uma lista de recomendações para correção dessas falhas e um documento de métricas que são afetadas pelas falhas, para ser usado na validação das correções. O processo apresentado oferece apoio a melhorias por meio da sistematização da documentação de falhas identificadas nas inspeções. Inicialmente, trata-se do processo de inspeção e dos modelos dos produtos de trabalho gerados, em seguida são apresentados os artefatos de apoio ao processo.

5.2 Processo

Não é objetivo deste trabalho propor um novo processo de inspeção, diferente do encontrado na literatura [Shull et al., 2000] [Sommerville, 2001]. De fato, pode ser observado que o processo aqui descrito é composto basicamente pelas mesmas atividades descritas por Fagan [1976] em seu trabalho sobre inspeções, as quais são descritas neste texto na seção 3.2. O foco deste trabalho está nos artefatos de apoio ao processo – já mencionados – e na sua contínua melhoria por meio da sistematização da documentação das falhas e métricas, durante as execuções do processo.

Na Figura 14 é ilustrado o processo de inspeção, as saídas de cada atividade e os pontos em

26 Leitura Baseada em Perspectivas (*Perspective-Based Reading*). Ver sub-seção 3.2.1.

que os artefatos de apoio são utilizados.

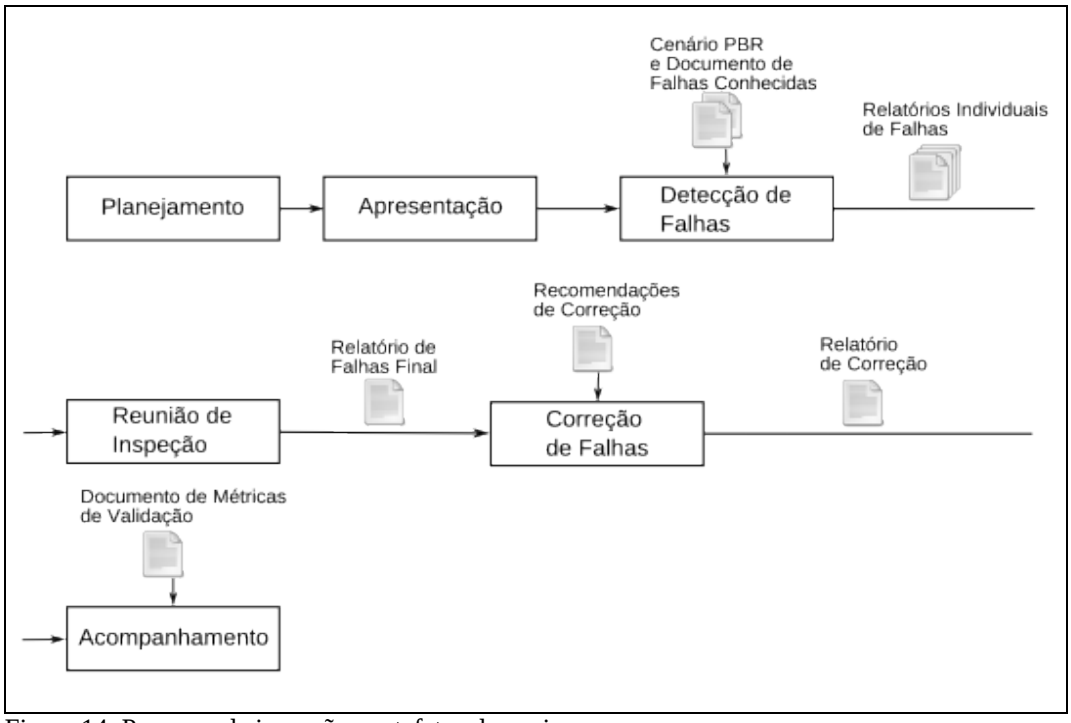

Figura 14: Processo de inspeção e artefatos de apoio

O processo se inicia pela atividade de planejamento, na qual trata-se do agendamento da apresentação e da reunião de inspeção, da reserva de recursos (tais como salas), do envio dos convites aos participantes e de se garantir que os artefatos de *software* – no contexto desse trabalho, basicamente classes descritas por código-fonte – estejam completos. Essas são atribuições do moderador da inspeção. A seguir, é feita uma apresentação coletiva do projeto e dos artefatos sob inspeção aos participantes. Essa apresentação é feita pelo autor dos artefatos e, nessa reunião inicial, é provida aos participantes a documentação disponível sobre o projeto e o *software*, além de cópias dos artefatos.

A atividade seguinte, a de detecção de falhas, é realizada individualmente e recebe como entrada o código-fonte dos artefatos que serão inspecionados e dois documentos de apoio: o documento de falhas conhecidas e o cenário PBR. O inspetor deve estudar o documento de falhas conhecidas para que saiba reconhecer falhas já identificadas e para que possa fazer analogias com outras que serão identificadas pela primeira vez. Os artefatos devem então ser

examinados conforme as instruções contidas no cenário PBR. O cenário esclarece ao inspetor o papel que deve assumir e dá a noção do que é relevante. Ao seguir as instruções, o inspetor é induzido a encontrar falhas. A combinação desses dois artefatos de apoio proporciona a disseminação do conhecimento e permite que inspetores com pouca experiência possam desempenhar o seu trabalho satisfatoriamente. Cada inspetor produz, nessa atividade, um relatório individual de falhas, as quais devem ser documentadas conforme o modelo apresentado na Figura 15, derivado do modelo de documentação usado na análise de falhas (Figura 10), porém sem a seção de prioridades.

Considera-se, neste trabalho, inspeções cujo único propósito seja a detecção de falhas de manutenibilidade. O objetivo principal das reuniões de inspeção é a detecção de falhas e não a sua correção [Fagan, 1976]. Por esse motivo, não há uma seção dedicada à correção das falhas no modelo de documentação. Correções simples, se identificadas facilmente, podem ser anotadas informalmente nos relatórios.

Número de Identificação
Número da falha (numeração seqüencial). Opcional nos relatórios individuais e indispensável no relatório final.

Exemplo: 4

Prioridade
Prioridade de correção da falha. Valores possíveis são: ALTA, MÉDIA e BAIXA. Usa-se apenas no relatório final.

Exemplo: MÉDIA

Descrição
Uma breve descrição da falha ou seu nome, em caso de falha conhecida.

Exemplo: Classe executando operações que não são de sua responsabilidade

Artefatos Envolvidos
Uma relação, feita de modo sucinto, dos artefatos envolvidos na falha.

Exemplo: Classe Extrato, método enviarEmail().

Evidências
Evidências da ocorrência da falha

Exemplo: A classe Extrato, cuja finalidade é manter uma seqüência de transações realizadas em uma conta, possui um método para envio do extrato por e-mail ao cliente. Isso compromete a coesão da classe.

Figura 15: Modelo de documentação de falha

Na reunião de inspeção, são utilizados os relatórios individuais produzidos na atividade anterior e o código-fonte do programa sob inspeção. Participam da reunião, a exemplo do que se encontra na literatura [Fagan, 1976] [Shull et al., 2000] [Sommerville, 2001], um moderador, o autor dos artefatos e pelo menos dois inspetores. As falhas descritas nos relatórios individuais dos inspetores são discutidas uma a uma na reunião e podem ser aceitas ou rejeitadas. Assim como é aconselhado por Fagan [1976], é responsabilidade do moderador permitir que as discussões não se prolonguem além do ponto em que se pode determinar a aceitação ou rejeição de uma falha. As falhas aceitas são priorizadas e compiladas no relatório final da reunião de inspeção, composto por descrições feitas segundo o modelo apresentado na Figura 15.

Na próxima atividade, a de correção de falhas, o autor dos artefatos examinados utiliza o

relatório de falhas final produzido na reunião e o documento de recomendações de correção. Pode-se recorrer ao documento de falhas conhecidas como um auxiliar para a compreensão do relatório de falhas, se isso for necessário. O autor deve proceder com a correção de cada uma das falhas, em ordem de prioridade, seguindo o processo descrito na Figura 16. Deve-se coletar as métricas antes da correção, corrigir a falha por meio das refatorações apropriadas, testar o programa para garantir que o comportamento externo tenha sido preservado, coletar as métricas após a correção e finalmente descrever, no relatório de correção, a maneira como a falha foi corrigida.

Nessa atividade, o documento de recomendações de correção deve ser utilizado como um guia para a correção de falhas conhecidas. O relatório de correção deve conter uma descrição sucinta da correção de cada uma das falhas, as quais devem ser identificadas pelo seu número. Quando se tratar de uma falha conhecida cuja correção tenha partido do documento de recomendações, deve-se apenas indicar qual das correções foi utilizada. A correção, ainda que para uma falha conhecida, deve ser descrita caso não faça parte do documento de recomendações. As métricas coletadas devem ser armazenadas em arquivos que, assim como o relatório de falhas, devem ser disponibilizados ao moderador.

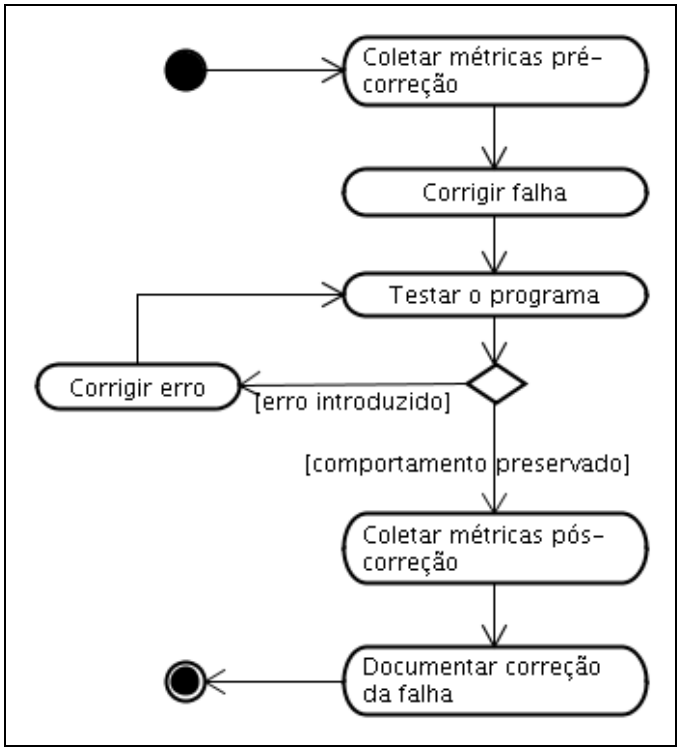

Figura 16: Processo de correção de falhas

Apesar dessa discussão estar fora do escopo deste trabalho, é altamente aconselhável o uso de ferramentas e procedimentos de gerência de configuração para a realização do processo de correção de falhas, a exemplo do que é feito no estudo descrito no capítulo 4. A organização do trabalho por meio desses procedimentos e ferramentas pode evitar transtornos e proporcionar melhorias na documentação das atividades com menos esforços, uma vez que, por meio das versões armazenadas, passa-se a depender menos de descrições detalhadas.

Finalmente, na atividade de acompanhamento (Figura 14), o moderador utiliza os relatórios de correção e de falhas para verificar se as falhas foram satisfatoriamente corrigidas. O moderador deve verificar se há falhas cuja correção é complexa e demanda tempo para avaliação e, sendo essas falhas previstas no documento de métricas de validação, pode fazer uma comparação entre os arquivos de métricas pré e pós correção – recebidos do autor do *software* – e analisar se as métricas esperadas foram afetadas. Se sim, tem-se então um indício de que a falha foi realmente corrigida, bastando para a confirmação uma análise cognitiva final, porém demandando menos esforços. Para falhas simples, pode-se dispensar o

apoio das métricas, visto que as análises cognitivas requerem poucos esforços nesses casos. As correções das falhas que não sejam previstas no documento de métricas de validação, devem ser validadas cognitivamente. Se houver falhas do relatório de inspeção que não foram corrigidas, o moderador deve reencaminhá-las ao autor dos artefatos para correção. Cabe ao moderador avaliar se as falhas foram corrigidas satisfatoriamente e também reencaminhá-las para ajustes, podendo inclusive agendar uma nova inspeção para que sejam analisadas, se julgar necessário.

A documentação das falhas encontradas, das correções e as métricas coletadas podem ser utilizadas para a melhoria dos artefatos de apoio, por meio de análises semelhantes à que foi feita no capítulo 4. A execução de tais análises tem o papel de adaptar os artefatos de apoio ao contexto em que as inspeções são conduzidas e, portanto, tem fundamental importância. Novas falhas podem ser reveladas, assim como a maneira de corrigi-las pode ser documentada poupando esforços futuros, padronizando as soluções adotadas e promovendo soluções maduras. Os resultados dessas análises também podem refletir em melhorias no cenário PBR, incluindo instruções específicas para revelar falhas sabidamente críticas, por exemplo. O mesmo vale para o documento de métricas de validação que pode ser continuamente refinado, fazendo com que as conclusões tiradas das análises do comportamento das métricas sejam cada vez mais precisas.

É importante salientar que as tabelas e relatórios textuais não são considerados como único meio de se documentar o trabalho realizado nas atividades intermediárias do processo de inspeção. Acredita-se, inclusive, que essa documentação possa ser feita de maneiras mais eficientes, desde que mantidos os procedimentos e o conteúdo sugeridos. Uma possível maneira de implementar os mesmos procedimentos de documentação seria substituir os relatórios de falhas (individuais e final) e de correção por uma ferramenta de acompanhamento de defeitos, havendo inclusive alternativas em *software* livre bem

conhecidas[27] e que podem ser até adaptadas para melhor atender a essas necessidades. O uso ou não de ferramentas de apoio e outras variações na implementação do processo ficam condicionados às necessidades presentes no ambiente de aplicação das inspeções.

5.3 Artefatos de Apoio

São apresentados nesta seção os artefatos de apoio citados anteriormente, gerados a partir dos estudos descritos no capítulo 4.

5.3.1 Cenário PBR

O cenário determina a perspectiva sob a qual os artefatos devem ser analisados (neste trabalho, apenas a do mantenedor) e esclarece quais são os objetivos que o inspetor deve ter ao fazer a leitura dos artefatos em busca de falhas. São oferecidas instruções e um questionário cujo objetivo é evidenciar falhas. O processo de elaboração do cenário, apresentado nesta sub-seção (Figura 18), foi inspirado por um processo descrito na literatura [Laitenberger e DeBaud, 1998], na abordagem GQM (*Goal Question Metric*) [Basili et al., 1994] e na análise de exemplos de cenários encontrados na literatura [Shull et al., 2000]. Esse processo é descrito na Figura 17.

1. Identificar as falhas que podem ocorrer (conhecidas)
2. Identificar as causas (evidências) das falhas
3. Elaborar instruções que levem o inspetor a encontrar as evidências das falhas
4. Elaborar perguntas sobre as evidências das falhas de maneira direcionar a atenção do inspetor enquanto segue as instruções

Figura 17: Processo de elaboração do cenário PBR

Por meio desse processo, foi produzido o cenário apresentado na Figura 18.

27 Exemplos: Bugzilla (http://www.bugzilla.org) e Trac (http://trac.edgewall.org/).

Assuma que você está inspecionando os artefatos sob a perspectiva do mantenedor do *software*. Ao mantenedor interessa garantir o melhor nível de manutenibilidade possível ao *software*. Isso implica em implementações o mais simples possível com o máximo de compreensibilidade e o mínimo de acoplamento entre classes. Implementações desnecessariamente extensas e complexas são indesejáveis.

Siga as instruções a seguir e responda às perguntas (no final) enquanto as segue.

Examine todo o código-fonte (dentro do escopo no qual se encerra a inspeção), inicialmente sem se ater aos detalhes de implementação. Procure identificar as responsabilidades de cada classe. Para tanto, examine seus atributos e operações (métodos) procurando compreender quais são os seus propósitos, por meio de seus nomes e parâmetros. Identifique as associações entre classes e as relações de herança. O uso de um IDE (*Integrated Development Enviroment*) com recursos de navegação em código-fonte pode facilitar bastante essa etapa. Procure relacionar o que está examinando com a documentação do *software* (se houver documentação).

Numa segunda etapa faça uma análise mais minuciosa do código-fonte, prestando atenção aos detalhes de implementação. Identifique um ponto onde uma ação é iniciada, isto é, o ponto de entrada da ação (método main() ou o tratador de um evento). Acompanhe, navegando entre as classes, as mensagens enviadas de uma classe para outra (chamadas de métodos) de maneira a compreender como as classes se comunicam para completar a tarefa. Entenda a origem e as finalidades dos parâmetros passados para as operações. Repita esse processo para todas as ações executadas pelo sistema (dentro do escopo no qual se encerra a inspeção).

Finalmente, já tendo um bom conhecimento do programa, procure conhecer cada classe com seus atributos e métodos. Procure entender o processo de criação dos objetos, isto é, como e de onde seus atributos recebem seus valores iniciais. Analise cada método procurando entender a sua implementação.

Perguntas

1. Algum método depende mais de atributos de outra classe do que dos da classe em que está contido?
2. Há variáveis ou parâmetros não utilizados?
3. Existem blocos de código aninhados em vários níveis?
4. Foi possível compreender a implementação de todos os métodos? Ou houve implementações cuja falta de clareza dificultou a compreensão?
5. Algum método faz uso intenso de estruturas condicionais?
6. São usadas variáveis de sinalização (*flags*)?
7. Há trechos de código muito similares ocorrendo em mais de um lugar?
8. Algum recurso da linguagem é usado de maneira inusitada? Por exemplo, derivar duas classes de uma classe base comum apenas para poder adicioná-las ao mesmo vetor.
9. Há classes diferentes com grupos de atributos iguais que se repetem?
10. As listas de parâmetros dos métodos são longas? Mais de 4 ou 5 parâmetros, por exemplo.
11. Há métodos cuja implementação consiste em muitas linhas? Mais de 40 linhas, por exemplo.
12. Ocorre o uso combinado de estruturas de decisão com variáveis de sinalização (*flags*) ou tipos enumerados?
13. O operador `instanceof` é usado em alguma situação em que o uso poderia ter sido evitado?

14. São feitas comparações envolvendo o retorno do método `getClass()` da classe `Object`?
15. Algum bloco de código é executado como resultado de uma comparação envolvendo o retorno do método `toString()` da classe `Object` ou de alguma outra comparação envolvendo o estado (valor de atributo ou retorno de método) de um objeto?
16. Há atributos de classes sendo acessados diretamente a partir de outras?
17. Os modificadores de acesso (`public`, `protected` e `private`) são usados adequadamente? Por exemplo, vários métodos (ou atributos) são indiscriminadamente marcados por `public` ou `protected` sem nenhuma razão aparente.
18. Métodos auxiliares, que só são relevantes para a classe que os declara, são marcados com `private`?
19. Os nomes das classes, métodos e variáveis refletem bem o seu propósito?

Figura 18: Cenário PBR

5.3.2 Relação de Falhas Conhecidas

O documento de falhas conhecidas, apresentado nesta sub-seção, muito se assemelha à relação de falhas apresentada como um dos resultados do estudo descrito no capítulo 4, uma vez que essa é a sua origem. Porém, a relação aqui apresentada foi complementada com exemplos ilustrativos.

Acoplamento Forte (por atributos)

Ocorre quando um método utiliza mais os atributos de outra classe (ainda que encapsulados) do que os de sua própria.

Exemplo: numa classe `Extrato`, o método `calcularRendimentos()` utiliza os atributos saldo e taxa (`getSaldo()` e `getTaxa()`) da classe `Poupanca` para calcular os rendimentos num determinado período. O método `calcularRendimentos()` deveria pertencer à classe `Poupanca`, e não à `Extrato`.

Algoritmo Ruim

Trecho de código (algoritmo) implementado de maneira complexa; normalmente caracterizada, entre outras coisas, pela presença de variáveis desnecessárias, blocos de código aninhados em vários níveis, uso de variáveis de sinalização (*flags*), pelo excesso de estruturas condicionais e trechos de código duplicado.

Exemplo: deseja-se calcular o total de tempo de utilização de um serviço durante um mês e a média de utilização diária, considerando como entradas valores de tempo de utilização diário no formato hora:minuto:segundo e expressar o resultado no mesmo formato. Uma má implementação consistiria em tratar cada campo (hora, minuto e segundo) separadamente, usando `if`'s para fazer com que valores maiores que 60 segundos (por exemplo) fossem tradados como minutos, mantendo a parcela menor que 60 no campo dos segundos; e o mesmo com relação a minutos e horas. Essa implementação é complexa e susceptível a erros. Uma solução mais simples e eficaz consiste em converter todos os valores para a menor unidade (segundos) antes dos cálculos, executar os cálculos na mesma unidade e então converter novamente o resultado para o formato desejado.

Mau Uso de Herança

Uso de herança para contornar problemas advindos da má modelagem de classes.

Exemplo: Foram derivadas de uma classe base duas árvores hierárquicas cujos propósitos não tinham nenhum relacionamento semântico, apenas para que objetos de ambas as árvores pudessem ser adicionados à mesma coleção. A conseqüência imediata dessa falha é obrigar o programador, sempre que acessar um objeto de uma dessas coleções, a verificar qual é o seu tipo e a realizar uma conversão do tipo geral (da classe base) para o tipo específico que desejar utilizar. Essa conversão é sempre necessária uma vez que a classe base é “vazia” e nada, além do seu tipo, é herdado dela. Isso acaba por tornar o programa mais complexo e mais susceptível a erros (de conversão de tipos).

Herança Negligenciada

Caracteriza-se pela repetição de grupos de atributos (ou mesmo de atributos únicos) em classes que fazem parte de uma mesma árvore hierárquica. Esse tipo de falha se dá devido à aplicação negligente do recurso de herança. Se os atributos são os mesmos, eles devem então ser concentrados na classe base, de maneira que a duplicação de código seja evitada, visto que-, métodos semelhantes que manipulam esses atributos repetem-se junto com eles.

Exemplo: repetir o atributo `saldo` nas classes `ContaCorrente` e `ContaPoupanca` ao invés de fazer com que ambas sejam derivadas de uma classe comum chamada `Conta` que tenha o atributo `saldo`.

Lista de Parâmetros Longa

Consiste em métodos que recebem vários parâmetros. Listas de parâmetros longas são difíceis

de se entender e tendem a mudar constantemente, além de normalmente prejudicarem o encapsulamento ao forçar a quebra de um objeto em seus vários atributos para que o método possa ser chamado.

Exemplo: o método `buscar()` da classe `ClienteDAO` recebe como parâmetros o nome do cliente, sobrenome, cidade, país e ano de nascimento, ou seja, 5 parâmetros. O método poderia receber apenas um parâmetro, um objeto da classe `Cliente` encapsulando todos esses atributos. Assim, mudanças nos critérios de busca não afetariam a assinatura do método. É importante observar que uma lista de parâmetros ser longa ou não depende do contexto.

Método Longo

É desejável que os métodos não sejam longos, uma vez que esses são mais difíceis de entender e de manter. Não há um número exato de linhas de código (LoC – *Lines of Code*) que caracterize um método longo, mas, com certa freqüência, métodos considerados longos têm mais de 40 LoC.

Exemplo: num simples aplicativo de agenda, um método (longo) concentra várias operações em forma de blocos de código orquestrados por estruturas condicionais, assumindo assim várias responsabilidades. Esse método manipula classes de dados, as quais não definem operações.

Parâmetro desnecessário

É passado para um método um parâmetro que nunca é utilizado. Parâmetros inúteis apenas dificultam a utilização do método.

Exemplo: qualquer caso em que um parâmetro recebido não seja utilizado na implementação de um método. Há casos em que os parâmetros são utilizados, mas que poderiam ser derivados de outros já recebidos; nesses casos o parâmetro que pode ser derivado também deve ser removido.

Uso de técnicas procedimentais ao invés de polimorfismo

Essa falha caracteriza-se pelo uso de estruturas de decisão (`if-else` e `switch`) combinadas com variáveis de sinalização (*flags*) e com outras maneiras de se determinar tipos de objetos (operador `instanceof`, por exemplo) para decidir qual operação executar. Selecionar qual operação deve ser executada com base no tipo de um objeto é exatamente o papel do polimorfismo, que permite que esse resultado seja conseguido de uma maneira mais simples e flexível do que por meio do uso de técnicas procedimentais.

Exemplo: no método `calcularRendimentos` da classe `Conta`, é verificado se o atributo tipo é igual a `CORRENTE` ou `POUPANCA` para que os rendimentos sejam calculados conforme as regras correspondentes (no caso de conta corrente, que não tem rendimentos, são apenas descontadas as taxas). Ao invés das variáveis de tipo deveriam existir duas classes: `ContaCorrente` e `ContaPoupanca`, com a segunda sendo derivada da primeira e o método `calcularRendimentos` deveria ser sobrescrito em `ContaPoupanca`, de maneira que o mecanismo de polimorfismo determinasse qual implementação usar baseado na classe à qual pertence o objeto.

Violação de encapsulamento

Caracteriza-se pela “quebra” do encapsulamento, um dos conceitos fundamentais da orientação a objetos. Se dá basicamente pelo uso indevido de modificadores de visibilidade (como `public` e `protected`).

Exemplo: o atributo `nome` da classe `Cliente` é público e acessado diretamente por outras classes. Outra situação possível é a existência de um método auxiliar que só seja relevante para a

classe em que é definido e que foi indiscriminadamente marcado como público, sendo acessível a partir de outras classes.

Figura 19: Relação de falhas conhecidas

5.3.3 Recomendações de Correção

Nesta sub-seção, é apresentado o documento de recomendações de correção para as falhas conhecidas. Sempre que possível, são indicadas para a correção da falha refatorações catalogadas na literatura. Essas recomendações são baseadas no estudo descrito no capítulo 4 e também, no caso de falhas descritas na literatura, nas recomendações contidas no catálogo de refatorações utilizado no estudo [Fowler, 1999a].

Acoplamento Forte (por atributos)

O método ou trecho de código que utiliza os atributos de outra classe deve ser movido para a classe à qual pertencem os atributos. No caso de trechos de código, esses podem ser extraídos para métodos próprios antes de serem movidos.

Refatorações: *Extract Method*, *Move Method*, *Encapsulate Field*.

Algoritmo Ruim

Aconselha-se analisar com atenção o problema que deve ser resolvido e desenvolver uma solução simples que o resolva de maneira correta. Uma possibilidade é pensar em pelo menos 3 soluções possíveis e então escolher uma delas.

Mau Uso de Herança

A solução depende da natureza do mau uso. Para o caso conhecido em que duas árvores hierárquicas são derivadas da mesma classe base apenas para que possa ser utilizada a mesma coleção, aconselha-se utilizar coleções separadas e remover a classe base comum, tornando as duas árvores independentes.

Herança Negligenciada

Os atributos comuns devem ser movidos para uma classe num nível de herança superior. Se não houver relação de herança entre as classes que apresentam os atributos semelhantes, deve-se então criar uma classe base para ambas e então mover os atributos comuns. Os atributos podem ter o mesmo propósito e nomes diferentes; nesse caso deve-se fazer com que os atributos tenham os mesmos nomes (renomeando-os) antes de serem movidos.

Refatorações: *Pull Up Field*, *Pull Up Method*, *Extract Superclass*

Lista de Parâmetros Longa

Freqüentemente é possível obter os dados passados como parâmetro de outras maneiras, consultando objetos que os tem, por exemplo. Deve-se procurar por maneiras alternativas de obter os dados passados como parâmetro sem, contudo, comprometer o encapsulamento. Pode ocorrer que os parâmetros sejam todos obtidos de um mesmo objeto, nesse caso convém substituir a lista de parâmetros pelo próprio objeto. Também pode ser conveniente criar esse objeto, caso não exista.

Refatorações: *Introduce Parameter Object*, *Preserve Whole Object*, *Remove Parameter*

Método Longo

Métodos longos são, com certa freqüência, mal implementados. Nesses casos, podem ser aplicadas as mesmas recomendações que em Algoritmo Ruim. Pode ocorrer que o método não seja todo mal implementado, mas que partes dele sejam. Essas partes podem ser reimplementadas, podendo inclusive ser conveniente extrai-las para métodos próprios para depois reimplementá-las.

Também é comum que métodos longos acumulem muitas responsabilidades. Convém nesses casos isolar cada responsabilidade (trechos de código) em métodos próprios e talvez até mover alguns desses métodos para outras classes onde façam mais sentido, balanceando assim a distribuição de responsabilidades entre as classes.

Outra razão para métodos longos são os trechos de código idênticos ou similares que se

repetem. Esses trechos devem ser extraídos para métodos próprios, podendo ser necessária a inclusão de parâmetros. Os pontos onde os trechos duplicados ocorrem devem ser substituídos por chamadas os novos métodos.

Refatorações: *Extract Method*, *Move Method*, *Introduce Parameter*, *Decompose Conditional*, *Replace Method with Method Object*

Parâmetro desnecessário

Parâmetros desnecessários devem ser removidos.

Refatoração: *Remove Parameter*

Uso de técnicas procedimentais ao invés de polimorfismo

A solução básica para esse problema é transformar um projeto de sistema procedimental (que pode se manifestar num ponto isolado, como uma classe) em projeto orientado a objetos. Isso quase sempre envolve a distribuição de métodos em classes numa árvore hierárquica de maneira que as estruturas condicionais possam ser substituídas pelo uso do polimorfismo. Esses métodos podem não existir na forma de métodos, mas na forma de trechos de código, sendo nesses casos necessário isolá-los em métodos. Freqüentemente é necessário criar a árvore hierárquica por ainda não existir ou reorganizá-la. A aplicação dos padrões de projeto *Strategy* e *State* também pode ser útil.

Refatorações: *Replace Conditional with Polymorphism*, *Replace Type Code with Subclasses*, *Replace Type Code with State/Strategy*, *Introduce Null Object*, *Extract Method*, *Extract Superclass*, *Move Method*

Violação de encapsulamento

Deve-se analisar os níveis de acesso de métodos e atributos e alterá-los de maneira a favorecer o encapsulamento. Uma boa abordagem é manter o menor nível de acesso possível para tudo e aumentá-lo apenas quando necessário. Atributos acessíveis devem ser encapsulados por meio de métodos `get/set`.

Pode ocorrer que um atributo encapsulado da maneira usual por `get/set` seja uma coleção. É possível obter uma referência para a coleção por meio de seu método `get` e alterá-la de fora do objeto, ou seja, é possível alterar o estado de um objeto sem que "ele saiba". Para evitar esse problema aconselha-se retornar uma visão da coleção (uma coleção embrulhada dentro de outra, padrão de projeto *Decorator*) só de leitura por meio do método `get` e fazer com que a adição e remoção de elementos seja feita por meio de métodos `add` e `remove` da classe que contém a coleção.

Refatorações: *Encapsulate Field*, *Encapsulate Collection*

Figura 20: Documento de recomendações de correção para falhas conhecidas

5.3.4 Documento de Métricas de Validação

No documento de métricas de validação são apresentadas as métricas sujeitas a variar quando uma determinada falha é eliminada juntamente com as respectivas probabilidades aproximadas de variação.

Métrica	Probabilidade aproximada
MLOC	100%
NBD	100%
NOM	100%
PAR	100%
SIX	100%
VG	100%
NOF	92%
NORM	92%
WMC	92%
LCOM	83%
DIT	75%
NOC	75%
NSC	75%
NSM	75%
RMA	75%
CE	33%
NSF	25%
CA	8%

Uso de técnicas Procedimentais ao invés de Polimorfismo

Métrica	Probabilidade aproximada
LCOM	100%
MLOC	100%
NBD	100%
NOM	100%
PAR	100%
VG	100%
WMC	100%
SIX	29%
NOF	14%
RMA	14%

Violação de Encapsulamento

Métrica	Probabilidade aproximada
MLOC	100%
NBD	100%
PAR	100%
VG	100%
WMC	100%
NOM	88%
LCOM	75%
NOF	50%
NSF	50%
NSM	50%
SIX	50%
DIT	38%
NORM	38%
NSC	25%
NOC	12%

Método Longo

Figura 21: Documento de métricas de validação

5.4 Considerações Finais

A descrição do processo de inspeção, indicando as entradas e saídas de cada atividade e os pontos em que os artefatos de apoio devem ser utilizados, esclarece aos usuários desse processo o que é um processo de inspeção e o papel de cada artefato de apoio. Isso é particularmente interessante quando os inspetores possuem pouca experiência.

O cenário PBR e a relação de falhas conhecidas, assim como em outros processos de inspeção, tornam a definição e a identificação de falhas independentes do julgamento pessoal dos inspetores. As recomendações de correção, além de auxiliar os autores na fase de correção, contribuem para a adoção de soluções maduras e para a padronização das estruturas de *software*.

O modelo de relatório de inspeção apresentado é adaptado ao contexto da manutenibilidade

e ao paradigma da orientação a objetos, prevendo o apontamento de mais de um artefato envolvido numa falha, o que é uma situação comum, ao invés do apontamento de linhas de código identificadas por seu número, como em outros modelos. Esse modelo também propicia ao autor entender o porquê de determinada implementação ou projeto ter sido considerado falho, favorecendo assim o seu aprendizado.

Os modelos de relatórios de inspeção e de correção associados favorecem a melhoria contínua do processo, sendo possível associar as falhas e as correções adotadas, de maneira que o documento de recomendações de correção pode ser melhorado. O acúmulo das métricas coletadas durante as correções possibilita também a melhoria do modelo de probabilidade de variação das métricas afetadas por correções.

É descrito no capítulo 6, a seguir, um estudo de caso de aplicação desse processo e dos artefatos de apoio.

Capítulo 6 - Estudo de Caso

6.1 Considerações Iniciais

Relata-se, neste capítulo, um caso de aplicação do processo de inspeção descrito no capítulo 5. Inicia-se pela contextualização do estudo de caso, apresentando o *software* inspecionado, os participantes do processo e o ambiente no qual foi realizada a inspeção. São discutidos os resultados da inspeção e, finalmente, faz-se uma análise da aplicabilidade do processo e dos artefatos de apoio baseada nas opiniões dos participantes do estudo de caso. Verificar a aplicabilidade do processo e dos artefatos de apoio é o principal objetivo do estudo de caso.

6.2 Aplicação do Processo

O processo de inspeção descrito no capítulo 5 foi aplicado em uma pequena empresa do ramo de desenvolvimento de *software*. Foi inspecionado um sistema *web*[28] de apoio a uma clínica de fisioterapia, que, além de funcionar como um gerenciador de conteúdo, usado para divulgar a programação de cursos e artigos sobre fisioterapia pela *Internet*, também gerencia o agendamento de consultas. Trata-se de um *software* de pequeno porte – escrito na linguagem Java – composto por 16 classes e aproximadamente 1.100 linhas de código distribuídas em 143 métodos.

A inspeção foi conduzida por duas pessoas, sendo uma delas o autor do *software* – e, portanto, desempenhando esse papel no processo de inspeção – e a outra um desenvolvedor da

28 Sistemas *web* são um misto de conteúdo multimídia e de funcionalidades que fazem uso intensivo de uma rede de computadores [Pressman, 2005]. Os sistemas *web* são normalmente acessados por meio de um navegador que recebe documentos descritos na linguagem HTML (*HyperText Markup Language*), os quais são gerados dinamicamente por uma aplicação remota como resposta às requisições feitas pelo usuário.

empresa que conhecia o *software* apenas superficialmente – e este desempenhou o papel de inspetor. Foi seguido o processo ilustrado na Figura 14, dispensando-se, contudo, as atividades de *planejamento* e *apresentação*, devido ao contexto em que foi realizada a inspeção.

Assim, o inspetor iniciou o processo pela atividade de *detecção de falhas*, usando os artefatos de apoio correspondentes (cenário PBR e relação de falhas conhecidas). Como resultado dessa atividade, foi produzido um relatório de falhas, já no formato do relatório final (conforme o modelo apresentado na Figura 15), visto que não haveria outros relatórios para, na reunião de inspeção, darem origem ao relatório final. Foram detectadas 3 falhas, sendo duas delas do tipo *Método Longo* e a terceira uma inconformidade com os padrões de nomenclatura de artefatos de *software*.

O inspetor realizou uma breve reunião de inspeção com o autor do *software* na qual foi apresentado o relatório de falhas. Iniciou-se, então, a atividade de correção, na qual o autor utilizou a relação de falhas conhecidas e o documento de recomendações de correção como artefatos de apoio. Uma vez que os participantes do estudo de caso não dispunham do programa necessário para coleta de métricas, foi acordado que após a correção de cada falha seria feita uma cópia do *software* para posterior análise do comportamento das métricas pelo autor deste trabalho, de maneira que foi conseguido um efeito semelhante ao obtido pelo uso de rótulos em um repositório (conforme ilustrado na Figura 12).

De posse do relatório de correção produzido pelo autor do *software*, o inspetor – fazendo o papel de *moderador* – realizou a validação das correções (atividade de *acompanhamento*), constatando que as falhas haviam sido corrigidas a contento e finalizando assim o processo de inspeção.

Após a finalização do processo de inspeção, o autor deste trabalho realizou a análise das

métricas a partir dos relatórios e das cópias do *software* mencionadas anteriormente. Foi constatado que, como esperado, a correção da inconformidade com os padrões de nomenclatura não causou variação alguma nos valores das métricas. Isso ocorre por que a correção dessa falha consiste apenas em renomear artefatos, o que não afeta nenhuma das métricas. Já as duas ocorrências de Método Longo afetaram as métricas apresentadas na Tabela 1[29].

Método Longo 1	**Método Longo 2**
MLOC	MLOC
NBD	NBD
NOM	–
PAR	–
VG	VG
WMC	WMC

Tabela 1: Métricas afetadas pelas correções de Método Longo realizadas no estudo de caso

Verifica-se que as métricas afetadas realmente foram as esperadas, de acordo com o documento de *Métricas de Validação* (Figura 21), o que indica a viabilidade do uso de métricas na atividade *acompanhamento* como indicadores da realização das correções. Verifica-se também que a métrica PAR (número de parâmetros de um método) não foi afetada na segunda ocorrência de *Método Longo* (coluna "Método Longo 2" da Tabela 1), apesar de ter uma probabilidade aproximada de variação de 100% (ver Figura 21). Este é um fato também esperado visto que as probabilidades apresentadas no documento de *Métricas de Validação* são aproximações, conforme discutido na sub-seção 4.4.1.

6.3 Aceitação dos Artefatos e Modelos Propostos

Após a execução do processo de inspeção foi requisitado aos dois participantes que

29 O significado das siglas empregadas pode ser verificado na sub-seção 3.3.1 e na seção 4.4.

respondessem a um pequeno questionário. O objetivo foi verificar a aplicabilidade do processo e a aceitação dos artefatos de apoio e dos modelos de documentação por parte dos usuários. A elaboração das perguntas do questionário teve como principal objetivo obter retorno dos usuários a respeito dos seguintes itens:

- eficácia do cenário PBR;
- utilidade do documento de falhas conhecidas;
- eficácia dos modelos de documentação; e
- utilidade das recomendações de correção.

As perguntas relacionadas a um determinado artefato foram direcionadas apenas ao seu usuário. No caso dos artefatos usados por ambos os participantes da inspeção, as respectivas perguntas foram direcionadas aos dois. As respostas foram compiladas e são apresentadas neste texto.

Sobre o cenário PBR, afirmou-se que o mesmo facilitou a identificação das falhas ao tornar claro para o inspetor quais tipos de falhas deveria procurar. O inspetor também afirmou acreditar que o cenário possa ser útil para nivelar o trabalho de vários inspetores com diferentes experiências e níveis de conhecimento. Foi afirmado ainda que as instruções contidas no cenário contribuíram para que cada artefato não precisasse ser relido diversas vezes. E, finalmente, o inspetor acredita que, ao ler o código-fonte da perspectiva de quem deveria realizar a manutenção do mesmo, conforme as instruções do cenário, foi possível identificar falhas de manutenibilidade que teriam passado desapercebidas.

Ambos os participantes da inspeção afirmaram que a relação de falhas conhecidas foi útil nas suas atividades por facilitar a identificação das falhas, principalmente pelos exemplos, os quais foram classificados como particularmente úteis para programadores com pouca

experiência. O inspetor do *software* – um desenvolvedor experiente – afirmou, também, que a relação chamou a sua atenção para questões que antes não considerava como falhas.

Os participantes do processo ressaltaram a importância da objetividade dos modelos de documentação. Foi apontado que há outros modelos de relatórios de defeitos nos quais são requeridas muitas informações que acabam sendo pouco úteis na atividade de correção. Foi observado ainda que as informações contidas nos relatórios de falhas permitem que essas sejam encontradas com facilidade. Por último, o documento de recomendações de correção foi classificado como objetivo e útil para a correção das falhas.

De modo geral, os dois participantes do estudo de caso consideraram úteis os artefatos de apoio propostos, pela sua objetividade e por proporcionarem o nivelamento entre os participantes de uma inspeção. Para concluir, o desenvolvedor que desempenhou o papel de inspetor considerou que seria interessante que um cliente aplicasse um processo de inspeção como esse para garantir a qualidade dos produtos de trabalho entregues por um fornecedor contratado. Esta é uma afirmação coerente, visto que as inspeções são uma maneira de se implementar o controle de qualidade[30].

6.4 Considerações Finais

A princípio, três falhas de manutenibilidade pode parecer um número pequeno. Contudo, tratando-se de um *software* do porte do que foi inspecionado, pode-se considerar esse um número razoável com base na média de falhas encontradas por *software* analisado durante o estudo apresentado no capítulo 4.

Outro fato que chama a atenção é não ter havido falhas do tipo *Uso de Técnicas Procedimentais ao invés de Polimorfismo* (UTPP, para simplificar), apesar de essa ser a falha mais freqüente conforme pode-se observar no gráfico da Figura 11. Isso pode se dever às

30 Ver sub-seção 2.2.3.

características do *software*, ou seja, a não haver situações em que uma ou outra operação é executada em decorrência do tipo de um objeto ou de uma variável de estado (*flag*). Além disso, o reconhecimento de uma UTPP pode não ser trivial, uma vez que, para reconhecê-la, pode ser necessário reconsiderar uma solução que parece indiscutível (uso de `if`'s ou `switch`'es, por exemplo) de um ponto de vista “mais orientado a objetos”. Apesar de tudo, uma das três falhas mais freqüentes – a *Método Longo* – foi encontrada durante o estudo de caso.

Observou-se que as métricas afetadas pelas correções foram justamente as esperadas. Isso é um forte indício da viabilidade do uso de métricas de produto de *software* como indicadores da realização de correções de falhas de manutenibilidade.

De maneira geral, os artefatos de apoio e os modelos de documentação propostos foram bem aceitos pelos participantes do estudo de caso. Este fato aliado à obtenção de resultados positivos por meio da inspeção realizada permite afirmar que a aplicabilidade desses modelos e artefatos foi comprovada. Pôde-se observar ainda, que a proposta desse trabalho foi bem recebida nesse primeiro teste de aceitação (o estudo de caso descrito neste capítulo).

Capítulo 7 - Conclusões e Trabalhos Futuros

7.1 Conclusões

A realização deste trabalho foi motivada pela visível falta de qualidade dos *software*s produzidos atualmente. Mesmo os grandes fornecedores – os maiores do mundo – produzem *software*s falhos. Tais falhas nem sempre se manifestam na forma da geração de resultados errados, mas também em outras formas como limitações e incompletudes.

Acredita-se que uma das raízes das falhas em *software*s seja o nível de complexidade de suas estruturas internas. Essa complexidade os torna menos flexíveis e mais susceptíveis a diversos tipos de erro (falhas de segurança e produção de resultados errados, por exemplo). Assim, ao controlar essa complexidade, ou em outras palavras, ao melhorar a manutenibilidade do *software*, por meio da manutenção preventiva, pode-se produzir *software*s de melhor qualidade.

Tratou-se, neste trabalho, de uma abordagem para a manutenção preventiva baseada em inspeções, refatoração e métricas de produto de *software*. Foram identificadas as falhas de manutenibilidade mais freqüentes numa amostra de *software*s orientados a objetos (OO), assim como as evidências da ocorrência de cada uma das falhas encontradas. Também foi estudado o comportamento de métricas de produto de *software* mediante a eliminação das falhas mais freqüentes.

A partir do estudo realizado, foram propostos modelos de documentação e artefatos de apoio a um processo de inspeção desde a atividade de *detecção de defeitos* até a atividade (final) de *acompanhamento*. Este texto também apresenta uma descrição detalhada do

processo pelo qual realizou-se o estudo mencionado, de maneira que é possível repetir livremente esse estudo com outras amostras, melhorando os resultados alcançados e até alcançando outros novos.

Conhecendo-se as falhas mais freqüentes em um certo contexto é possível promover melhorias no plano de qualidade e direcionar melhor os treinamentos. Tanto a identificação das evidências de falhas – que deram origem às descrições contidas na relação de falhas conhecidas – como as recomendações de correção, proporcionam a disseminação do conhecimento entre os desenvolvedores, contribuindo para que melhores resultados sejam alcançados nas inspeções.

Foi possível observar que os modelos de documentação (relatórios) – adaptados ao contexto da OO – propostos foram bem aceitos devido, principalmente, à sua objetividade, e o mesmo vale para o processo de inspeção como um todo. Percebe-se que o processo deve ser bem definido, porém leve, visto que as pessoas tendem a oferecer resistência a processos burocráticos e repletos de documentação. Acredita-se que a seção de *evidências* do modelo de relatórios contribuiu para essa aceitação ao permitir que o autor do *software* pudesse não só encontrar o ponto em que há uma falha, mas que também pudesse compreender os motivos pelos quais a falha foi assim considerada.

Constatou-se que as refatorações, por apresentarem uma abordagem disciplinada e com foco unicamente na melhoria da manutenibilidade, são uma ferramenta de grande utilidade para a prática da manutenção preventiva.

A aplicação de métricas de produto de *software* como um indicador da realização de correções, na atividade de acompanhamento, também mostrou-se uma prática viável. Porém, para que melhores resultados sejam alcançados, o repositório de métricas deve ser constantemente melhorado, tal qual o processo de inspeção como um todo. A melhoria do

processo de inspeção exige como esforço extra apenas a análise dos relatórios e métricas coletadas, visto que a aplicação do próprio processo gera os dados necessários para a análise.

Conclui-se, portanto, que tanto os artefatos de apoio quanto os modelos de documentação propostos, assim como o processo de inspeção, tiveram sua aplicabilidade comprovada e demonstraram potencial, como uma abordagem para a execução da manutenção preventiva, para o controle da complexidade do *software*.

Acredita-se, contudo, que por mais completo e por melhor que seja um processo de trabalho, não há técnica ou disciplina que possa substituir o comprometimento de um indivíduo em realizar um bom trabalho e em aperfeiçoar as suas habilidades constantemente.

7.2 Trabalhos Futuros

São sugeridos como trabalhos futuros os seguintes pontos que não puderam ser explorados ao longo deste trabalho:

- **Analisar mais amostras de *softwares***: o processo de realização do estudo descrito neste trabalho, composto basicamente pela identificação e correção de falhas de manutenibilidade, requer um certo esforço que limita o tamanho da amostra analisada. Analisar mais amostras pode aperfeiçoar os resultados alcançados.

- **Substituir os relatórios por uma ferramenta**: apesar de o modelo de relatórios de falhas ser sucinto, a substituição dos relatórios textuais por uma ferramenta específica pode tornar o processo mais dinâmico e facilitar a análise dos dados acumulados centralizando-os num repositório único. Sugere-se analisar a aplicabilidade de um rastreador de defeitos (*bug tracker*) para este fim. Há *software*s livres[31] para esse propósito, o que possibilita, inclusive, eventuais adaptações.

31 Exemplos: Bugzilla (http://www.bugzilla.org) e Trac (http://trac.edgewall.org/).

- **Automatizar a análise de métricas**: com a substituição dos relatórios por uma ferramenta específica, é interessante fazer com que a identificação das métricas mais freqüentes e a análise do comportamento das métricas mediante a eliminação de uma falha sejam automatizadas. Além disso, a ferramenta poderia fornecer diversos relatórios para que um ser humano pudesse melhorar o processo de inspeção e os artefatos de apoio. O relatório de todas as evidências conhecidas para uma determinada falha é um exemplo, assim como o relatório das correções já aplicadas para resolver certa falha de manutenibilidade.

Referências Bibliográficas

[Baranauskas, 1995] Baranauskas, M. C. C. **Procedimento, Função, Objeto ou Lógica? Linguagens de Programação vistas pelos seus Paradigmas**. In: VALENTE, José Armando (Org.) Computadores e conhecimento: repensando a educação, UNICAMP, 1995. Disponível em <http://www.nied.unicamp.br/publicacoes/separatas/Sep3.pdf>. Acesso em 29 de dezembro de 2007.

[Basili *et al.*, 1994] Basili, V., Caldiera, G., Rombach, H. D. **The Goal Question Metric Approach**, 1994. Disponível em <http://www.cs.umd.edu/~basili/publications/technical/T86.pdf>. Acesso em 04 de janeiro de 2008.

[Basili et. al, 1996] Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., Zelkowitz, M. **The Empirical Investigation of Perspective-Based Reading**, Empirical *Software* Engineering: An International Journal, vol. 1, n. 2, p. 133-164, 1996.

[Beck, 2004] Beck, Kent, **Programação Extrema Explicada: Acolha as mudanças**, Bookman, 2004.

[Binkley e Schach, 1998] Binkley, A. B., Schach, S. R., **Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures**. Proc. of the 1998 International Conference on *Software* Engineering, p. 452-455, 1998.

[Boegh *et al.*, 1999] Boegh, J.; Depanfilis, S.; Kitchenham, B.; Pasquini, A. **A method for *software* quality planning, control, and evaluation**. IEEE *Software*, vol. 16, n. 2, p. 69-77, 1999.

[Brooks, 1987] Brooks, Frederick P., **No Silver Bullet: Essence and Accidents of *Software* Engineering**, Computer, vol. 20, n. 4, 1987, p. 10-19.

[Bryant, 2000] Bryant, Antony. **'It's engineering Jim... but not as we know it' - *Software* Engineering: Solution for the *software* crisis, or part of the problem?** Proceedings of the 2000 International Conference, Junho 2000.

[Carneiro, 2003] Carneiro, Glauco F., **Usando Medição de Código Fonte para *Refactoring***, Dissertação de mestrado. Universidade Salvador. Abril, 2003. Salvador, Bahia.

[Cartwright e Shepperd, 2000] Cartwright, M., Shepperd, M., **An empirical investigation of an object-oriented *software* system**. IEEE Transactions on *Software* Engineering, vol. 26, p. 786-796, 2000.

[Chidamber e Kemerer, 1994] Chidamber, Shyam R.; Kemerer, Chris F., **A Metrics Suite for Object Oriented *Design***, IEEE Transactions on *Software* Engineering, vol. 20, n. 6, 1994, p. 476-493.

[Demeyer *et al.*, 2000] Demeyer, Serge; Ducasse, Stéphane; Nierstrasz, Oscar, **Finding Refactorings via Change Metrics**, in: Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM SIGPLAN Notices, vol. 35, n. 10, 2000, p. 166-177.

[Fagan, 1976] Fagan, M. E. ***Design* and Code Inspections to Reduce Errors in Program Development**. IBM Systems Journal, vol. 15, n. 3, 1976, p. 182-211.

[Figueiredo, 2005] Figueiredo, André Luís Gouvêa de, **ECO: Um ecossistema para o desenvolvimento ágil de sistemas *web***. Dissertação de mestrado apresentada ao Instituto de Ciências Matemáticas e de Computação (ICMC). Universidade de São Paulo. Abril de 2005. São Carlos, São Paulo.

[Fowler, 1999a] Fowler, M. **Refactoring: Improving the *Design* of Existing Code**. Addison-Wesley, 2000.

[GoF, 1994] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, ***Design* Patterns: Elements of Reusable Object-Oriented *Software***. Addison-Wesley, 1994.

[Horstmann e Cornell, 2004] Horstmann, C. S., Cornell, G. **Core Java: Volume I - Fundamentals**. Prentice Hall, 2004, 7 ed.

[Java, 2007] Sítio da empresa *Sun Microsystems* sobre a tecnologia Java. Disponível em <http://java.sun.com/>. Acesso em 27 de dezembro de 2007.

[Kataoka *et al.*, 2002] Kataoka, Y.; Imai, Y.; Andou, H.; Fukaya, T., **A Quantitative Evaluation of Maintainability Enhancement by Refactoring**, Proc. International Conf. *Software* Maintenance, p. 576-585,

2002.

[Laitenberger e Atkinson, 1999] Laitenberger, O., Atkinson, C. **Generalizing Perspective-Based Inspection to handle Object-Oriented Development Artifacts**. Proc. 21st international conference on *Software* engineering, p. 494-503, 1999.

[Laitenberger e DeBaud, 1998] Laitenberger, O., DeBaud, J-M., **An Encompassing Life-Cycle Centric Survey of *Software* Inspection**. Journal of Systems and *Software*, vol. 50, n. 1, p. 5-31, 1998.

[Mafra e Travassos, 2005] Mafra, Sômulo N.; Travassos, Guilherme H., **Técnicas de Leitura de *Software*: Uma Revisão Sistemática**, SBES 2005.

[McCabe, 1976] McCabe, T.J. **A Complexity Measure**. IEEE Trans. on *Software* Engineering, vol. 2, n. 4, p. 308-320, 1976.

[Mens e Tourwé, 2004] Mens, Tom; Tourwé, Tom, **A Survey of *Software* Refactoring**, IEEE Transactions on *Software* Engineering, vol. 30, n. 2, 2004, p. 126-139

[Mens *et al.*, 2003] Mens, T.; Demeyer, S.; Du Bois, B.; Stenten, H.; Van Gorp, P., **Refactoring: Current Research and Future Trends**. Language descriptions, Tools and Applications (LDTA), 2002.

[Opdyke, 1992] Opdyke, W. F., **Refactoring: A Program Restructuring Aid in *Design*ing Object-Oriented Application *Frameworks***, Tese de doutorado, Universidade de Illinois em Urbana-Campaign, 1992.

[Pressman, 2005] Pressman, Roger S., ***Software* Engineering: a practitioner's approach**. McGraw-Hill Higher Education, 6 ed., 2005.

[Rai *et al.*, 1998] Rai, Arun; Song, Haidong; Troutt, Marvin, ***Software* Quality Assurance: An Analytical Survey and Research Priorization**, 1998.

[Roberts *et al.*, 1997] Roberts, D.; Brant, J.; Johnson, R., **A refactoring tool for Smalltalk**, Theory and Practice of Object Systems , vol. 3, n. 4, 1997, p. 253-263

[Runeson e Isacsson, 1998] Runeson, P.; Isacsson, P. ***Software* quality assurance-concepts and misconceptions**. Proc. 24th Euromicro Conference, vol. 2, p. 853-859, 1998.

[Russel e Norvig, 2002] Russel, S., Norvig, P. **Artificial Intelligence: A Modern Approach**. Prentice Hall, 2002, 2 ed.

[Schefler, 1988] Schefler, W.C. **Statistics: Concepts and Applications**. Benjamin/Cummings, 1988.

[Schroeder, 1999] Schroeder, M. **A Practical Guide to Object-Oriented Metrics**. IEEE IT Professional, vol. 1, n. 6, p. 30-36, 1999.

[Shull *et al.*, 2000] Shull, F., Rus, I., Basili, V. **How Perspective-Based Reading Can Improve Requirements Inspections**. Computer, vol. 33, n. 7, p. 73-79, 2000.

[Simon *et al.*, 2001] Simon, F.; Steinbrückner, F.; Lewerentz, C., **Metrics Based Refactoring**, Proc. European Conf. *Software* Maintenance and Reengineering, p. 30-38, 2001.

[Singh, 1995] Singh, R. **The *Software* Life Cycle Process Standard**. Computer, vol. 28, n. 11, 1995, p. 89-90

[Sommerville, 2001] Sommervile, Ian, ***Software* Engineering**. Addison-Wesley, 6 ed., 2001.

[Subramanyam e Krishnan, 2003] Subramanyam, R.; Krishnan, M. S., **Empirical Analysis of CK Metrics for Object-Oriented *Design* Complexity: Implications for *Software* Defects**, IEEE Trans. on *Software* Engineering, vol. 29, p. 297-310, 2003.

[Walker, 1999] Walker, A. J. **A *software* quality perspective on the evolution of ISO 9001:1994 to ISO 9001:2000**. Proc. Fourth IEEE International Symposium and Forum on *Software* Engineering Standards, p. 58-66, 1999.

Bibliografia Consultada

Fowler, M. **Capítulo 15 não publicado do livro "Refactoring: Improving the *Design* of Existing Code"**. Addisson-Wesley, 2000. Disponível em <http://www.refactoring.com/rejectedExample.pdf>. Acesso em fevereiro de 2008.

Fowler, M., **UML Distilled: A Brief Guide to the Standard Modeling Language**. Addisson-Wesley Professional, 2 ed., 1999.

Martin, R. **OO *Design* Quality Metrics: an Analisys of Dependencies**. Disponível em <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>. Acesso em 30 de dezembro de 2007.

Paduelli, Mateus Maida, **Manutenção de *Software*: problemas típicos e diretrizes para uma disciplina específica**. Dissertação de mestrado apresentada ao Instituto de Ciências Matemáticas e de Computação (ICMC). Universidade de São Paulo. Maio de 2007. São Carlos, São Paulo.

Apêndice

Questionário utilizado no Estudo de Caso

A seguir, é apresentado o questionário utilizado no estudo de caso descrito no capítulo 6. Entre parênteses indica-se a quem são destinadas as perguntas.

(Inspetor)

1. Você considera que o cenário PBR o ajudou a saber o que estava procurando?

2. De maneira geral, você considerou o cenário PBR útil para a sua leitura? Por quê?

(Inspetor e Autor)

3. O documento de falhas conhecidas foi útil para você? Por quê?

4. Os exemplos fornecidos junto com a descrição de cada falha (no documento de falhas conhecidas) foram úteis ou são dispensáveis?

5. Qual a sua opinião sobre o formato proposto para o relatório de falhas? Faltou alguma informação importante a ser adicionada? Há informações requeridas que são dispensáveis?

(Autor)

6. O relatório de falhas facilitou a localização das mesmas? Foi possível entender, por meio do relatório, por que motivo as falhas foram assim consideradas?

7. O documento de recomendações de correções foi útil?

8. Alguma crítica ou comentário sobre o documento de recomendações?

(Inspetor e Autor)

10. Qual a sua opinião, de maneira geral, sobre o processo de inspeção? Tem alguma crítica a fazer? Algum comentário?

11. Imagine como seria executar o mesmo processo sem os artefatos de apoio (cenário PBR, relação de falhas conhecidas, etc). Isso faria alguma diferença?

12. Você acha que o trabalho poderia se tornar mais ágil se os relatórios fossem substituídos por uma ferramenta similar a um rastreador de defeitos (como o *Bugzilla*, por exemplo)?