FACULDADE ZACARIAS DE GÓES

CAMILO LOPES DE MEDEIROS NETO

IMPLICAÇÕES DA TÉCNICA DE REFATORAÇÃO EM DESENVOLVIMENTO E MANUTENÇÃO DE SOFTWARE

Valença

2008

CAMILO LOPES DE MEDEIROS NETO

IMPLICAÇÕES DA TÉCNICA DE REFATORAÇÃO EM DESENVOLVIMENTO E MANUTENÇÃO DE SOFTWARE

Monografia apresentada a Faculdade Zacarias de Góes – FAZAG como requisito parcial obrigatório para a obtenção do grau de Bacharel em Sistemas de Informação.

ORIENTADORA GERAL: PROFª. SIMONE P. CRUZ
ORIENTADOR ESPECÍFICO: PROF. ADONAI ESTRELA MEDRADO

Valença
2008

R586a Neto, Camilo Lopes de Medeiros
As implicações da técnica de refatoração em desenvolvimento
e manutenção de *Software*/Camilo Lopes de Medeiros Neto – Valença,2008

Monografia (Bacharel Sistemas de
Informação) - Faculdade Zacarias de Góes

CDD 370.1523

CAMILO LOPES DE MEDEIROS NETO

IMPLICAÇÕES DA TÉCNICA DE REFATORAÇÃO EM DESENVOLVIMENTO E MANUTENÇÃO DE SOFTWARE

Monografia apresentada a Faculdade Zacarias de Góes – FAZAG como requisito parcial obrigatório para a obtenção do grau de Bacharel em Sistemas de Informação.

Aprovada em 12 de dezembro de 2008.

BANCA EXAMINADORA

Prof. Adonai Estrela Medrado (Orientador Especifico)
Especialista em
Prof.:
Prof.:
Faculdade Zacarias de Góes - FAZAG

DEDICATÓRIA

A minha avó, minha mãe e meu avô pela oportunidade de realizar este trabalho.

AGRADECIMENTOS

Agradeço a minha avó e minha mãe por todo amor e apoio que me deram longo de minha vida, principalmente no Bacharelado, vibrando com minhas conquistas e incentivando nas minhas derrotas. Agradeço também pela oportunidade que me deram para exercer este trabalho. Outra pessoa que não posso deixar de agradecer é ao meu pai Camilo Lopes que não estar mais presente nessa vida o qual o dedicado com todo empenho e amor este trabalho. Amo vocês!

À minha noiva Efigênia Maurício, pelo amor, carinho, compreensão e motivação para comigo durante um ano, já que em alguns momentos durante esse tempo deixei de dar atenção devida por estar empenhado neste trabalho. E por entender o quão é importante pra me conseguir esse título. Amo você!

Aos professores Adonai Estrela e Simone Cruz pela competência demonstrada na orientação deste trabalho, pela atenção, empenho e parceria para que chegasse a conclusão com êxito.

E ao meu amigo Alberto Leal por toda ajuda que foi me dado desde o inicio do trabalho, como sugestões sobre o tema e pelas indicações bibliográficas. Desejo tudo de bom para você!

Ao meu colega de turma Márcio Santiago agradeço pela ajuda dada para que este trabalho fosse entregue ao orientador no momento que fiquei ausente das orientações presenciais. Desejo a você sucesso em sua carreira e na sua vida pessoal!

RESUMO

Estima-se que cerca de 50% do tempo de um engenheiro de *Software* é gasto com tarefas de manutenção e compreensão de código. A técnica de refatoração é o processo de alterar um *Software* para melhorar sua qualidade interna preservando seu comportamento. Sendo assim, essa técnica é considerada uma forma disciplinada de reorganizar o código facilitando o reuso e diminuindo o tempo gasto com as tarefas de manutenção. O objetivo geral desta monografia, é verificar as implicações da técnica de refatoração em desenvolvimento e manutenção de *Software,* analisando suas aplicações em sistemas *Java* e a contribuição para a manutenção e desenvolvimento do *Software*. Como objetivo especifico, tem-se a realização de um estudo bibliográfico visando descrever e analisar de forma qualitativa as implicações da técnica de refatoração no desenvolvimento e manutenção de *Software* e apresentar uma análise quantitativa através de um estudo de caso de literaturas existentes relacionando refatoração com o desenvolvimento e manutenção de *Software*. Para atingir os objetivos, será utilizada uma abordagem qualitativa, com levantamento bibliográfico e quantitativo por dados, onde a tabulação das informações serão no modo texto, tabelas, estudo de caso e códigos na linguagem *Java*. Os resultados obtidos é que a técnica de refatoração contribui para o desenvolvimento e manutenção de *Software*, melhorando a legibilidade do código e mantendo o comportamento do sistema, contribuindo assim para o custo do projeto e tempo de vida do *Software,* tornando o *Software* mais flexível para adição de novas funcionalidades sem desestruturar o código.

Palavras-Chave: Refatoração, Manutenção de *Software* e Desenvolvimento de *Software*

ABSTRACT

Esteem that about of 50% of the time of the engineer's *Software* is spent with task of maintenance and understanding of code. The technical of refactoring is the process of change a *Software* to improve your quality internal preserving your behavior, being, so this technical is considered a form disciplined of reorganize the code easing the reuse and reducing the time spent with the task of maintenance. The objective overview this monograph is verify the implication of technical of refactoring on development and maintenance of *Software* analyzing your implication on system *Java*, and the contribute to the maintenance and development's *Software* . How objective specific have the fulfillment of the study aimed bibliography and analyst described of form qualitative implication of technical of refactoring on development and maintenance of *Software*. To achieve the objective will be using an approach qualitative with survey bibliography and quantitative by data, where the tabulation of information will be on mode text, table, study case and code at *Java* language. The results obtained is that the technical refactoring contribute to the development and maintenance of *Software,* improving the clear of code and keeping the behavior of system.

Key-word: Refactoring, Maintenance *Software* and Development's *Software*

SUMÁRIO

CAPITULO 1 - INTRODUÇÃO

O custo e a complexidade de se manter um *Software* são amplamente reconhecidos. Estima-se que cerca de 50% do tempo de um engenheiro de *Software* é gasto com tarefas de manutenção e compreensão de código e que ao longo das últimas três décadas mais de 60% dos custos de desenvolvimento de *Software* das organizações foram gastos com manutenção. Refatoração é o processo de mudar um *Software* de tal forma que melhore a estrutura interna, sem contudo, alterar o comportamento externo. Portanto, é uma forma disciplinada de re-organizar o código, minimizando as chances de introduzir erros. Refatorar tem a vantagem de melhorar a estrutura do código, facilitando o reuso e diminuindo o tempo gasto com tarefas de manutenção. O termo refatoração é aplicado a sistemas orientados a objetos; para outros paradigmas de programação esse mesmo processo é descrito como reestruturação (MAIA,2004).

A refatoração ajuda a tornar o código mais legível e a resolver problemas de códigos mal escritos. Já no contexto da evolução de *Software*, a refatoração é usada para melhorar os atributos de qualidade como extensibilidade, modularidade e reusabilidade.

O tema desta monografia é avaliação das implicações da técnica de refatoração em desenvolvimento e manutenção de *Software*. Partindo-se desse contexto, segue a seguinte questão: quais as implicações da técnica de refatoração em desenvolvimento e manutenção de *Software*?

A principal motivação para a realização deste trabalho é auxiliar no desenvolvimento de sistemas orientados a objetos implementados na linguagem *Java* que possuem baixa reusabilidade de código, manutenção difícil e onde a adição ou adequação a novos requisitos demande muito esforço do engenheiro de *Software*, programador e analista de sistemas. Pontos que podem gerar grande custo ao projeto. Refatorando esses sistemas objetiva-se aumentar manutenibilidade, reusabilidade e o tempo de vida. Com a técnica de refatoração aplicada no desenvolvimento de *Software,* estima-se uma diminuição nos atrasos dos projetos e facilidade no processo de manutenção contribuindo desse modo para o mercado levando em conta custo x beneficio para as organizações.

Serão apresentados conceitos e técnicas que demonstrem como, quando e quais áreas devem ser aplicadas a refatoração em *Software,* a fim de contribuir para o desenvolvimento e manutenção.

A sociedade acadêmica pode se beneficiar com a técnica de refatoração nos estudos voltados para o desenvolvimento de *Software*, na criação de técnicas e ferramentas para refatoração na linguagem *Java.*

As hipóteses deste trabalho são:

a) a técnica de refatoração pode ser aplicada no desenvolvimento e manutenção do *Software*;
b) refatoração possibilita o acréscimo de nova funcionalidade de maneira fácil, evitando a desestruturação do código;
c) com o uso da refatoração se gasta menos tempo na manutenção do código, melhora sua legibilidade e estrutura.

Esta monografia tem como objetivo geral verificar as implicações da técnica de refatoração em desenvolvimento e manutenção de *Software,* analisando suas aplicações em sistemas *Java.*

Como objetivos específicos têm-se:

a) realizar um estudo bibliográfico, visando descrever e analisar de forma qualitativa as aplicações da técnica de refatoração no desenvolvimento e manutenção de *Software;*
b) apresentar uma análise quantitativa através de estudos de casos da literatura existente, relacionando refatoração com o desenvolvimento e manutenção de *Software*.

Visando atingir os objetivos propostos, escolheu-se uma abordagem qualitativa com um levantamento bibliográfico consultando livros, trabalhos acadêmicos e artigos na Internet. Em seguida, iniciou-se o processo quantitativo por dados, onde a tabulação das informações será no modo texto, tabelas, estudo de caso e códigos na linguagem *Java* como exemplos para as explanações.

Essa monografia foi dividia em seis capítulos. No capitulo 1 apresenta-se o objetivo, as hipóteses e as contribuições. É feito uma introdução sobre o tema e o cenário atual na área de desenvolvimento e manutenção *Software.*

No capítulo 2 apresentam conceitos sobre a Engenharia de *Software* que serviram de base para desenvolvimento desta monografia. São apresentados diretrizes que auxiliam na refatoração de sistemas, tais como, o processo de engenharia, qualidade, aplicações e manutenção de *Software.*

O conteúdo principal desta monografia, é abordado no capítulo 3o. É apresentado o conceito, a história, as justificativas para refotaração, os problemas identificados com a técnica, entre outros temas relacionados.

O capítulo 4, é composto por um catálogo de refatorações. Neste capítulo será apresentado algumas refatorações aplicada em código *Java*.

No capítulo 5, são apresentados alguns estudos de casos e como a técnica de refatoração contribuiu para o desenvolvimento e manutenção do *Software*.

No capítulo 6 encontram-se as considerações finais que comentam os resultados obtidos com este projeto.

CAPITULO 2 - ENGENHARIA DE SOFTWARE

A Engenharia de *Software* visa à criação de produtos de *Software* que atendam as necessidades de pessoas e instituições e, portanto, tenham valor econômico. Para isso, utiliza-se de conhecimentos científicos, técnicos e gerenciais, tanto teóricos quanto empíricos. Ela atinge seus objetivos de produzir *Software* com alta qualidade e produtividade, quanto é praticada por profissionais treinados e bem informados, utilizando tecnologias adequadas, dentro de processos que tirem proveito tanto da criatividade quando da racionalização do trabalho (PÁDUA, 2007).

Segundo Fritz (1969 apud PRESSMAN, 2006), a Engenharia de *Software* é a criação e a utilização de sólidos principios de engenharia, a fim de obter *Softwares* econômicos que sejam confiaveis e que trabalhem eficientemente em máquinas reais.

Para Pressman (2006), a engenharia de *Software* é uma tecnologia em camadas, que deve se apoiar num compromisso organizacional com a qualidade, sendo assim na Figura 1, este autor apresenta as camadas com foco na qualidade.

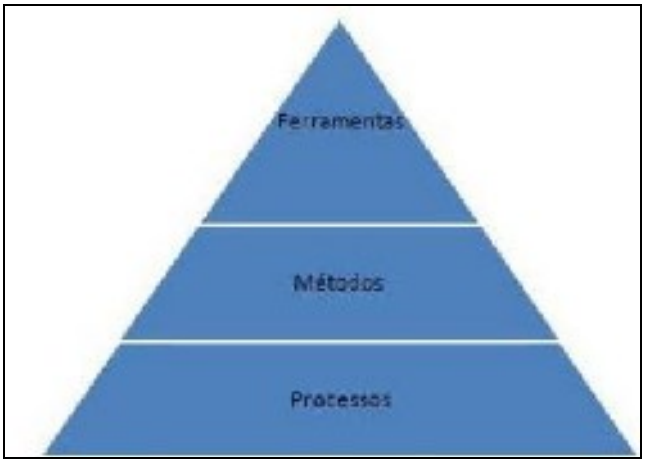

Figura 1 - Engenharia de *Software* em Camadas
Fonte: modificado de Pressman (2006).

Segundo Pressman (2006), o alicerce da engenharia de *Software* é a camada de processos. Esta camada define o que deve ser estabelecido para a efetiva utilização da tecnologia de engenharia de *Software.* Também são definidos os processos que formam a base para o controle gerencial do projeto e estabelece o contexto no qual os métodos técnicos são aplicados, os produtos de trabalho (dado, relatórios, formulários etc.) são produzidos. Ainda segundo este autor, a camada de métodos fornece a técnica de "como fazer" para construir *Softwares*. Essa área abrange um amplo conjunto de tarefas que incluem comunicação, análise

de requisitos, modelagem de projeto, construção de programas, teste e manutenção. A camada de ferramentas fornece apoio automatizado ou semi-automatizado para o processo e para os métodos.

2.1 QUALIDADE DE *SOFTWARE*

Segundo Geek (2008), qualidade não é apenas um diferencial de mercado para a empresa conseguir vender e lucrar mais, é um pré-requisito que a empresa deve conquistar para conseguir colocar o produto no mercado global. Na área de *Software,* a qualidade é crítica para sobrevivência e sucesso no mercado que está se desenvolvendo de forma global. Uma organização não se sobressairá no mercado global, a menos que produza *Software* de boa qualidade e seus clientes percebam essa qualidade. Segundo McCall (1977, apud PRESSMAN, 2006) os fatores de qualidade de um *Software* concentra-se em três aspectos importantes: suas caracteristicas operacionais, sua habilidade de passar por modificações e sua adaptabilidade a novos ambientes. Na Tabela 1 Geek (2008) mostra algumas razões para se considerar um *Software* de qualidade.

Razão	Justificativa
Qualidade é competitividade	A única maneira de diferenciar o produto do competidor é pela qualidade do *Software* e do suporte que é fornecido juntamente.
Qualidade é essencial para a sobrevivência	Clientes estão pedindo por qualidade. Se a empresa não tiver habilidade de sobreviver em um mercado altamente competitivo, ela está em débito com o mercado.
Qualidade é custo/benefício	Um sistema de qualidade direciona para o aumento da produtividade e permanentemente reduz custos, habilitando o gerenciamento para reduzir a correção de defeitos, dando ênfase à prevenção.
Qualidade retém consumidores e aumenta lucros	Pouca qualidade normalmente custa muito mais do que contratar mais desenvolvedores e ainda continuar sem qualidade. A maioria dos consumidores não tolerará falta de qualidade e irão procurar outros desenvolvedores.

Tabela 1 - Razões para qualidade de um *Software.*
Fonte: Geek (2008)

Como apresentado na Tabela 1, um *Software* de qualidade é um ponto importante tanto a nível de cliente quanto para as empresas de desenvolvimento de *Software*.

Para Geek (2008), os princípios da qualidade de *Software* são:

a) tentar prevenir defeitos ao invés de consertá-los;
b) a segurar que os defeitos sejam corrigidos o mais rápido possível;

c) estabelecer e eliminar as causas, bem como os sintomas dos defeitos.

Segundo Geek (2008), a Organização Internacional de Padrões (ISO) publicou uma norma que representa a atual padronização mundial para a qualidade de produtos de *Software*. Esta norma chama-se ISO/IEC 9126 e foi publicada em 1991. A tradução para o Brasil foi publicada em agosto de 1996 como NBR 13596. A norma NBR 13596 lista um conjunto de características que devem ser verificadas em um *Software* para que seja considerado um "*Software* de qualidade". A Tabela 2 apresenta normas nacionais e internacionais na área de qualidade de *Software*:

Normas	Comentário
ISO 9126	Características da qualidade de produtos de *Software*.
NBR 13596	Versão brasileira da ISO 9126
ISO 14598	Guias para a avaliação de produtos de *Software*, baseados na utilização prática da norma ISO 9126
ISO 12119	Características de qualidade de pacotes de *Software*
IEEE P1061	*Standard for Software Quality Metrics Methodology* (produto de *Software*)
ISO 12207	*Software Life Cycle Process*. Norma para a qualidade do processo de desenvolvimento de *Software*.
NBR ISO 9001	Sistemas de qualidade - Modelo para garantia de qualidade em Projeto, Desenvolvimento, Instalação e Assistência Técnica (processo)
NBR ISO 9000-3	Gestão de qualidade e garantia de qualidade. Aplicação da norma ISO 9000 para o processo de desenvolvimento de *Software*.
NBR ISO 10011	Auditoria de Sistemas de Qualidade (processo)
CMM	*Capability Maturity Model*. Modelo da SEI (Instituto de Engenharia de *Software* do Departamento de Defesa dos EEUU) para avaliação da qualidade do processo de desenvolvimento de *Software*.
SPICE ISO 15504	Projeto da ISO/IEC para avaliação de processo de desenvolvimento de *Software*.

Tabela 2 - Normas nacionais e internacionais na área de qualidade de *Software*.
Fonte: Geek (2008)

A escolha da norma a ser usada em um projeto de *Software* deve ser feita pelo engenheiro de *Software* ou gerente do projeto. Cabe a cada profissional decidir qual(is) a(s) norma(s) apresentada (s) na Tabela 2 devem fazer parte do projeto.

A Tabela 3 mostra os seis grandes grupos de características que devem ser verificados em um *Software* para que seja considerado um *Software* de qualidade. Os dados apresentados nesta tabela são de caráter técnico. Desta forma um profissional que trabalha com desenvolvimento de *Software* não terá dificuldade em entendê-los.

Para um desenvolvedor de *Software* é extremamente importante realizar as "perguntas chave" citadas na Tabela 3, a fim de construir um *Software* de qualidade,

atendendo as abordagens apresentadas na Tabela 1. Por outro lado Glass (1998 apud PRESSMAN, 2006, p. 579) argumenta “que a qualidade é importante, mas se o usuário não está satisfeito, nada mais realmente importa”.

Característica	Sub-característica	Pergunta chave para a sub-característica
Funcionalidade (satisfaz as necessidades?)	Adequação	Propõe-se a fazer o que é apropriado?
	Acurácia	Faz o que foi proposto de forma correta?
	Interoperbilidade	Interage com os sistemas especificados?
	Conformidade	Está de acordo com as normas, leis, etc.?
	Segurança de acesso	Evita acesso não autorizado aos dados?
Confiabilidade (é imune a falhas?)	Maturidade	Com que freqüência apresenta falhas?
	Tolerância a falhas	Ocorrendo falhas, como ele reage?
	Recuperabilidade	É capaz de recuperar dados em caso de falha?
Usabilidade (é fácil de usar?)	Intelegibilidade	É fácil entender o conceito e a aplicação?
	Apreensibilidade	É fácil aprender a usar?
	Operacionalidade	É fácil de operar e controlar?
Eficiência (é rápido e "enxuto"?)	Tempo	Qual é o tempo de resposta, a velocidade de execução?
	Recursos	Quanto recurso usa? Durante quanto tempo?
Manutenibilidade (é fácil de modificar?)	Analisabilidade	É fácil de encontrar uma falha, quando ocorre?
	Modificabilidade	É fácil modificar e adaptar?
	Estabilidade	Há grande risco quando se faz alterações?
	Testabilidade	É fácil testar quando se faz alterações?
Portabilidade (é fácil de usar em outro ambiente?)	Adaptabilidade	É fácil adaptar a outros ambientes?
	Capacidade para ser instalado	É fácil instalar em outros ambientes?
	Conformidade	Está de acordo com padrões de portabilidade?
	Capacidade para substituir	É fácil usar para substituir outro?

Tabela 3 - Conjunto de Características que devem ser verificadas em um *Software* para que seja considerado "*Software* de Qualidade”.
Fonte: (GEEK, 2008)

2.2 APLICAÇÕES DO *SOFTWARE*

Segundo Pressman (2006), o *Software* pode ser aplicado a qualquer situação em que um conjunto previamente especificado de passos procedimentais tiver sido definido. Desenvolver categorias genéricas para as aplicações de *Software* é uma tarefa um tanto difícil. À medida que a complexidade do *Software* cresce, desaparece a clara divisão em compartimentos. Na Tabela 4 este autor apresenta algumas áreas de aplicações do *Software*.

Área	Aplicações
Software de sistemas	Coleção de programas escritos para dar apoio a outros programas. Por exemplo: compiladores, editores etc.
Software aplicação	Um *Software* que monitora/analisa/controla eventos do mundo real.
Software comercial	O processamento de informações comerciais é a maior área particular de aplicação de *Software*. Por exemplo: folha de pagamento, estoque etc.
Software científico e de engenharia	O *Software* científico e de engenharia tem sendo caracterizado por algoritmos de processamento de números.
Software embutido	É usado para controlar produtos e sistemas para os mercados industriais e de consumo. Por exemplo: funções digitais em automóveis.

Tabela 4 - Áreas e aplicações de *Software*.
Fonte: Pressman (2006).

As áreas citadas na Tabela 4 e apresentadas por Pressman (2006), são as categorias de *Software,* de computadores que apresentam desafios contínuos para os engenheiros de *Software* desde a fase de desenvolvimento até o processo de manutenção.

2.3 MANUTENÇÃO DE *SOFTWARE*

De acordo com Pressman (2006), a manutenção de Software pode ser responsável por mais de 70% de todo o esforço despendido por uma organização de Tecnologia da Informação voltada para desenvolvimento. Essa porcentagem continua aumentando à medida que mais *Softwares* é produzido. Para este autor, manutenção de *Software* é bem mais que "consertar erros". Cerca de 20% de todo trabalho de manutenção é gasto "consertando erros" e os 80% restantes, são gastos adaptando sistemas existentes a modificações no seu ambiente externo, fazendo melhorias solicitadas por usuário e submetendo uma aplicação à reengenharia para uso futuro. Quando a manutenção abrange todas essas atividades é relativamente fácil ver porque absorve tanto esforço.

Para Pressman (2006), há quatro atividades que são levadas a efeito depois que o *Software* é liberado para uso. São elas:

a) Manutenção: ocorre porque não é razoável presumir que as atividades de testes do *Software* apresentará todos os erros latentes num grande sistema. Durante o uso de qualquer programa, erros ocorrerão e serão relatados ao desenvolvedores;
b) Mudança rápida: no aspecto da computação com novos sistemas operacionais, novas gerações de *hardware* por exemplo;
c) Um *Software* bem-sucedido: à medida que o *Software* é usado, recomendações de novas capacidades de modificações em funções existentes e de ampliações gerais são recebidas dos usuários;

d) Alteração de um *Software*: quando acontecem alterações para melhorar a confiabilidade ou a manutenibilidade futura, ou para oferecer uma base melhor para futuras ampliações.

Segundo Sommer-Ville (2007), as mudanças feitas no *Software* podem ser simplesmente para corrigir erros de codificação, podem ser mudanças extensas para corrigir erros de projetos ou melhorarias significativas para corrigir erros de especificação para acomodar novos requisitos. Segundo este autor existem três tipos diferentes de manutenção de *Software*:

a) Manutenção para reparo de defeitos de *Software*: que podem envolver correção de erros de codificação, erros de projetos que podem envolver a reescrita de vários componentes dos programas e os erros de requisitos que podem ser necessário o reprojeto do sistema existente;
b) Manutenção para adaptar o *Software* a um ambiente operacional diferente: esse tipo de manutenção é necessária quando algum aspecto do ambiente do sistema, como o *hardware*, plataforma do sistema operacional ou outro *Software* de apoio mudam, sendo assim o sistema da aplicação deve ser modificado para lidar com essas mudanças;
c) Manutenção para adicionar funcionalidade ao sistema ou modificá-la: este tipo de manutenção é necessária quando os requisitos do sistema mudam em resposta as mudanças organizacionais ou de negócios.

Pesquisas feitas por Lientz e Swanson(1980, apud SOMMER-VILLE, 2007) e Nosek e Palvia (1990, apud SOMMER-VILLE, 2007) sugerem que aproximadamente 65% das manutenções estão relacionadas à implementacoes de novos requisitos , 18% a mudanças do sistemas para adapta-lo a um novo ambiente operacional e 17% a corrigir defeito do sistema, conforme apresentado no Figura 2.

Para aqueles autores, reparo de defeitos do sistema não é uma atividade de manutenção dispendiosa. Porém promover a evolução do sistema para lidar com novos ambientes e os novos requisitos ou a alteração destes consomem mais esforço na manutenção.

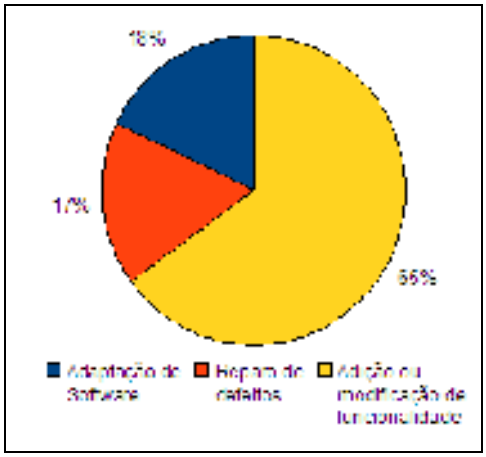

Figura 2 Distribuição de esforços de manutenção.
Fonte:Sommer-Ville(2007)

Segundo Pressman (2006), o custo da manutenção de *Software* tem aumentado firmemente durante os últimos 20 anos. Durante a década de 1970, a manutenção para uma organização de sistemas de informação era responsável por um índice entre 35% à 40% do orçamento de *Software* Esse valor pulou para aproximadamente 60% durante a década de 1980. Se nada for feito para melhorar o processo de manutenção, muitas empresas gastarão 80% de seus orçamentos de *Software* em manutenção.

CAPITULO 3 - REFATORAÇÃO

Refatoração é o processo de reestruturar o sistema sem alterar suas funcionalidades. Segundo Fowler (2004), refatoração torna o *Software* mais fácil de entender e modificar. A refatoração não altera o comportamento observável do *Software*. Esse ainda executa a mesma função de antes. Qualquer usuário seja ele final ou programador não é capaz de dizer que algo mudou.

Segundo Nunes (2003), com a refatoração pode-se ter uma estrutura ruim e redesenhá-la dentro de um código bem estruturado, onde cada passo é executado alterando um atributo de uma classe para outra, movendo um trecho de código de um método para dentro de um novo, colocando um método acima ou abaixo de sua hierarquia atual. Para este autor, o efeito cumulativo destas pequenas alterações pode melhorar radicalmente a estrutura do código deixando robusta.

3.1 HISTÓRICO

Os dois primeiros autores a reconhecer a importância da refatoração foram Cunningham e Beck (2000 apud FOWLER, 2004), que trabalhavam com *Smalltalk*[1] no processo de desenvolvimento de *Software* a partir dos anos de 1980. Estes autores perceberam que a refatoração era importante na melhoria de sua produtividade e desde então trabalharam com refatoração aplicando-a a projetos de *Software* sérios e refinando o processo (FOWLER, 2004).

Em seguida Ralf Johnson professor na *University of Illinois* em *Urbana-Champaing* deu início ao desenvolvimento de *frameworks*[2] de *Software* usando a técnica de refatoração em 1992 (GAMMA; JOHNSON, 1995). Este autor explorou como a refatoração pode ajudar a desenvolver um *framework* eficiente e flexível (FOWLER, 2004).

Segundo Fowler (2004), outro momento que fez parte da história da refatoração foi o estudo de um construtor de ferramentas para refatoração. Opdyke (1992), estudante de doutorado da *University of Illinois* em *Urbana-Champaing,* investigou como as refatorações

[1] *Smalltalk* "é uma linguagem de programação orientada a objeto fracamente tipada com um ambiente dinâmico que permite escrever rapidamente *Software* altamente funcional" (FOWLER, 2004).
[2] *Framework* "é um conjunto de classes implementadas em uma linguagem especifica usadas para auxiliar o desenvolvimento de *Software*" (SCHMIDT, 1997).

seriam úteis para o desenvolvimento de um *framework* em C++[3] que preservassem a semântica e como uma ferramenta poderia implementa essas idéias.

Brant e Roberts (2004) levaram as idéias das ferramentas na refatoração muito mais longe, produzindo uma ferramenta de refatoração para *Smalltalk*. Esses autores estenderam o conceito de refatoração de Opdyke (1992), acrescentando pós-condições que cada refatoração deve obedecer. Essas pós-condições descrevem o que deve ou não haver depois da aplicação da refatoração e podem ser usadas para derivar pré-condições de refatorações compostas, calcular dependências entre refatorações e realizar a quantidade de análises que refatorações posteriores em uma seqüência devem realizar para garantir que preservem o comportamento (FOWLER, 2004).

Uma metodologia que foi responsável pelo crescimento da visibilidade dar refatoração foi *Extreme Programming (*XP). Nessa metodologia deve-se aplicar refatorações até que o caso de uso possa ser implementado de uma maneira coerente. Um dos principais pilares de XP é a continua e agressiva aplicação de refatorações. Sem elas, essa metodologia não funcionaria (MAIA, 2004).

3.2 JUSTIFICATIVAS PARA REFATORAÇÃO

Para Fowler (2004), refatorar é uma ferramenta valiosa que ajuda a não desestruturação do código, porém a técnica de refatoração não cura todos os problemas de *Software*, mas é uma ferramenta valiosa.

Segundo Fowler (2004), refatora-se para:

a) melhorar o projeto do *Software*: um projeto sem refatoração termina por se deteriorar. À medida que as pessoas alteram o código e sem compreende-lo totalmente acabam desestruturando. Através da técnica de refatoração consegue garantir a estruturar do código;
b) ter um o *Software* mais fácil de entender a nível de código: eliminado códigos duplicados e facilidade na localização de falhas;
c) programar mais rapidamente: com o melhoramento do código, do projeto e facilidade na localização de falhas ajuda no processo de desenvolvimento do *Software,* já que a base para a construção de um *Software* com velocidade está diretamente ligada a construção de um bom projeto.

[3] C++ é uma linguagem de programação com o nível de abstração relativamente elevado, longe do código da máquina e mais próximo da linguagem humana (DEITEL,2006)

Segundo Beck (2000 apud FOWLER, 2004, p.58), “Programas têm dois tipos de valor: o que eles fazem por você hoje e o que podem fazer por você amanhã, na maior parte das vezes quando estamos desenvolvendo um *Software* focalizamos no que queremos que o programa faça hoje.”

Ainda segundo aquele autor, “Se você consegue fazer hoje o trabalho de hoje, mas o faz de uma maneira na qual possivelmente não conseguirá fazer amanhã o trabalho de amanhã, então você perde”. Para este autor uma situação como apresentada acontece quando se tem programas difíceis de ler a nível de código, adição de novas funcionalidades requer alteração de código existente a solução proposta é aplicar a refatoração.

Beck (2000 apud FOWLER, 2004), identifica quatro situações que torna os programas difíceis de modificar como apresentado na Tabela 5.

Situação	Difíceis de modificar
Baixa legibilidade	Dificulta o trabalho do desenvolvedor durante o desenvolvimento e manutenção
Lógica duplicada	Difícil modificação, onde alterar uma das lógicas tem que sair buscando às outras para realizar alteração
Necessidade de alterar códigos existentes para inclusão novas funcionalidades	Programas que para inclusão de novas funcionalidades, requerem a alteração de código existente
Lógica condicional complexa	Quando existem muitas condições aninhadas dificultando o entendimento e a manutenção do código.

Tabela 5 - Partes que tornam os programas difíceis de trabalhar.
Fonte: BECK (2000 apud FOWLER, 2004).

Para Beck (2000 apud FOWLER, 2004), refatorar é o processo de pegar um programa em produção e agregar a ele valor, não por meio da alteração de seu comportamento mais dando qualidades que nos permitem continuar desenvolvendo rapidamente. É preciso ter programas fáceis de ler e que tenham a lógica especificada em apenas um lugar.

3.3 PROBLEMAS COM REFATORAÇÃO

Segundo Fowler (2004), não há experiências em quantidade suficiente para definir onde as limitações se aplicam à técnica de refatoração. Este autor considera que as áreas descritas na Tabela 6 são difíceis de aplicar a técnica de refatoração.

Área	Motivo
Banco de dados	É uma área problemática para refatoração. As maiorias das aplicações comerciais são altamente acopladas ao esquema do banco de dados que as suporta. Esse é um motivo pelo qual bancos de dados são difíceis de modificar. O outro motivo é a migração de dados, onde alterar o esquema do banco de dados lhe força a migrar os dados, o que pode ser uma tarefa longa e pesada.
Interface	Um problema com a interface é se estiver sendo usada por código que não é possível encontrar e alterar. Sendo assim se torna uma interface publicada onde não é mais possível alterá-la com segurança e sim apenas modificar a chamada nos métodos que a invocam e isso dificulta o processo de refatoração. Se uma refatoração alterar uma interface publicada tem que conservar tanto a interface antiga quanto a nova e isso leva ao caminho da duplicação de código.
Linguagens não orientadas a objetos	São mais difíceis de reestruturar, pois fluxos de controle e de dados são fortemente interligados e por causa disso as reestruturações são limitadas a nível de função ou bloco de código.

Tabela 6 Áreas difíceis de aplicar a técnica de refatoração.
Fonte: Fowler (2004).

3.4 APLICAÇÃO DA TÉCNICA DE REFATORAÇÃO NO DESENVOLVIMENTO E MANUTENÇÃO DE *SOFTWARE*

Para Fowler (2004), decidir quando começar a refatorar e quando parar é tão importante quanto a mecânica de refatoração. Explicar como apagar uma variável de instância ou criar uma hierarquia são questões simples, porém tentar explicar quando deve-se fazer, não é algo tão consolidado.

As indicações apresentadas na Tabela 7 por Fowler (2004), são problemas que podem ser resolvidos através da refatoração, mas cabe ao analista de sistemas, engenheiro de *Software* e programador a própria percepção sobre quantas variáveis de instâncias são demais, quantas linhas de código em um método são excessiva.

Para Nunes (2003), existem três situações especificas em que refatorar é uma obrigação para o desenvolvedor de *Software,* que são:

a) adição de uma nova função: essa é uma situação comum para aplicação da técnica de refatoração. Se tiver um código organizado e com uma boa legibilidade então no momento que for necessário adicionar uma nova função esta tarefa será mais simples e rápida de ser executada;

b) correção de um erro: com uma estrutura de código clara e organizada a localização de erros é simples. Ao ocorrer um erro ficará mais simples descobrir qual bloco de código falhou e por conseqüência será mais fácil corrigi-lo com o código refatorado;
c) revisão de código: algumas organizações utilizam revisões de código para expandir o conhecimento do grupo de desenvolvimento, onde desenvolvedores mais experientes passam seus conhecimentos sobre o sistema para os mais novos. Essa técnica visa à melhora na estrutura do código do sistema e com refatoração isso vira um processo constante.

Situação	Momento de refatorar	Justificativa
Código duplicado	Quando existe o mesmo código em mais de um lugar, dificultando a manutenção e aumentando o número de falhas durante a manutenção do *Software*.	Buscar um meio de unificá-los, melhora a legibilidade do código e o processo de manutenção.
Método longo	Quando um método tem muito código.	Quanto maior for o método, mais difícil é entendê-lo.
Classes grandes	Quando uma classe tem muitas variáveis de instâncias.	Há existência de muitas variáveis de instancia criar um "solo fértil" para o código duplicado.
Lista de parâmetros longa	Quando um método tem muitos parâmetros.	Dificulta o entendimento, porque se tornam inconsistente e difíceis de usar, já que alteração sempre acontecerá a medida que precisar de mais dados. O recomendável é usar objetos para que eles possam obter tudo que for preciso.
Comando *switch*	Quando existir o comando switch que pode ser alterado por polimorfismo ou enumeração.	Os comandos *switch* essencialmente levam a duplicação. É comum encontrar o mesmo comando *switch* espalhado por diversos lugares do mesmo programa.
Comentários	Quando existir comentários supérfluos	Comentários que estão no código sem explicar o porquê, são considerados supérfluos. O comentário deve ser usado quando não sabe o que fazer e indicar áreas sobre os quais o programador não esta seguro. Um comentário deve dizer o porquê fez algo e que tipo de informação ajuda os futuros modificadores do *Software*.

Tabela 7 - Quando deve refatorar um *Software*.
Fonte: Fowler (2004).

3.5 REFATORAÇÃO E MANUTENÇÃO DE *SOFTWARE*

Segundo Guimarães (1983, apud SOMMER-VILLE, 2007), os custos de manutenção como parte dos custos de desenvolvimento variam de um domínio de aplicacao para outro. Para este autor um exemplo são os sistemas em tempo real nos quais os custos de manutenção podem ser até quanto vezes mais altos do que os custos de desenvolvimento os requisitos de alta robustez e desempenho desses sistemas fazem com que os modulos tenham de ser muito acomplados e, por isso, dificieis de serem modificados.

Fowler (2004), considera que a refatoração pode ser considerada uma técnica ou ferramenta de auxilio no desenvolvimento e manutenção, contribuindo para o tempo de vida do *Software*, já que o acréscimo de novas funcionalidades é de maneira fácil e rápida. Ainda para aquele autor um sistema mal projetado normalmente precisa de mais código para fazer as mesmas coisas muitas vezes porque o mesmo código foi duplicado em diversos lugares diferentes. Quanto mais difícil é visualizar o projeto a partir do código, mais difícil será preservá-lo e mais rapidamente ele se desestruturará. A eliminação de código duplicado é um aspecto importante na melhora do projeto, reduzindo a quantidade de código há grande diferença sobre a manutenção desse projeto.

Fowler (2004), defende quanto mais código houver em um *Software*, mas difícil será modificá-lo corretamente.

> [...] Reduzir a quantidade de código faz, todavia, uma grande diferença sobre a manutenção desse código. Quanto mais código houver, mais difícil será modificá-lo corretamente. Há mais código para ser entendido. Você altera trecho de código aqui, e o sistema não faz o esperado porque faltou alterar em outro lugar que faz a mesma coisa em um contexto levemente diferente [...] (FOWLER, 2004, p. 54).

Segundo este autor, a refatoração ajuda a desenvolver *Software* mais rapidamente porque evita que o projeto do sistema se deteriore, melhorando o processo de manutenção e otimização do tempo gasto no desenvolvimento.

3.6 A IMPORTÂNCIA DOS TESTES PARA A REFATORAÇÃO

Pressman (2006) afirma que teste é um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. Para aquele autor teste de *Software* é um elemento de aspecto amplo que refere-se a verificação e validação. Segundo este autor, verificação refere-se ao conjunto de atividades que garantem que o *Software*

implementa corretamente uma função especifica e validação refere-se atividades diferentes que garantem que o *Software* construído corresponde aos requisitos do cliente. Contudo, para este autor os testes oferecem efetivamente o último reduto no qual a qualidade podem ser avaliada e erros podem ser descobertos. Nas palavras de Miller (1977 apud PRESSMAN, 2006, p.290),

> [..]relacionar o teste de *Software* à garantia de qualidade, afirmando que a motivação subjacente ao teste de programa é afirmar a qualidade do *Software* com métodos que podem ser aplicados econômica e efetivamente tanto a sistemas de grande porte quanto de pequeno porte [...]

Para Sommer-Ville (2007), a meta do teste de *Software* é convencer os desenvolvedores e clientes do sistema de que o *Software* é bom o suficiente para o uso operacional. Para este autor, o teste de *Software* é um processo voltado a atingir a confiabilidade do *Software*. Na Figura 3 aquele autor apresenta um modelo geral do processo para execução de teste.

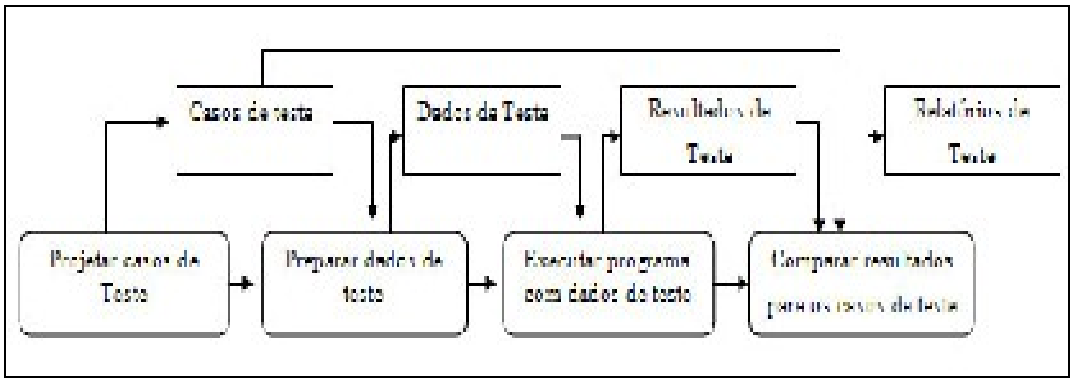

Figura 3 - Modelo de processo de Teste de Software
Fonte: Sommer-Ville(2007).

Segundo Sommer-Ville(2007), casos de teste são especificações de entradas as saídas esperadas mais a declaração do que está sendo testado. Dados de teste são entradas preparadas para o teste de sistema podendo muitas vezes ser gerado automaticamente. A saída dos testes pode somente ser prevista por pessoas que compreendem o que o sistema deve fazer.

Segundo Fowler (2004), bons testes aumenta bastante a velocidade de programação. Para este autor o maior tempo gasto é depurando o código, há casos de levar um dia ou mais procurando falha. Consertar a falha normalmente é um processo rápido, mas encontrá-la pode se tornar uma tarefa difícil.

Para Fowler (2004), o processo de refatoração requer testes. Refatorar necessita-se escrever teste a fim de manter o comportamento do *Software* após a refatoração. Em *Software*

Java os testes podem ser inseridos no *main,* onde cada classe deve ter um método *main* que teste a classe, dessa maneira criando testes que sejam totalmente automáticos e que possam verificar seus próprios resultados. Outra abordagem é construir classes de teste separadas que trabalhem em um *framework* para tornar o processo de teste mais fácil. Para *Software Java* aquele autor recomenda o *framework* JUnit (2008), que permite a realização de teste em *Software Java*.

A conclusão de Fowler (2004), é que um conjunto de teste é um detector poderoso de falhas que diminui o tempo que se leva para encontrá-las, sendo assim uma ferramenta crucial para desenvolvimento e uma pré-condição para a refatoração.Na Figura 4 apresenta os resultados possíveis de teste de *Software*.

Figura 4 - Resultados teste de Software.
Fonte: O autor.

3.7 REFATORANDO COM SEGURANÇA

Segundo Opdyke (1995 apud FOWLER, 2004), a segurança é uma preocupação, especialmente para organizações desenvolvendo e evoluindo sistemas grandes. Em muitos *Softwares*, há considerações financeiras, legais e éticas obrigatórias para fornecer serviço contínuo, confiável e livre de erros.

Uma refatoração segura é aquela que não danifique o programa (OPDYKE 1995, apud FOWLER, 2004, p. 332). De modo que a refatoração é aplicada para reestruturar um programa, sem alterar o comportamento observável, já que o mesmo deve desempenhar as mesmas funcionalidades que tinha antes do processo de refatoração.

Há diversas opções para refatorar com segurança um *Software*, mas Opdyke (1995, apud FOWLER, 2004) destaca a seguir como refatorar com segurança um *Software*:

a) O desenvolvedor do *Software* deve confiar nas suas habilidades de codificação;
b) Confiar que o compilador irá identificar os erros;
c) Confiar que o conjunto de teste desenvolvido irá encontrar erros, não encontrados pelo compilador e pelo próprio profissional;
d) Confiar que a revisão de código irá identificar erros não encontrados pelo compilador, conjunto de teste e pelo desenvolvedor;

Para Opdyke (1995, apud FOWLER, 2004, p. 332) as abordagens voltadas para segurança têm por objetivo garantir que a refatoração não introduza novos erros em um programa. Essas abordagens não detectam ou consertam falhas que estavam no programa antes dele ser refatorado. Entretanto refatorar pode tornar mais fácil localizar tais falhas e corrigi-las. Toda a verificação de segurança pode ser implementada sob a proteção de uma ferramenta de refatoração.

3.8 FERRAMENTAS PARA REFATORAÇÃO

Refatorar com suporte de uma ferramenta automatizada é diferente da refatoração manual. Mesmo com a rede de segurança de conjunto de testes, a refatoração não automatizada consome tempo (ROBERTS, 1997 apud FOWLER, 2004).

Para Maia (2004), as ferramentas que automatizam o processo diminuem o risco de erros e inconsistência no código, além de poupar um grande trabalho em se tratando de sistemas com centenas ou milhares de linhas de códigos.

Roberts (1997 apud FOWLER, 2004) defende que com as ferramentas de refatoração automática o projeto se torna mais elástico, já que alterá-lo é muito menos custoso, sendo assim estender o projeto adicionar flexibilidade no futuro sem grandes custos. Para este autor o principal propósito de uma ferramenta de refatoração é permitir ao programador, refatorar código sem ter que testar novamente o programa. A Tabela 8 apresenta algumas ferramentas que implementam a técnica de refatoração. Com o crescimento do número de IDE[4] para *Java*, algumas ferramentas de refatoração foram criadas como *plugins*[5] para esses ambientes que é o caso do JFactor (JFactor) e RefactorIt (refactorit) apresentado na Tabela 8.

Ferramenta	Descrição
	É uma ferramenta profissional que permite refatorar o código. Permite

[4] IDE é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de *Software* com o objetivo de agilizar este processo (DEITEL, 2006).
[5] *Plugins* é um programa de computador que serve para adicionar funções a outros programas maiores, provendo alguma funcionalidade especial ou muito especifica (DEITEL,2006).

Ferramenta	Descrição
XRefactoring (xref-tech)	trabalhar nas plataformas *Unix* e *Windows*. As linguagens compatíveis são C[6] e *Java* essa ferramenta possibilita a geração de documentação no formato HTML. Algumas funções de refatoração encontradas são: renomeação de pacotes, classes, variáveis, métodos, extração de métodos.
RefactorIT (refactorit)	É uma ferramenta para refatoração que pode integrada em varias IDEs. Algumas técnicas de refatoração que pode aplicadas: renomeação de classes, métodos; extração de métodos, movendo classes etc.
JFactor (JFactor)	É uma ferramenta que permite a aplicação automática da refatoração a programas em *Java*. As técnicas de refatoração aplicáveis: extração de método, renomeação de métodos e variáveis, extração de variáveis etc.

Tabela 8 - Ferramentas para Refatoração.

Fonte: (FOWLER, 2004).

[6] C é uma linguagem imperativa e procedural, para implementação de sistemas (RITCHIE,1996)

CAPITULO 4 - APLICANDO REFATORAÇÃO EM CÓDIGOS *JAVA*

Nesta seção será apresentado como aplicar a técnica de refatoração em código *Java*. Os métodos aqui foram selecionados a fim de serem usados no ambiente de desenvolvimento de um *Software*. É importante ressaltar que as técnicas aplicadas neste capitulo não são algo estático, ou seja, elas podem sofrer modificações seja pela evolução da linguagem, ou por um novo processo de refatoração do código.

4.1 APRESENTAÇÃO

De acordo com as técnicas apresentadas por Fowler (2004), a apresentação das refatorações será feita da seguinte forma:

a) Nome: será dado um nome ao caso de refatoração, sendo que este deve dar uma idéia do que o método faz e servirá de referência em outras partes desse trabalho;
b) Motivação: diz porque o método deve ser utilizado ou não deve;
c) Mecânica: é uma enumeração dos passos que devem ser seguidos a fim de utilizar com sucesso a refatoração em questão.
d) Exemplo: será feita uma breve descrição de quando esse método deve ser usado e uma breve explicação do método em questão;

4.2 EXTRAIR MÉTODO

Essa técnica de refatoração é usada quando tem-se um fragmento de código que pode ser agrupado, transformando esse em um método (FOWLER,2004).

4.2.1 Motivação

De acordo com Fowler (2004) quando têm-se um método muito longo ou um código que precise de um comentário para ter seu propósito compreendido. Segundo este autor deve-se transformar esse fragmento de código em um método, e de preferência que seja um método curto e com nome apropriado já que isso favorece a legibilidade do código.

4.2.2 Mecânica

A seguir será apresentado os passos para executar a técnica de refatoração *Extrair método.*

a) Um novo método deve ser criado, lembrando que o nome desse deve ser condizente com o que ele faz.
b) Copiar o código extraído para o novo método.
c) Procurar no código extraído referências a variáveis locais no escopo do método de origem. Essas variáveis locais podem ser parâmetros do método de origem, além das próprias variáveis locais desse método.
d) Verificar a existência de variáveis locais apenas dentro do trecho extraído. Caso positivo, as declare como variáveis temporárias dentro do novo método.
e) Verificar se algumas das variáveis locais são modificas no trecho de código extraído. Caso positivo, verifique se o código extraído pode ser tratado como uma consulta e atribua o valor da consulta à variável local em questão.
f) Passar como parâmetro as variáveis de escopo locais que sejam lidas pelo código extraído.
g) Substituir o código extraído no método de origem por uma chamada do novo método;
h) Compilar e testar.

4.2.3 Exemplo

O código não refatorado na Figura 5 apresenta um bloco de código que precisa de um comentário para explicar o seu propósito conforme apresentado nas linhas 2 e 6. Também foi identificado a necessidade de transformar fragmentos de códigos(da linha 3 à 5) em pequenos métodos para facilitar a legibilidade e entendimento do código.

A Figura 6 apresenta o código refatorado, aplicando a técnica de refatoração *Extrair Método* para resolver o problema apresentado na Figura 4. Para aplicar esta técnica (*extrair o método)* basta remover os comentários que consta no código como pode ser visto na Figura 4 nas linhas 2 e 6 e transformando em um método curto como pode ser visto nas linhas de 4 à 6 e 7 à 10. Vale ressaltar que o comportamento do *Software* é o mesmo após aplicação da técnica.

```
1. void imprimirDivida(double quantia){
2. //imprimir cabeçalho
3. System.out.println("**************************");
4. System.out.println("***** O Cliente Deve *****");
5. System.out.println("**************************");
6. //imprimir detalhes
7. System.out.println("nome: " + this.nome);
8. System.out.println("quantia: " + divida);}
```

Figura 5 – código não refatorado precisa de um comentário para explicar seu propósito
Fonte: Fowler (2004).

A Figura 6 têm-se a solução para o problema apresentado na figura anterior(Figura 4). Após aplicação da técnica obteve-se uma melhoria na legibilidade e a compreensão do código.

```
1. void imprimirDivida(double quantia){
2. imprimeCabeçalho();
3. imprimeDetalhes(quantia);}
4. private void imprimeDetalhes(double quantia){
5.     System.out.println("nome: " + this.nome);
6.     System.out.println("quantia: " + divida);}
7. private void imprimirCabecalho(){
8.      System.out.println("**************************");
9.      System.out.println("***** O Cliente Deve *****");
10.     System.out.println("**************************");}
```

Figura 6 – Aplicação da técnica *Extrair Método*
Fonte: Fowler (2004).

4.3 EXPRESSÃO CONDICIONAL

Esse procedimento de refatoração é utilizando quando há uma grande seqüência de testes condicionais sendo que esses retornam um mesmo resultado (BECK, 2000).

4.3.1 Motivação

Em alguns blocos de códigos é possível encontrar expressões condicionais remetendo a um mesmo resultado. Para Beck (2000) condensar as expressões em apenas uma pode ser considerado uma técnica de refatorar.

Para aquele autor condensar as expressões condicionais é importante pelo seguintes motivos:

a) Primeiro: com apenas uma expressão condicional, torna a verificação mais clara mostrando o que está realmente executando uma única verificação. Ao contrário quando se tem condições separadas onde transmite a impressão de ter verificações independentes.
b) Segundo: com as expressões condensadas fica mais fácil de aplicar a técnica *Extrair Método* a partir que se tem código mais claro.

4.3.2 Mecânica

A técnica de refatoração a seguir foram definidos por Fowler (2004) para uma situação semelhante a Figura 6.

a) verificar se nenhuma das expressões condicionais causa efeito colateral, ou seja, alteração em uma das mudanças faz o programa não funcionar como antes. Caso haja esse procedimento não deve ser executado;
b) substituir um conjunto de condicionais por apenas uma expressão condicional usando operadores lógicos;
c) compilar e testar.

4.3.3 Exemplo

O código da Figura 7 apresenta expressões condicionais remetendo o mesmo resultado, conforme é apresentado nas linhas 2, 3 e 4. O que dificulta a leitura do código, onde há mais linhas para serem compreendidas. A Figura 7 apresenta um código não refatorado.

```
1. double valorPorIncapacidade(){//calcular o valor incapacidade
2.      if(_antiguidade <2)     return 0;
3.              if(_mesesIncapacitado > 12)return 0;
4.              if(_eTempoParcial)return 0;}}
```

Figura 7 - Expressões condicionais que retornam o mesmo resultado.
Fonte: Beck (2000).

Código refatorado da Figura 8 consolidou as expressões condicionais utilizando o operador “OU” para obter único resultado, como pode ser visto nas linhas 2 e 3.

```
1. double valorPorIncapacidade(){//calcular o valor por incapacidade
2. if((_antiguidade < 2)||(_mesesIncapacitado > 12)|| (_eTempoParcil))
3.              return 0;
4. }
```

Figura 8 - Consolidar Expressão Condicional.
Fonte: Beck (2000).

4.4 ENCAPSULAR CAMPO

Segundo Fowler (2004), essa refatoração é utilizada quando há um campo público sendo acessado de maneira indevida, por outras classes.

4.4.1 Motivação

Quando os dados não estão encapsulados, ou seja, outros objetos podem acessar e modifica-los, isso dificulta no processo de manutenção do programa dificultando alterações a nível de código.

Para aquele autor *encapsular campo* envolve a alteração do modificador de acesso do campo e a criação de métodos de acesso a esse.

4.4.2 Mecânica

Para aplicar essa refatoração deve-se seguir os passos a seguir:

a) Criar métodos de gravação e leitura para o dado.
b) Encontrar as classes que utilizam esse dado. Nas classes que utilizam o valor altere a referência pelo método de leitura. Nas classes que modificam o valor altere a referência pelo método de escrita.
c) Compilar e testar após cada modificação.
d) Alterar o modificador de acesso para privado.
e) Compilar e testar;

4.4.3 Exemplo

Código da sem refatorar apresenta um campo na linha 2 da Figura 9, com o modificador *public*, possibilitando que objetos de outras classes possam alterar e acessar seus valores diretamente, levando ao acoplamento alto do programa.

```
1. public class Funcionario{
2. public String nome;}
```

Figura 9 - Encapsular Campo (sem refatorar)
Fonte: Fowler (2004).

Código refatorado na Figura 10 está substituindo o modificador de acesso do campo *nome* e criando os métodos de leitura e escrita para estes.

A partir de agora as classes externas precisam chamar os novos métodos da linha 3 e 5 da Figura 10 para ter acesso ao campo *nome* (linha 2).

```
1. public class Funcionario{
2. private String nome;
3. public String getNome( ){
4.  return nome;}
5. public void setNome(String nome){
6.  this.nome = nome;}}
```

Figura 10 - Encapsular Campo (refatorado)
Fonte: Fowler (2004).

4.5 MOVER CAMPO

Esse procedimento de refatoração deve ser utilizando quando um campo for mais usado por outra classe do que pela classe que o define (FOWLER, 2004).

4.5.1 Motivação

A principal motivação dessa técnica de refatoração é o maior uso de um campo por outras classes, sendo que esse uso pode ser direto ou indireto através de métodos de acesso.

4.5.2 Mecânica

Para Fowler(2004) essa técnica deve ser aplicada com os passos a seguir:

a) mover o campo para a classe de destino e cria-se os métodos de acesso
b) compilar a classe de destino
c) determinar como referenciar a classe de destino a partir da classe de origem
d) substituir as referencias ao campo na classe de origem por referencias aos métodos de acesso criados na classe de destino;
e) compilar e testar.

4.5.3 Exemplo

Código antes da refatoração (Figura 11), apresenta um campo (*taxaJuros*) criado na linha 3 da Figura 11, que está sendo utilizado com menor freqüência em relação a classe *TipoConta* da Figura 12.

```
1. class Conta. . .
2.      private TipoConta tipo;
3.      private double taxaJuros;
4.
5. double jurosPorQuantia_dias(double quantia, int dias){
6.      return taxaJuros * quantia * dias / 365;
7. }
```

Figura 11- Mover Campo(antes da refatoração)
Fonte: Fowler(2004).

Código refatorado da Figura 12 está movendo o campo *taxaJuros* para a classe *TipoConta*. Está classe é onde há maior freqüência do uso do campo *taxaJuros* como pode ser visto nas linhas 4 e 7 do código fonte.

```
1. class TipoConta...
2.      private double taxaJuros;
3. void gravarTaxaJuros(double arg) {
4.      taxaJuros = arg;}
5. double lerTaxaJuros( )
6. {
7.  return taxaJuros;}
```

Figura 12 - Movendo campo (código Refatorado)
Fonte: Fowler (2004).

Na Figura 13 foi removido o campo (*taxaJuros*) utilizado com menor freqüência nessa classe. Porém foi criada uma chamada ao método para obter informações desse campo conforme pode ser observado na linha 4.

```
1. class Conta...
2. private TipoConta tipo;
3. double jurosPorQuantia_dias(double quantia, int dias ){
4.      return tipo.lerTaxaJuros() * quantia *dias / 365;}
```

Figura 13 - Removendo o campo usado com menor freqüência
Fonte: Fowler (2004).

4.6 SUBSTITUIR EXCEÇÃO POR TESTE

Essa técnica de refatoração ocorre geralmente quando os testes condicionais são substituídos por exceções. Dessa forma está gerando uma exceção em uma condição que o solicitante poderia ter verificado primeiro (FOWLER, 2004).

4.6.1 Motivação

Segundo Fowler (2004), as exceções são um avanço importante nas linguagens de programação, elas nos permitem evitar códigos complexos. Por outro lado, as exceções geradas em demasia podem deixar de ser prazerosas. Exceções devem ser usadas para o comportamento excepcional, ou seja, um comportamento que é um erro inesperado. Elas não devem agir como um substituto para testes condicionais.

4.6.2 Mecânica

Os passos a seguir apresenta como a técnica de refatoração vista nesse tópico pode ser executada.

a) Fazer um teste inicial e copiar o código do bloco *catch* para o corpo da estrutura do *if* inserido;
b) certifica-se que o bloco *catch* não é mais executado, acrescentando uma asserção (*assertion*) nesse bloco para notificá-lo caso esse for executado;
c) compilar e testar;
d) remover o bloco *catch* e o bloco *try;*
e) compilar e testar.

4.6.3 Exemplo

O código não refatorado na Figura 14 apresenta uma situação onde o teste condicional foi substituído por exceções. Conforme pode ser visto nas linhas 2 à 5. Na linha 3 retorna um valor do *Array* na posição especificada *numeroDoPeriodo.* Caso seja especificado um valor que não exista no *Array* uma exceção irá ocorrer.

```
1. double getValoresDoPeriodo(int numeroDoPeriodo){
2. try {
3.      return _valores[numeroDoPeriodo];
4. }catch(ArrayIndexOutOfBoundsException e){
5.      return 0;}}
```

Figura 14 - Substituindo exceção por teste condicional
Fonte: Fowler (2004).

O código refatorado na Figura 15 faz a substituição das exceções por um teste condicional. Como pode ser visto na linha 2 onde o comando *if* testa o *numeroDoPeriodo* com o tamanho do *Array*. Se for maior retornará o valor 0, mas se for menor retorna um valor na posição *numeroDoPeriodo*. Porém se o valor for negativo, retornará o valor 0.

```
1. double getValoresDoPeriodo(int numeroDoPeriodo){
2. if(numeroDoPeriodo >= _valores.length) return 0;
3. if (numeroDoPeriodo < 0 ) return 0;
4. return _valores[numeroDoPeriodo];}
```

Figura 15- Inserindo um condicional *if* e removendo as exceções
Fonte: Fowler (2004).

4.7 MOVER MÉTODO

Algumas vezes, temos um método sendo utilizado por mais recursos de outras classes do que pela classe onde foi definido. Em situações como essa deve-se mover o método a fim de adequar seu uso a classe (FOWLER, 2004).

4.7.1 Motivação

Mover métodos é uma prática comum da refatoração. Normalmente move-se métodos quando as classes têm comportamento demais ou estão colaborando entre si. Movendo alguns métodos ficam mais claros os objetivos e as responsabilidades de cada classe. Um ponto importante é que os métodos da classe devem ser examinados antes de serem mudados. Deve-se procurar métodos que pareçam referenciar outros objetos mais do que a classe na qual está inserido. Após encontrar um método candidato, é necessário avaliar os métodos que ele chama e os métodos que o chama a fim de definir com qual objeto o método interage mais.

4.7.2 Mecânica

A refatoração *mover método* foi executada com os passos a seguir:

a) examinar se os atributos da classe de origem são utilizados pelo método e se esses devem ser movidos também;
b) procurar declarações dos métodos nas subclasses e superclasses da classe de origem;
c) mover o método para a classe destino;
d) ajustar o método a fim de funcionar na nova classe;
e) compilar e testar;
f) caso o método seja removido da classe de origem, as referencias ao método devem ser substituídas por referências de sua novas localidade;
g) compilar e testar.

4.7.3 Exemplo

O código da Figura 16 apresenta um código não refatorado onde teremos diversos novos tipos de contas, cada uma das quais com suas próprias regras para calcular o custo do saque a descoberto. Observe que na linha 3 o método *cobrançaPorSaqueADescoberto* está usando mais recursos de outra classe onde não foi definido.

```
1. class Conta . . .
2.      private TipoConta tipo ;
3. double cobrancaPorSaqueADescoberto(){
4. if(tipo.ePremium()){
5.      double resultado = 10;
6. if( diasDescorbetos > 7)
7.      resultado += (diasDescorbetos - 7) *0.85;
8.      return resultado;
9. }else return diasDescorbetos * 1.75;}
10.double tarifaBancaria(){
11.     double resultado =4.5;
12.if(diasDecorbetos > 0)
13.     resultado += cobrancaPorSaqueADescoberto();
14.return resultado}
```

Figura 16 - Método que tem recursos demais sobre outra classe
Fonte: Fowler (2004).

A Figura 17 apresenta o código refatorado onde o método que foi transferido (*cobrancaPorSaqueADescoberto*) para a class *TipoConta* como pode ser visto na linha 2 à 13

a referência *tipo* foi removida na linha 4. Agora a *class Conta* retorna o método *cobrancaPorSaqueADescoberto* na linha 17. Veja a seguir como ficaram as classes:

```
1. class TipoConta. . .
2. double cobrancaPorSaqueADescoberto(int diasDescobertos){
3. if(ePremium()){
4.      double resultado = 10;
5. if( diasDescorbetos > 7)
6.      resultado += (diasDescorbetos - 7) *0.85;
7.      return resultado;}
8. else return diasDescorbetos * 1.75;}
9. double tarifaBancaria(){
10.     double resultado =4.5;
11.if(diasDecorbetos > 0)
12.     resultado += cobrancaPorSaqueADescoberto();
13.return resultado;}
14.
15.class Conta{
16.double cobrancaPorSaqueADescoberto(){
17.return tipo.cobrancaPorSaqueADescoberto(diasDescorbetos);}}
```

Figura 17 - Movendo método para a classe onde está sendo mais utilizado
Fonte: Fowler (2004).

4.8 SUBSTITUIR NÚMERO LITERAL POR CONSTANTES SIMBÓLICAS

O entendimento de número literais pode ser claro para as pessoas que o codificam, porém seriam mais claros se fosse substituído por constantes simbólicas com um nome intuitivo e auto-explicativo. Esses números simplesmente largados no código dificultam muito a leitura e entendimento desse, dificultando no processo de manutenção e diminuindo a velocidade no desenvolvimento do *Software* (FOWLER,2004).

4.8.1 Motivação

Números literais são um dos males mais antigos da computação. São números com valores especiais no sistema e que normalmente não são óbvios. É muito oneroso quando se quer utilizar um mesmo número lógico em mais de um lugar. Se houver a possibilidade dos números mudarem tem-se um trabalho enorme em mudar todas as ocorrências, levando a dificuldade no processo de manutenção do *Software*.

4.8.2 Mecânica

Para execução dessa refatoração basta seguir os passos a seguir:

a) declarar uma constante e atribua a ela o valor do número literal;

b) encontrar aonde o número aparece;

c) veja se a mudança pode ser feita, caso positivo altere o número pela constante;

d) compilar;

e) assim que os números literal forem substituídos, testar.

4.8.3 Exemplo

Na Figura 18 temos um código não refatorado que apresenta um número literal 9.81. Como pode ser visto na linha 2.

```
1. double energiaPotencial(double massa, double altura){
2.  return massa * 9.81 * altura;
3. }
```

Figura 18 - A presença de número literal no sistema
Fonte: Fowler (2004).

Na Figura 19 com o código refatorado foi criada uma constante para representar o número literal e a nomeação auto-explicativa, conforme pode ser visto na linha 1 e 3. Agora o processo de manutenção se torna mais fácil e conseqüentemente mais rápido.

```
1. static final double CONSTANTE_GRAVITACIONAL = 9.81;
2. double energiaPotencial(double massa, double altura){
3.  return massa * CONSTANTE_GRAVITACIONAL * altura;
4. }
```

Figura 19 - Substituindo número literal por uma constante simbólica
Fonte: Fowler (2004).

CAPITULO 5 - APRESENTAÇÃO ESTUDO DE CASO

Este capítulo tem como objetivo apresentar estudos de caso realizados com a técnica de refatoração em sistemas orientados a objetos, apresentando os resultados obtidos por alguns autores após aplicação da técnica de refatoração e sua devida contribuição para o sistema.

5.1 ESTUDO CASO 1 REFATORAÇÃO NO DIREITOLIVRE

O estudo de caso "*Refactoring* no direitolivre aplicado padrões de projetos" desenvolvido por Ramos (2006), tem como desafio planejar e melhorar um sistema já existente denominado *direitolivre*, construído pela Companhia de Processamento de Dados do Rio Grande do Sul (PROCERGS). O objetivo segundo este autor é trazer ganhos e melhorias para arquitetura. O enfoque neste estudo de caso foram padrões de projetos voltados a tecnologia *JavaEE* em conjunto com técnicas de refatoração.

Segundo Ramos (2006), foram encontrados os seguintes problemas na arquitetura do sistema: código duplicado, teste condicional com lógica complexa, lista grandes nos parâmetros dos métodos. Com base nos problemas identificados na arquitetura segundo Ramos (2006), foi identificado à necessidade de utilizar a técnica de refatoração como resolução.

Para Ramos (2006), na nova arquitetura mostrou que as técnicas de refatoração não são simplesmente modelos para manutenção de códigos. Foi perceptível a contribuição da técnica para incorporação de benefícios provenientes da evolução da área de engenharia de *Software*. Sendo que para aquele autor a técnica é totalmente cabível a evolução de projetos sem perda financeira e intelectual de uma nova criação, simplesmente delegando tempo para que seja adaptada a realidade em questão à tecnologia atual. Sendo que com aplicação da técnica o comportamento do sistema manteve-se inalterado, onde qualquer usuário que não participou do processo de refatoração será capaz de dizer se algo mudou.

5.2 ESTUDO DE CASO 2 REFATORANDO O SIMGRIP

"Refatorando o SimGrIP: Um estudo de caso acerca da aplicação de técnicas de refatoração de *Software*" desenvolvido por Lima et. al (2007) teve como objetivo demonstrar

a eficiência da refatoração de código em sistemas construídos com o paradigma de orientação a objetos. Para este estudo foi utilizado a aplicação Simulador Gráfico de Roteamento IP (SimGrIP) como forma de melhorar a qualidade do *Software*.

Segundo Lima, et. al (2007), o sistema SimGrIP foi desenvolvido no centro federal de educação tecnologica da Paraíba, a fim de apoiar o ensino do funcionamento de redes de computadores baseados no protocolo IP (*internet protocol*).As recentes necessidades de expansão encontraram impasse e dificuldade de serem realizadas, pois o forte acomplamento entre algumas classes e trechos de códigos com díficil compreensão tornando a manutenção custosa.

Na análise feita por Lima, *et. al* (2007) no código do *Software* notou-se alguns pontos que necessitavam de refatoração. Dentre estes é possivel citar:

a) a classe *Application*, que sozinha montava, controlava e tratava os elementos de interface, isso a tornava uma estrutura monolítica;
b) Nomes de classes não condizentes com seus papéis;
c) Uma classe interna à classe *Application* para manipular arquvios XML[7];
d) existencia de códigos duplicados.

Os pontos citados acima pelos autores tornavam o SimGrIP um *Software* dificil de ler e a inserção de novas funcionalidades uma tarefa muito custosa.

A Tabela 9 apresenta as técnicas de refatoração aplicadas no sistema SimGrIP.

Técnica de Refatoração	Utilização
Extract Class	Técnica que explana os procedimentos necessários para a extração de uma classe de dentro de outra. Utilizada no SimGrIP para extrair as classes internas e fábricas.
Extract Method	Técnica que descreve como extrair métodos menores e concisos de um método extenso que faz várias tarefas. Foi usada para dividir o método main(String args[]) da antiga classe Application em métodos menores, que foram inseridos nas fábricas, responsáveis por criar componentes específicos.
Move Method	Técnica que explica como mover métodos de uma classe para outra.
Inline Temp	Técnica usada para diminuir a criação de variáveis temporárias. Foi utilizada nas fábricas de painéis e menus, nas chamadas dos métodos de adição de componentes.

Tabela 9 Descrição de técnicas de refatoração e seus usos no SimGrIP.

Fonte: Lima, *et. al* (2007)

[7] XML É uma Linguagem Padronizada de Marcação Genérica capaz de descrever diversos tipos de dados. Seu propósito principal é a facilidade de compartilhamento de informações através da Internet (W3C, 2008)

Segundo Lima, *et. al* (2007), existem outras classes que não estão bem projetadas e devem ser alvos de futuras refatorações, porém, o ponto de partida foi a classe *Application*. Para este autores, dado as necessidades de expansão, manutenção e legibilidade do código a refatoração aplicada ao *Software* SimGrIP possibilitou uma melhor adequação das funcionalidades modificadas aos seus requisitos funcionais beneficiando desenvolvedores e clientes. Para aqueles autores a aplicação dos conceitos e técnicas de refatoração, obteve uma funcionalidade do *Software* mais adequável e manutenível. Dessa forma, foi possivel ter elementos incluidos com maior facilidade, tornando-se um *Software* estendivel, atualizavel.

CAPITULO 6 - CONCLUSÃO

Do que foi apresentado, conclui-se que:

a) a técnica de refatoração pode ser aplicada no desenvolvimento e manutenção do *Software,* uma vez que, como demonstrado ao longo do texto, pode auxiliar os engenheiros e programadores no processo de desenvolvimento e manutenção preventiva de *Software;*

b) refatoração possibilita o acréscimo de nova funcionalidade de maneira fácil evitando a desestruturação do código, já que propicia a legibilidade a nível de código que evita desestruturação do código;

c) com o uso da refatoração se gasta menor tempo no processo de manutenção em relação ao código anterior, já que com a técnica, melhora a legibilidade e estrutura o código, uma vez que remove código duplicados, comentários desnecessários, códigos condicionais difíceis de ler, etc.

Através de uma análise quantitativa de estudos de casos da literatura existente, foi possível perceber os efeitos da técnica de refatoração e sua influência positiva dentro do processo de manutenção de sistemas, preservando o comportamento e facilitando a inserção de novas funcionalidades, possibilitando um acréscimo rápido e fácil sem desestruturar o código.

Para aplicação da técnica de refatoração o engenheiro ou programador pode utilizar ferramentas que dão suporte a linguagem utilizada para o desenvolvimento do *Software*. Essa ainda é uma área que está em fase de crescimento, onde estudantes e profissionais vêm desenvolvendo e aprimorando as técnicas e criando novas ferramentas para auxilio no processo de refatoração.

Uma área que está em fase de crescimento é a aplicação de refatoração em sistemas de banco de dados, a fim de obter uma manutenção menos custosa desde o processo de gerenciamento até a implantação desses bancos. Sendo assim, esta é uma área que propicia novos estudos e criação de novas ferramentas que implementem as técnicas de refatoração nesses sistemas, ficando assim, uma sugestão para futuros estudos.

Uma área futura para refatoração é aplicar esta técnica em sistemas de Banco de Dados a fim de obter uma manutenção menos custosa, sendo assim, sugestão para estudos e desenvolvimento é a na criação de novas ferramentas que apliquem as técnicas de refatoração em sistemas de Banco de Dados.

REFERÊNCIAS

BECK, K. Kent. **Extreme Programming Explaned**. Embrace Change.2000.

BORLAND. disponível em <http://www.borland.com>.Acesso em 4 de Junho de 2008.

DEITEL Harvey,**Java como Programar** 6.ed. São Paulo SP – Pearson, 2006.

FOWLER, M. **Refatoração Aperfeiçando o Projeto de Código Existente** 1. ed. Porto Alegre - RS: Bookman,2004.

GAMMA, H. E., JOHNSON, R. **Design Patterns:Elements of Reusable Object Oriented Software.** Porto Alegre - RS: Bookman,1995.

GEEK. **GEEK BRASIL,** Disponivel em: <http://www.geekbrasil.com.br>.Acesso em 10 de Setembro de 2008

JFACTOR. disponível em <http://old.instantiations.com/jfactor/default.htm>.Acesso em 16 de setembro de 2008.

JUNIT **Testing framework,** disponível em JUnit: <www.junit.org>.Acesso 13 de Maio de 2008

LIMA, C., Guedes, et al. **O SIMGRIP: Um estudo de caso acerca da aplicação de técnicas de refatoração de Software** . João Pessoa, PB: Centro Federal de Educação Tecnológica da Paraíba, 2007.

MAIA, P. H. **REFAX:Um arcabouço para desenvolvimento de ferramentas de refatoração baseado XML.** Monografia (Programa de Pós Graduação em Ciência da Computação) Universidade FederalCeará, Fortaleza 2004.

NUNES, E. **JRefactor um componente para Refactoring em código-fonte Java.** Monografia: Universidade Luterana do Brasil, Gravatai, 2003.

OMG.**The Object Management Group** disponível em Object Management Group: <http://www.omg.com> s.Acesso em 7 de Setembro de 2008.

OPDYKE, W. F. **Refactoring Object-Oriented Frameworks**. Dissertação Ph.D. Univeristy of Illinois, Urbana Champaign,1992.

PÁDUA, W. d. **A**lguns Fundamentos da Engenharia de Software**.** Engenharia de Software **Magazine** (1), 4-9. 2007.

PRESSMAN, R. S. **Engenharia de Software** 3ª ed. São Paulo: McGraw-Hill. 2006.

RAMOS, L. **Refactoring aplicando padrões de projetos JavaEE** . Monografia. Universidade Luterana do Brasil, Gravatai, 2006.

REFACTORIT. disponível em <http://www.refactorit.com>. Acesso em 16 de setembro de 2008.

ROBERTS, D. **Refatoração. In: M. Fowler, Aperfeiçoando o Projeto de Código Existente** Bookman, 2004.

SCHMIDT, D. **Object-Oriented Application FrameWorks**. New York: Communications of the ACM. 1997.

SOMMER-VILLE, L. **Engenharia de Software.** 1.ed São Paulo: Pearson Addison Wesley,2007.

W3C. disponível em <http://www.w3.org/XML/ >.Acesso 21 de novembro de 2008.

XREF-TECH.XRefactory disponível em <http://www.xref-tech.com/> Acesso em 04 de Junho de 2008