Refatoração em um Sistema Legado

Julio Cesar do Carmo Heredia[1]
Hugo Bastos de Paula[2]

Resumo

Este trabalho aborda a refatoração de código legado em um sistema de venda de passagens escrito em Java. Esta abordagem se apoia na dificuldade dos desenvolvedores ao encontrar um código em que a sua estrutura original esteja com a sua integridade comprometida e com tecnologias obsoletas. São apresentadas neste trabalho, técnicas de refatoração que levaram este código a ter principalmente mais inteligibilidade, para que seja um sistema de fácil compreensão para seus desenvolvedores. O trabalho também tem o intuito de quantificar a melhoria do código refatorado. O código será metrificado com o auxílio das ferramentas SonarQube e Metrics. Essas ferramentas farão o comparativo do antes e pós refatorações, com o propósito de relatar a melhoria na qualidade de software após o término das refatorações. Foram avaliados dois projetos, o projeto Cliente e o Servidor. O projeto Cliente teve uma melhor margem na melhoria da qualidade de software em relação ao projeto Servidor, onde ouve uma melhoria significativa nas métricas relacionadas à complexidade do código, de 14,44% na análise do SonarQube e de 44,26% na complexidade ciclomática no relatório do Metrics.

Palavras-chave: Refatoração, Técnicas de Refatoração, Sistemas Legados, Qualidade de Software e Métricas de Software.

[1]Aluno, Trabalho de Conclusão de Curso em Sitemas de Informação da PUC Minas, Brasil– jul.heredia@gmail.com
[2]Orientador, Instituto de Ciências Exatas e de Informática da PUC Minas, Brasil– hugo@pucminas.br

1 CONTEXTUALIZAÇÃO

Sistemas legados são sistemas críticos, em uso há um longo período, que permanecem de forma vital para uma organização, tendo risco significativo a sua desativação. Com o passar do tempo, estes sistemas têm sua estrutura original modificada e a sua integridade enfraquecida, tornando-se um sistema de difícil manutenibilidade e extensão (ARBOLEDA et al., 2013; FOWLER, 2004; GRIFFITH et al., 2011).

Cerca de 70% do custo do desenvolvimento de software, é usado na manutenção e os 30% restantes são usados em novas funcionalidades. Uma das maneiras de melhorar este fator de custo/produtividade na manutenção e extensão de códigos legados é a refatoração. Com a refatoração, a produtividade na manutenção de software pode ser maior, uma vez que a estrutura interna de software pode ser alterada mais facilmente, para que seja mais compreensível e modificável (BOEHM, 1982; FOWLER, 2004).

Outra questão abordada, é que os donos do projeto devem dar mais tempo e dinheiro para melhorar a qualidade do software. Algum tempo investido melhorando uma parte do código pode economizar horas do tempo de cada desenvolvedor para realizar correções, modificações ou para acrescentar alguma regra de negócio ao sistema (FOWLER, 2004).

As principais dificuldades para os desenvolvedores de sistema legado são: o entendimento do fluxo do sistema; como as suas funcionalidades e módulos se integram; a baixa confiança para dar manutenções e implementar novas funcionalidades; extensas baterias de testes; impactos negativos de suas alterações em outras funcionalidades (alterando pontos distintos do sistema pode se alterar várias funcionalidades desconexas); operações em vários pontos do sistema que se repetem e o baixo desempenho. Pelo fato de que o sistema continuará sendo utilizado e desenvolvido por um longo período, é necessário que ele se adapte a novas tecnologias e que sua base seja melhorada, ou seja, ele precisa ser refatorado (FOWLER, 2004; SCHÄFER et al., 2010).

O objetivo deste trabalho é mostrar como a refatoração pode interferir diretamente na qualidade do software, ou seja, melhorar a inteligibilidade do código a ponto que se torne um sistema de fácil compreensão para seus desenvolvedores, mostrando através de ferramentas e técnicas a melhoria na qualidade do código. As mudanças feitas através da refatoração podem levar este sistema a ter uma vida útil muito maior, além de diminuir o custo de correções e reparos no sistema (FOWLER, 2004; PEREZ; CRESPO, 2007).

2 PRINCÍPIOS DA REFATORAÇÃO

> Refatoração é uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável (FOWLER, 2004).

A refatoração se baseia nos princípios básicos de mudar o programa em passos peque-

nos, o bom código deve ser compreendido por pessoas e não só por computadores. Se o código tem mais de três duplicidades e precisa de um comentário para explicar o que foi feito é melhor refatorar. Ela é uma ferramenta valiosa que ajuda a manter os códigos mais seguros, de modo que pode ser usada para diversos propósitos. A primeira coisa a ser abordada sobre a refatoração é o porquê desta ferramenta ser utilizada e as vantagens reais que ela traz ao desenvolvimento e ao produto final de software. Ou seja, refatorar melhora o projeto de software (FOWLER, 2004).

Normalmente, durante o desenvolvimento, o código é feito para funcionar, e nem sempre é escrito dentro dos padrões e das melhores práticas. Com o passar do tempo, a alteração do código fará com que o projeto do programa sofra deterioramento. Com a refatoração, pode-se arrumar o código pois, a refatoração sistemática o ajuda a conservar a sua forma. Quando a refatoração do código é feita corretamente, o projeto fica melhor, mais legível e menos propenso a falhas. Em um código bem projetado e estruturado, o desenvolvimento é mais rápido e seguro. A refatoração pode ser feita em alguns momentos: quando se acrescenta novas funções, ao consertar falhas, ou ao revisar o código (FOWLER, 2004).

No desenvolvimento das refatorações, existem soluções mais sofisticadas, flexíveis e elegantes, que é a utilização de padrões de projetos. Padrões de Projeto é a reutilização de soluções que funcionaram no passado e que ajudam os desenvolvedores a resolverem problemas comuns que ocorrem em sistemas orientados a objetos (GAMMA et al., 1994; KERIEVSKY, 2008).

Projetos de refatoração que utilizam padrões resultam em refatorações mais flexíveis e reutilizáveis. A refatoração por padrões eleva o nível do código, tornado a qualidade deste software maior (KERIEVSKY, 2008).

O desenvolvimento de refatorações por padrões não será apresentado neste trabalho.

3 RETORNO DE INVESTIMENTO (ROI)

Softwares de baixa qualidade geralmente possuem muitos defeitos, isto resulta em um uso menor do sistema e em receita menor do produto. Softwares com alta qualidade têm maior utilização e maiores receitas. Mas se os defeitos do sistema forem descobertos com o software já em produção, geram um aumento nos custos do projeto (EMAM, 2005).

Segundo Emam (2005), o investimento feito na melhoria da qualidade de software é uma decisão tomada pelos gerentes de negócios. O investimento certo a ser destinado para a qualidade de software é o quanto este investimento poderá gerar de lucro.

A refatoração é uma forma de um software se manter competitivo e sustentável com o passar do tempo. No modelo atual de desenvolvimento de software, as empresas devem se preocupar com a qualidade do código que é produzido. Neste contexto, o retorno de investimento na qualidade de software está ligado ao custo em detectar defeitos e corrigi-los em um sistema já legado que é maior do que o custo de refatorá-los. Desta forma, a produtividade e a satisfação dos desenvolvedores tendem sempre a aumentar (FOWLER, 2004; EMAM, 2005).

4 TÉCNICAS DE REFATORAÇÃO

Segundo Fowler (2004), para se aplicar a refatoração em um código, é necessário conhecer primeiro as técnicas de refatoração que serão utilizadas. Deve ser feita uma análise no código para se detectar os pontos que necessitam passar por uma refatoração. A decisão de quando começar a refatorar e quando parar é tão importante quanto a mecânica da refatoração.

Os indícios que são normalmente detectados em trechos do código que precisam de refatoração são chamados de maus cheiros de código. Estes maus cheiros de código estão descritos no Quadro 1 (FOWLER, 2004).

Quadro 1 - Maus Cheiros do Código

Indícios de Refatoração	
Cirurgias com Rifle	Classes de Dados
Grupos de Dados	Comentários
Métodos Longos	Códigos Duplicados
Listas de Parâmetros Longas	Classes Grandes
Comandos Switch	Alterações Divergentes
Classes Ociosas	Inveja de Dados
Campos Temporários	Obsessão Primitivas
Objetos, interfaces e classes intermediárias	Hierarquias Paralelas de Heranças
Generalidades Especulativas	Classes Alternativas com Interfaces Diferentes
Bibliotecas de Classes Incompletas	Cadeias de Mensagens
Heranças Recusadas	Intimidade Inadequada

Fonte: (FOWLER, 2004)

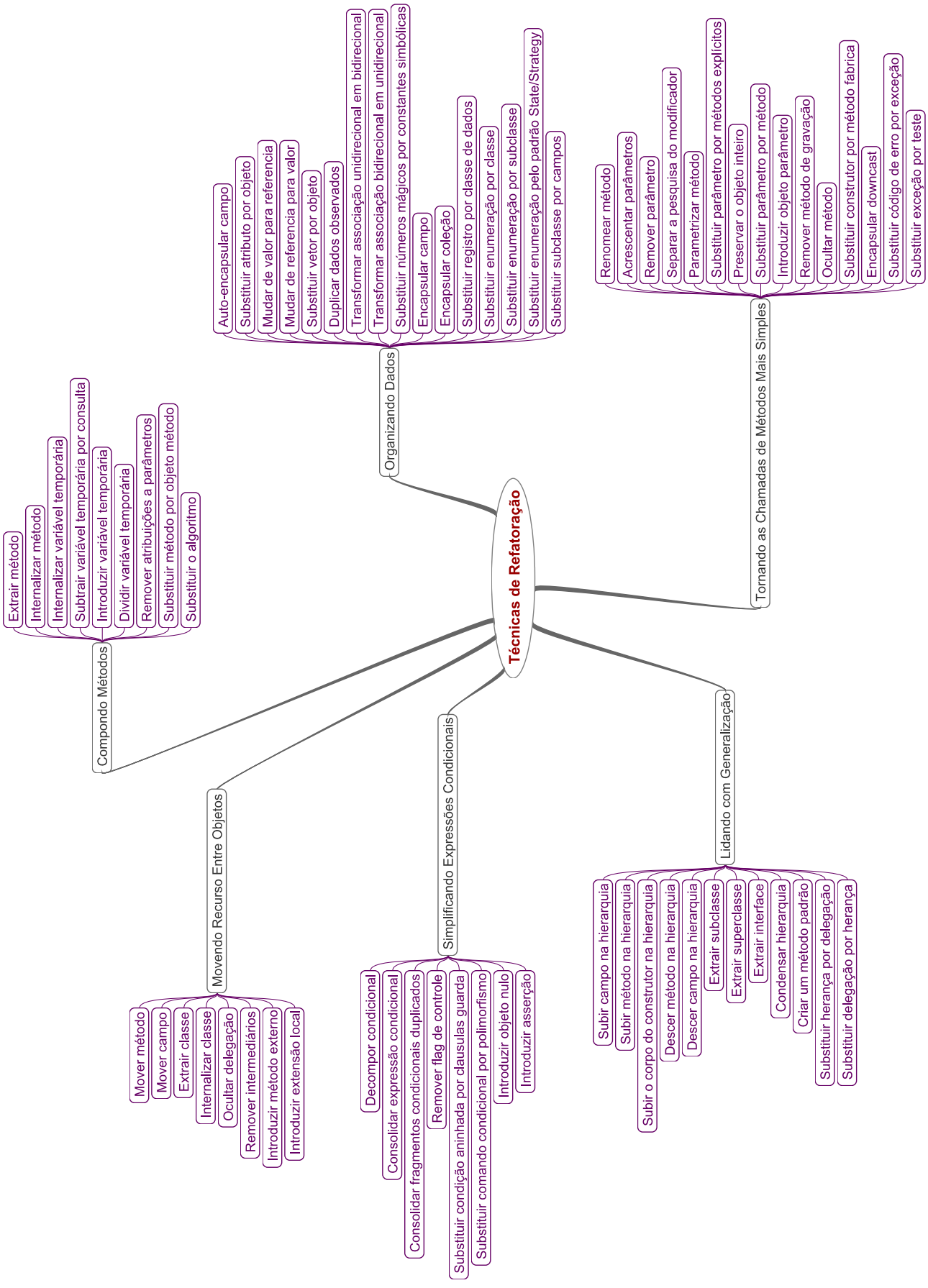
Segundo Fowler (2004), as técnicas de refatoração são diferentes padrões que se dividem em seis categorias. Serão apresentadas abaixo as descrições das técnicas de refatoração que serão utilizadas neste trabalho.

A Figura 1 sintetiza todas as técnicas de refatoração descritas por Fowler (2004), agrupadas em suas respectivas categorias.

4.1 Compondo Métodos

Esta categoria agrupa técnicas que são utilizadas para empacotar códigos reduzindo o acoplamento e a complexidade do método que é constituído.

Figura 1 – Catálogo Completo de Técnicas de Refatoração



Fonte: (FOWLER, 2004)

4.1.1 Extrair método

Extrair método constitui-se em dividir um método grande em pequenos métodos que terão nomes apropriados, tornando-os um código "autoexplicativo", que terá maior legibilidade para seus desenvolvedores.

4.1.2 Internalizar Método

Quando o grupo de métodos está mal organizado, aplica-se a internalização do método, que elimina a indireção desnecessária das chamadas de método.

4.1.3 Substituir Variável Temporária por Consulta

Variáveis temporárias podem motivar métodos longos. Ao trocar variáveis temporárias por consulta, substitui as referências à expressão por um método, deixando o código mais limpo, além de transformá-las em um método que pode ser reutilizável.

4.2 Movendo Recursos Entre Objetos

Quando as responsabilidades não são bem definidas ao longo do desenvolvimento do projeto, é utilizada a refatoração para redefinir estas decisões fundamentais, que podem ser alteradas com as seguintes técnicas.

4.2.1 Mover método

Mover método é usado, normalmente, quando classes têm um excesso de comportamento ou estão colaborando demais entre si, gerando um alto nível de acoplamento. Com esta técnica pode-se simplificar classes, definir melhor a responsabilidade da classe, deixando seus métodos mais claros.

4.2.2 Mover campo

Mover estado e comportamento entre classes é a essência da refatoração. Mover campo é quando se define a importância desse campo e se ele deve ser usado por uma ou inúmeras classes.

4.3 Organizando Dados

Organizando Dados é um grupo de técnicas de fatorações que são extremamente úteis ao tratamento de dados, utilizando linguagem de programação orientada a objetos, que podem permitir a definição de novos tipos que vão além do que pode ser feito com os tipos de dados simples.

4.3.1 Auto-encapsular campo

Com esta técnica, é possível definir o acesso aos campos por apenas métodos que tenham esta responsabilidade.

4.3.2 Substituir vetor por objeto

Basicamente, substitui-se o vetor por um objeto que tenha um atributo para cada elemento do vetor.

4.4 Simplificando Expressões Condicionais

Decompõe as expressões e simplifica as estruturas condicionais organizando e utilizando métodos para cada fluxo apresentado.

4.4.1 Decompor condicional

Lógicas condicionais podem ser escritas de forma bem complexa, e normalmente são encontradas dentro de métodos muito longos. Esta técnica apresenta a intenção de tornar as expressões condicionais mais claras, decompondo e substituindo blocos de códigos por chamadas a métodos cujo o nome representa a intenção deste bloco de código.

4.5 Tornado as Chamadas de Métodos Mais Simples

Este conjunto de refatorações explora a utilização da refatoração para tornar as interfaces mais diretas; produzindo interfaces que sejam mais fáceis de entender e usar, dando uma habilidade chave no desenvolvimento do bom software orientado a objetos.

4.5.1 Renomear método

Altera o nome do método para nomes que comuniquem sua intenção.

4.5.2 Parametrizar método

Quando um método tem mais de uma função, parametrizando pode-se fazer variações na funcionalidade do mesmo.

4.6 Lidando com Generalização

Generalizações produzem seu próprio lote de refatorações, a maior parte delas lidando com a movimentação de métodos por uma hierarquia de herança.

5 VALIDAÇÃO E METRIFICAÇÃO DE QUALIDADE DE CÓDIGO

A refatoração tem inúmeros benefícios, mas é preciso medir as melhorias que a refatoração trouxe ao código. As metrificações destas melhorias influenciam diretamente na qualidade de código do sistema (KRISTIANSEN; STOLZ, 2014).

5.1 Complexidade ciclomática

Complexidade ciclomática pode ser definida como a quantificação do número de caminhos independentes em código fonte de um software, e fornecimento de uma métrica do nível em que está a sua manutenibilidade e testabilidade (MCCABE, 1976).

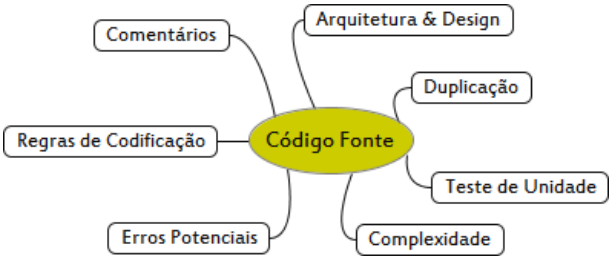
Manutenibilidade é a quantidade de esforço necessária para encontrar e retirar *bugs* em um sistema. Testabilidade é a quantidade de esforço que é necessário para testar um sistema a fim de garantir que a execução da determinada funcionalidade seja realizada com êxito (GOMES, 2008).

Pode-se avaliar todo o sistema ou apenas uma parte do código. A complexidade ciclomática leva em consideração o software (código fonte) como objeto de análise, e não, os requisitos do software (MCCABE, 1976).

5.2 SonarQube

SonarQube é um software *open-source* que verifica a qualidade do código-fonte. Ele controla inúmeras métricas de software e aponta possíveis *bugs*. Os resultados obtidos são gerados a partir de uma interface web em forma de gráficos e *dashboards*. A Figura 2 apresenta as sete dimensões do código fonte que são exploradas pelo SonarQube (SONARQUBE, 2014).

Figura 2 – Sete Pecados Capitais do Desenvolvimento de Software



Fonte: (SONARQUBE, 2014)

As análises de riscos do SonarQube são indicadores de possíveis problemas que existem no software. Os riscos são subdivididos da seguinte forma risco de bloqueio, crítico, grande, pequeno e de informação. A métrica de risco indica a soma destes riscos (SONARQUBE, 2014).

O Risco de Bloqueio pode ser um *bug* com alta probabilidade de impactar o comportamento do sistema, tais como: vazamento de memória e conexão JDBC não fechada. Este código deve ser imediatamente corrigido. Risco crítico pode ser tanto um *bug* com uma baixa probabilidade de impactar o comportamento do sistema ou um problema que representa uma falha de segurança, como: bloco *catch* vazio e injeção de SQL. O código deve ser imediatamente revisto. Risco grande é uma falha de qualidade, que pode impactar altamente a produtividade do desenvolvedor, como: código 'perdido', blocos duplicados e parâmetros não utilizados. Risco pequeno é uma falha na qualidade que pode afetar ligeiramente a produtividade do desenvolvedor, como: linhas não devem ser muito longas e condicionais longas. Risco de informação não é nem um *bug* e nem uma falha de qualidade, é apenas um aviso (SONARQUBE, 2014).

As ferramentas que podem ser usadas para apoiar o trabalho com o SonarQube a garantir a qualidade do código são as seguintes: PMD, Checkstyle e Findbugs.

5.3 PMD

PMD é uma ferramenta que escaneia o código-fonte Java à procura de potenciais problemas, como pode ser visto no Quadro 2 (PMD, 2014).

Quadro 2 - PMD

Escaneamento do PMD	
Possíveis Erros	variáveis vazias, try/catch/finally e instruções switch
Código Morto	variáveis locais não utilizadas, parâmetros e métodos privados
Otimização de Código	por exemplo, utilização de StringBuffer ao invés de String para textos longos
Expressões Complicadas	declarações condicionais desnecessárias, laços "for"que poderiam ser "while"
Código Duplicado	códigos copiados/colados podem significar códigos copiados/colados com erros

Fonte: (PMD, 2014)

5.4 Checkstyle

Checkstyle tem por objetivo analisar código-fonte Java para que o mesmo siga os melhores padrões de estilo de código usados pelo mercado. A ferramenta indica as incidências onde o código analisado não segue estes padrões (CHECKSTYLE, 2014).

5.5 Findbugs

FindBug é um programa para encontrar bugs em programas em Java. Ele busca "padrões de erros", ou seja, instâncias de código que são susceptíveis de serem erros (FINDBUGS, 2014).

6 METODOLOGIA

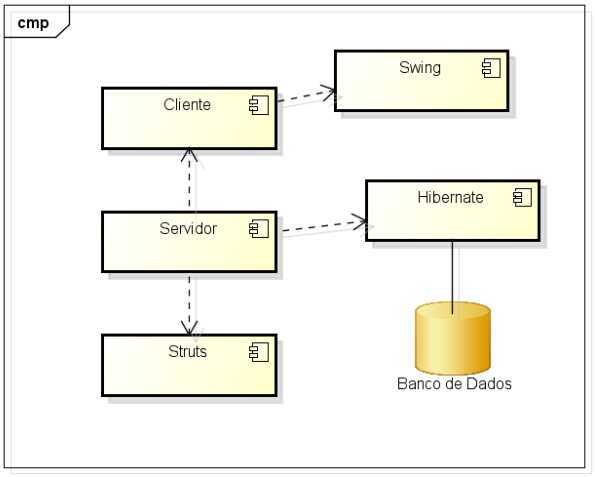
O software avaliado foi desenvolvido na linguagem de programação Java. O desenvolvimento foi feito em ambiente Windows utilizando a IDE Eclipse. O sistema é um software legado de venda de passagem de ônibus, onde os usuários que o utilizam são de nível operacional das empresas de ônibus.

Será feita a refatoração em apenas um módulo do sistema. Este módulo é o de cancelamento de item. As alterações serão feitas em pontos onde contém indícios de necessidade de refatoração.

O módulo utilizado possui o modelo cliente/servidor. Este módulo foi dividido em dois projetos Cliente e Servidor. O projeto Cliente é desenvolvido utilizando o framework *swing* sendo um sistema de plataforma *desktop*. O Servidor utiliza os frameworks *struts* e *hibernate*, este projeto é um sistema web.

Na Figura 3 está representado o um diagrama de componentes do sistema mostrando como os projetos Cliente e Servidor se relacionam com os seus respectivos frameworks.

Figura 3 – Diagrama de Componentes do Sistema



O Servidor é o local estão as regras de negócios do sistema em que se faz a comunicação do sistema com o banco de dados. O projeto Cliente se relaciona diretamente com o Servidor, é enviado requisições (com parâmetros ou não) do projeto Cliente para o Servidor, que após executar estes processos, retorna mensagens no formato xml para o Cliente.

O projeto Cliente possui o total de 60 classes e de 606 métodos, já o projeto Servidor possui 574 classes e 5.484 métodos. Não foi contabilizada, nem no projeto Cliente e nem no Servidor, a quantidade de classes e métodos refatorados.

Swing é uma biblioteca de interface gráfica com o usuário. Um framework para desenvolvimento de aplicações Java para *desktop* (MECENAS, 2008).

Struts é um framework que implementa o modelo MVC (*Model-View-Controller*), e que fornece uma melhor separação entre a camada de apresentação e a camada lógica (HIBERNATE, 2015).

Hibernate é um framework para o mapeamento objeto relacional desenvolvido em Java. Ele transforma as tabelas de um banco de dados em objetos (STRUTS, 2015).

Inicialmente, foram feitas análises de qualidade do código a partir do módulo original, ou seja, antes do processo de refatoração do código dos dois projetos. Ao final de todas as refatorações foi feita a segunda coleta das análises. Desta forma, foi demonstrado a melhoria da qualidade de software após a refatoração.

As ferramentas que foram utilizadas para auxiliar a verificação da melhoria na qualidade software foram o SonarQube e o Metrics no Eclipse. O Metrics foi utilizado para a verificação das métricas de complexidade ciclomática (METRICS, 2014; KRISTIANSEN; STOLZ, 2014).

As técnicas desenvolvidas ao longo trabalho foram: parametrizar método, extrair método, internalizar método, substituir variável temporária por consulta, auto-encapsular campo, mover método, mover campo, renomear método, decompor condicional e substituir vetor por objeto. Todas as técnicas de refatoração foram desenvolvidas seguindo as descrições e exemplificações de Fowler (2004).

7 DESENVOLVIMENTO

Nesta seção será demonstrado como as refatorações foram feitas e como as técnicas de refatoração foram aplicadas.

Inicialmente, as técnicas de refatoração utilizadas foram a **parametrizar método** e **renomear método**. Foi removida a assinatura do método original em que os atributos do *struts* (os parâmetros de assinatura da classe Action do *struts*) e os nomes dos métodos foram alterados para se adequarem as suas funcionalidades. Na refatoração de parametrizar método, a assinatura do método foi substituída por atributos simples (ao invés de buscar os atributos do objeto HttpServletRequest do *struts*), tornando o método independente da tecnologia, além de tornar o método desacoplado e reutilizável. Na refatoração de renomear método, o nome do método, foi alterado para selecionaItensACancelar.

Original:

```
1 public void seleciona(ActionMapping mapping, Form form, HttpServletRequest request,
      HttpServletResponse response) throws Exception {
2     String dataVenda = request.getParameter("dataVenda") == null ? null : (request.
          getParameter("dataVenda").equals("null") == true ? null : request.getParameter(
          "dataVenda").toString());
3     ...
4 }
```

Refatorado:

```
public void selecionaItensACancelar(String dataVenda, String numItem, String
    itemVinculado, String numOperacao) throws Exception {
}
```

Na próxima refatoração, foi utilizada a técnica de **extrair método**, na qual havia uma consulta ao banco de dados dentro do método. Ao refatorar neste ponto do método, a consulta se torna independente a ele, podendo ser reutilizada e dando ao método um nome claro para a sua finalidade.

Original:

```
// Buscar o detalhamento do ItemVendido
String stringQuery = " select iv from ItemVendido iv " +
" where iv.ativo = 1 ";
...
Query query = DB.session().createQuery(stringQuery);
...
List<ItemCancelar> lsItensVendidos = query.list();

if (lsItensVendidos.size() > 0) {
    ItemCancelar itemSelecionado = (ItemCancelar) lsItensVendidos.get(0);
}
```

Refatorado:

```
public ItemCancelar buscarInfoItemVendido(String dataVenda, String numItem, String
    itemVinculado, String numOperacao) throws Exception {
    StringBuilder hql = new StringBuilder();
    hql.append(" select iv from ItemVendido iv ");
    hql.append(" where iv.ativo = 1 ");
    ...
    Query query = DB.session().createQuery(hql.toString());
    ...
    return query.uniqueResult()();
}
```

Outra técnica de refatoração utilizada foi a de **auto-encapsular campo**, na qual o atributo era inserido novamente a um objeto do *struts* e posteriormente capturado. A refatoração faz com que o método retorne o objeto.

Original:

```
request.setAttribute("itemSelecionado", itemSelecionado);
```

Refatorado:

```
public ItemCancelar selecionaItensACancelar(String dataVenda, String numItem, String
    itemVinculado, String numOperacao) throws Exception {
    ...
return itemSelecionado;
}
```

Após a refatoração, o método que era grande, com várias operações e altamente acoplado a um framework, se tornou um método que pode ser reutilizado independente do framework, com uma assinatura clara e com a sua responsabilidade bem definida.

Original:

```
public void seleciona(ActionMapping mapping, Form form, HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    try {
```

```
3         String dataVenda = request.getParameter("dataVenda") == null ? null : (request.
              getParameter("dataVenda").equals("null") == true ? null : request.
              getParameter("dataVenda").toString());
4         ...
5         // Buscar o detalhamento do ItemVendido
6         String stringQuery = " select iv from ItemVendido iv " +
7         " where iv.ativo = 1 ";
8         ...
9         Query query = DB.session().createQuery(stringQuery);
10        ...
11        List<ItemCancelar> lsItensVendidos = query.list();
12
13        if (lsItensVendidos.size() > 0) {
14            ItemCancelar itemSelecionado = (ItemCancelar) lsItensVendidos.get(0);
15            ...
16            // Antes de registrar o ItemVendido valida aos produtos relacionados para
                  registrar o total a devolver
17            stringQuery = "" +
18            "from Produto p " +
19            "where p.activo = 1" +
20            "and p.produtoId = :produtoId ";
21            query = DB.session().createQuery(stringQuery);
22            query.setLong("produtoId", itemSelecionado.getProdutoId());
23
24            List<Produto> lsProdutos = query.list();
25
26            BigDecimal totalProdutos = new BigDecimal(0);
27            for (Produto produto : lsProdutos) {
28                totalProdutos = totalProdutos.add(produto.getPrecio());
29            }
30
31            itemSelecionado.setTotalProduto(totalProdutos);
32
33            request.setAttribute("itemSelecionado", itemSelecionado);
34        }
35    } catch (Exception e) {
36        log.error("Erro ao buscar item a cancelar", e);
37    }
38}
```

Refatorado:

```
1 public ItemCancelar selecionaItensACancelar(String dataVenda, String numItem, String
      itemVinculado, String numOperacao) throws Exception {
2     try {
3         ItemCancelar itemSelecionado = buscarInfoItemVendido(dataVenda, numItem,
              itemVinculado, numOperacao);
4         ...
5         List<Produto> lsProdutos = buscaProdutosRelacionados(itemSelecionado.
              getProdutoId());
6
7         BigDecimal totalProdutos = BigDecimal.ZERO;
8         for (Produto produto : lsProdutos) {
9             totalProdutos = totalProdutos.add(produto.getPreco());
10        }
11
12        itemSelecionado.setTotalProduto(totalProdutos);
13
14        return itemSelecionado;
15
16    } catch (Exception e) {
17        log.error("Erro ao buscar item a cancelar", e);
```

```
18      }
19      return null;
20  }
21
22  public ItemCancelar buscarInfoItemVendido(String dataVenda, String numItem, String
        itemVinculado, String numOperacao) throws Exception {
23      StringBuilder hql = new StringBuilder();
24      hql.append(" select iv from ItemVendido iv ");
25      ...
26      Query query = DB.session().createQuery(hql.toString());
27      ...
28      return query.uniqueResult();
29  }
30
31  public List<Produto> buscaProdutosRelacionados(Long produtoId) {
32      StringBuilder hql = new StringBuilder();
33      hql.append(" from Produto p ");
34      hql.append(" where p.activo = 1 ");
35      hql.append(" and p.produtoId = :produtoId  ");
36
37      Query query = DB.session().createQuery(hql.toString());
38      ...
39      return query.list();
40  }
```

As técnicas de refatoração que serão apresentadas a seguir são: as de **mover método**, **mover campo** e **internalizar método**. Essas técnicas foram aplicadas para resolver alguns problemas como o de possuir métodos com assinaturas iguais contendo os mesmos parâmetros (estes parâmetros possuem a assinatura do método *execute* da classe *Action* do *struts*, ao utilizar a herança desta classe é necessária a implementação deste método), possuir um fluxo confuso da chamada de métodos e não possuir retorno correto dos seus métodos. Além da classe, todos os métodos que são utilizados por ela foram refatorados. A refatoração constituiu-se em remover os vários métodos e substituí-los por um único método que mantém o fluxo da operação de forma ordenada. Todas essas refatorações serão apresentadas no código a seguir.

Original:

```
1   public final class BuscaItemCancelar extends Action {
2       public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
3       final ActionForward forward = seleciona(mapping, form, request, response);
4           return forward;
5       }
6
7       private ActionForward seleciona(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
8           ItemController.getInstance().seleciona(mapping, form, request, response);
9           return validarItem(mapping, form, request, response);
10      }
11
12      private ActionForward validarImpostos(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
13          ItemController.getInstance().validarImpostos(mapping, form, request, response);
14          return retornarItem(mapping, form, request, response);
15      }
16
17      private ActionForward validarItem(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
18          ItemController.getInstance().validarItem(mapping, form, request, response);
```

```
19
20          if (ItemController.getInstancia().getItemCancelar() == null) {
21              return retornarItem(mapping, form, request, response);
22          } else {
23              return validarImpostos(mapping, form, request, response);
24          }
25      }
26
27      private ActionForward retornarItem(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
28          ItemController.getInstance().retornarItem(mapping, form, request, response);
29          return null;
30      }
31  }
```

Refatorado:

```
1   public final class BuscaItemCancelar extends Action {
2       public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
3           try {
4               String dataVenda = request.getParameter("dataVenda") == null ? null : (
                    request.getParameter("dataVenda").equals("null") == true ? null :
                    request.getParameter("dataVenda").toString());
5               ...
6               ItemCancelar itemCancelar = ItemController.getInstance().seleciona(
                    dataVenda, numItem, itemVinculado, numOperacao);
7               boolean itemValido = ItemController.getInstance().validarItem(itemCancelar)
                    ;
8
9               boolean impostoValido = false;
10              if (itemValido) {
11                  boolean impostoValido = ItemController.getInstance().validarImpostos(
                        itemCancelar);
12              }
13
14              if(impostoValido){
15                  itemCancelar.setValido(true);
16              } else {
17                  itemCancelar.setValido(false);
18                  itemCancelar.setMsg("ERRO");
19              }
20
21              XStream xstream = new XStream();
22              xstream.alias("ItemCancelar", ItemCancelar.class);
23              String xml = xstream.toXML(itemCancelar);
24
25              RequisicionHttpToXML.enviaRespuesta(response, xml);
26
27          } catch (Exception e) {
28              e.printStackTrace();
29          }
30          return null;
31      }
32  }
```

Para a utilização da técnica de **decompor condicional**, foi adicionada a uma classe especialista em validação de datas um método que faz esta operação. O método do exemplo é o de validarImpostos. A validação de datas foi abstraída. Após a refatoração, a validação é feita pela classe DataUtil. A técnica de refatoração de *substituir variável temporária por consulta* foi

usada para abstração dos cálculos com datas. Ao invés de serem usadas as variáveis temporárias prazoLimiteCompra e prazoLimiteCancelamento, os cálculos destas datas são feitos pela classe DataUtil pelos métodos somarDias e subtrairDias.

Original:

```
public void validarImpostos(ItemCancelar itemCancelar) {
    ...
    Date dataVenda = itemCancelar.getDataVenda();
    Calendar prazoLimiteCompra = Calendar.getInstance();
    prazoLimiteCompra.setTime(dataVenda);
    prazoLimiteCompra.set(Calendar.DAY_OF_MONTH, prazoLimiteCompra.get(Calendar.
        DAY_OF_MONTH) - PRAZO_COMPRA);

    Calendar prazoLimiteCancelamento = Calendar.getInstance();
    prazoLimiteCancelamento.setTime(dataVenda);
    prazoLimiteCancelamento.set(Calendar.DAY_OF_MONTH, prazoLimiteCancelamento.get(
        Calendar.DAY_OF_MONTH) + PRAZO_CANCELAMENTO);

    if (dataVenda.before(prazoLimiteCompra.getTime()) || dataVenda.after(
        prazoLimiteCancelamento.getTime())) {
        taxa = imposto / valorCompra;
    }
    else {
        taxa = 0;
    }
    ...
}
```

Refatorado:

```
public class ItemController {
    ...
    public void validarImpostos(ItemCancelar itemCancelar) {
        ...
        Date dataVenda = itemCancelar.getDataVenda();
        if (DataUtil.validarImposto(dataVenda)) {
            taxa = imposto / valorCompra;
        }
        else {
            taxa = 0;
        }
        ...
    }
    ...
}

public class DataUtil {
    ...
    public static Date somarDias(Date data, int dias)
    ...
    public static Date subtrairDias(Date data, int dias)
    ...
    public static Date validarImposto(Date data)
    ...
}
```

A refatoração de **substituir vetor por objeto** foi utilizada para facilitar o acesso ao resultado de uma consulta. A consulta retornava um vetor de String com elementos que representam tipos de dados diferentes como numerários, booleanos, textos, etc.. Além do problema com os diferentes tipos de dados retornados pelo o vetor, ainda era necessário saber a posição

do vetor referente ao atributo que desejava acessar. Com a refatoração, o vetor foi substituído por um objeto com os atributos contendo o nome e o tipo de cada dado da consulta.

Original:

```
String[] item = new String[3];
item [0] = dataVenda.toString();
item [1] = numItem.toString();
item [2] = numOperacao.toString();
```

Refatorado:

```
ItemCancelar item = new ItemCancelar();
item.setDataVenda(dataVenda);
item.setNumItem(numItem);
item.setNumOperacao(numOperacao);
```

Estes são alguns dos exemplos práticos das refatorações feitas no desenvolvimento do trabalho.

8 RESULTADOS

Como forma de acompanhar a melhoria da qualidade do código com as refatorações, foram coletadas métricas ao longo do desenvolvimento do trabalho As primeiras métricas foram coletadas antes do código ser refatorado para se fazer o comparativo ao final das refatorações. Como o sistema foi dividido em dois projetos, foi feita a apuração das métricas de cada projeto (ABREU et al., 2008).

O primeiro projeto analisado foi o projeto Cliente com os dados das avaliações de qualidade de código do SonarQube e do Metrics.

Tabela 1 – Índices de Qualidade do Projeto Cliente no SonarQube

Métrica	Inicial	Final	Ganho com a Refatoração
Risco	1.870	1.547	17,27%
Risco de bloqueio	3	0	100,00%
Risco crítico	17	7	58,82%
Risco grande	1.601	1.330	16,93%
Risco pequeno	244	206	15,57%
Risco informação	5	4	20,00%
Pendência técnica	71	57	19,72%
Complexidade	2.022	1.730	14,44%
Comentários (%)	11,60%	9,00%	22,41%
Linhas de comentário	1.249	937	24,98%
Documentação pública de API (%)	15,90%	11,80%	25,79%
Indocumentação pública de API	719	732	-1,81%
Linhas Duplicadas (%)	3,50%	3,20%	8,57%

Na Tabela 1, é possível perceber que houve uma melhoria na qualidade. As métricas que devem ser destacadas são as relacionadas a riscos, complexidade e duplicação. Na análise

de risco houve uma melhoria de 17,27%, a complexidade teve um ganho de 14,44% e nas linhas duplicadas o ganho foi de 8,57%. Contudo, os pontos que tiveram a melhora mais significativa foram os de risco de bloqueio e risco crítico que respectivamente tiveram o ganho de 100% e 58,82% segundo à análise do SonarQube.

Tabela 2 – Índices de Qualidade do Projeto Cliente no Metrics

Métrica	Inicial	Final	Ganho com a Refatoração
Linhas de Código por Método	16,39	9,69	40,88%
Aninhamento na Profundidade dos Blocos	1,158	0,978	15,54%
Profundidade da Árvore de Herança	1,853	1,845	0,43%
Acoplamento Aferente	3,825	3,62	5,36%
Complexidade Ciclomática	4,092	2,281	44,26%
Instabilidade	0,305	0,247	19,02%
Número de Parâmetros	1,152	1,142	0,87%
Falta de Coesão de Métodos	0,415	0,412	0,72%
Acoplamento Eferente	3,05	3,01	1,31%
Distância de Normalização	0,294	0,277	5,78%
Abstração	0,35	0,348	0,57%
Índice de Especialização	0,181	0,179	1,10%
Métodos Ponderados por Classe	58,105	48,264	16,94%

A redução das linhas de comentários é devido à refatorações como a de extrair método. Nos casos em que os métodos eram grandes e complexos, os comentários foram retirados e novos métodos com nomes bem definidos à função que ele executa foram implementados.

Tabela 3 – Índices de Qualidade do Projeto Servidor no SonarQube

Métrica	Inicial	Final	Ganho com a Refatoração
Risco	5.459	4.336	20,57%
Risco de bloqueio	3	0	100,00%
Risco crítico	44	22	50,00%
Risco grande	4.209	3.134	25,54%
Risco pequeno	1.196	1.170	2,17%
Risco informação	7	2	71,43%
Pendência técnica	323	217	32,82%
Complexidade	12.923	12.464	3,55%
Comentários (%)	7,30%	3,70%	49,32%
Linhas de comentário	5.789	2.789	51,82%
Documentação pública de API (%)	5,80%	5,50%	5,17%
Indocumentação pública de API	10.993	11.018	-0,23%
Linhas Duplicadas (%)	12,10%	12,00%	0,83%

Na Tabela 2, estão as análises do Metrics no projeto Cliente, que mostram os dados referentes ao desvio padrão do código. A análise da complexidade ciclomática contém um alto ganho de 44,26% após as refatorações. Esta métrica é um ponto importante na análise feita pelo

Metrics. Outra métrica que também possui relevância é a de instabilidade, que possui um ganho de 19,02%.

Posteriormente, foi analisado o projeto Servidor. Como na análise do projeto Cliente, as métricas destacadas foram as de riscos, complexidade e duplicação. A melhoria após as refatorações foram maiores no risco de bloqueio e no risco crítico. Já nos critérios de complexidade e duplicação, o ganho foi baixo de 3,55% e 0,83% respectivamente, e com um ganho de 20,57% no risco. A análise do SonarQube no projeto Servidor pode ser visto na Tabela 3.

Na análise do Metrics no projeto Servidor, a melhoria nos índices de complexidade ciclomática e instabilidade tem o ganho de 22,42% e 0,36% respectivamente. Os ganhos da refatoração nas duas métricas são menores do que no projeto Cliente como pode ser visualizado na Tabela 4.

Tabela 4 – Índices de Qualidade do Projeto Servidor no Metrics

Métrica	Inicial	Final	Ganho com a Refatoração
Linhas de Código por Método	8,601	7,825	9,02%
Aninhamento na Profundidade dos Blocos	0,527	0,523	0,76%
Profundidade da Árvore de Herança	0,111	0,111	0,00%
Acoplamento Aferente	35,583	35,358	0,63%
Complexidade Ciclomática	2,534	1,966	22,42%
Instabilidade	0,275	0,274	0,36%
Número de Parâmetros	2,478	1,881	24,09%
Falta de Coesão de Métodos	0,443	0,428	3,39%
Acoplamento Eferente	69,057	69,057	0,00%
Distância de Normalização	0,296	0,251	15,20%
Abstração	0,111	0,11	0,90%
Índice de Especialização	0,151	0,148	1,99%
Métodos Ponderados por Classe	53,828	44,363	17,58%

Todas as tabelas mostram um ganho na qualidade de software após as refatorações, as ferramentas usadas foram utilizadas em seu modo *default*, sem customização dos índices de análise. As métricas referentes ao projeto Cliente possuem um maior ganho em relação ao projeto Servidor.

Com as análises das métricas obtidas após a refatoração, é visível que existe uma grande diferença nos ganhos de refatoração no projeto Cliente em relação ao projeto Servidor.

Essa diferença ocorre por vários motivos. Uma delas é que no projeto Servidor, algumas classes não foram refatoradas pelo fato de serem classes globais, muitos outros módulos do sistema utilizam os recursos das mesmas. Outro motivo pelo maior ganho no projeto Cliente, principalmente nas métricas relacionadas a complexidade, é de que no projeto Cliente haviam classes com excessos de relacionamentos entre outras classes, excessos de comportamentos e principalmente condicionais muito longas e complexas.

As técnicas de refatoração que mais foram utilizadas no projeto Cliente foram: decompor condicional, substituir vetor por objeto, mover método e substituir variável temporária por

consulta. Estas técnicas ajudaram o projeto Cliente a ter maiores ganhos nos índices de refatoração em relação ao projeto Servidor.

9 CONCLUSÃO

Este trabalho aplicou dez técnicas de refatoração em um sistema legado, mostrando como a refatoração interferiu diretamente na qualidade do software e o retorno de investimento que pode trazer a projetos com problemas.

As técnicas usadas foram: parametrizar método, extrair método, internalizar método, substituir variável temporária por consulta, auto-encapsular campo, mover método, mover campo, renomear método, decompor condicional e substituir vetor por objeto. Em seguida as métricas do SonasQueb e do Metrics foram utilizadas nos projetos de Cliente e Servidor.

As métricas são apenas comparativas, com o intuito de demonstrar que a refatoração pode melhorar a qualidade de software do sistema. Porém, as métricas de software são complexas, pois, deve se levar em consideração o que se deseja melhorar no sistema. Em determinados casos, os critérios de métricas podem ser customizados. Os gerentes de projeto de software podem definir como será a coleta dos índices de análise, além de quais métricas são importantes para a melhoria da qualidade de software do sistema específico.

Além da melhoria da qualidade de software após as refatorações, foi demonstrado como a inteligibilidade do código se tornou mais compreensível. Apesar de ter sido feita a refatoração em apenas um módulo do sistema, um projeto de refatoração bem definido e planejado, pode levar este sistema a ter uma vida útil muito maior. Além de diminuir o esforço para manutenções e novos desenvolvimentos dos seus desenvolvedores, assim diminuindo os custos de desenvolvimento no sistema.

9.1 Trabalhos futuros

Para possíveis trabalhos futuros, sugerimos um estudo com as métricas de sistema e desenvolvimento do processo de refatoração, com o intuito de melhorar determinadas métricas específicas. Além de um estudo específico de como definir as métricas de software de acordo com o problema. Outro trabalho futuro, seria o de fazer refatorações, com o objetivo de metrificar cada técnica, apontando quais técnicas de refatoração são as melhores para o percentual em determinadas métricas. Outro tema que poderia ser estudado é o de ferramentas para auxiliar na refatoração.

Referências

ABREU, Thamine Chaves de; ARAÚJO, Marco Antônio Pereira; MOTA, Leonardo da Silva. Métricas de software como utilizá-las no gerenciamento de projetos de software. **Engenharia de Software Magazine**, p. 50 – 55, 2008.

ARBOLEDA, H.; PAZ, A.; ROYER, J. Component-based java legacy code refactoring. **Revista Facultad de Ingeniería Universidad de Antioquia**, n. 68, p. 104–114, 2013.

BOEHM, Barry W. **Software Engineering Economics**. [S.l.]: Prentice Hall, 1982.

CHECKSTYLE. **Checkstyle**. 2014. Disponível em: <http://checkstyle.sourceforge.net/>.

EMAM, Khaled El. **The ROI from Software Quality**. [S.l.]: Auerbach Publications, 2005.

FINDBUGS. **FindBugs**. 2014. Disponível em: <http://findbugs.sourceforge.net/>.

FOWLER, Martin. **Refatoração: Aperfeiçoando O Projeto de Código Existente**. [S.l.]: Bookman, 2004.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Pearson Education, 1994.

GOMES, Nelma da Silva. Qualidade de software - uma necessidade. v. 5, 2008.

GRIFFITH, I.; WAHL, S.; IZURIETA, C. Evolution of legacy system comprehensibility through automated refactoring. **MALETS '11 Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering**, p. 35–42, 2011.

HIBERNATE. **Hibernate**. 2015. Disponível em: <http://hibernate.org/orm/>.

KERIEVSKY, Joshua. **Refatoração para Padrões**. [S.l.]: Bookman, 2008.

KRISTIANSEN, Erlend; STOLZ, Volker. Search-based composed refactorings. **Norsk Informatikkonferanse (NIK)**, 2014.

MCCABE, Thomas. A complexity measure. **IEEE Transactions on Software Engineering**, v. 2, n. 4, p. 308, 1976.

MECENAS, Ivan. Java 6: Fundamentos, swing, bluej e jdbc. **3ª Edição, Rio de Janeiro, Alta Book**, 2008.

METRICS. **Metrics**. 2014. Disponível em: <http://metrics.sourceforge.net/>.

PEREZ, J.; CRESPO, Y. A refactoring discovery tool based on graph transofrmation. **Proceedings of the 1st Workshop on Refactoring Tools**, p. 11–12, 2007.

PMD. **PMD**. 2014. Disponível em: <http://pmd.sourceforge.net/>.

SCHÄFER, M. et al. Correct refactoring of concurrent java code. **ECOOP'10 Proceedings of the 24th European conference on Object-oriented programming**, p. 225–249, 2010.

SONARQUBE. **SonarQube Documentation**. 2014. Disponível em: <http://docs.codehaus.org/display/SONAR/Documentation>.

STRUTS. **Struts**. 2015. Disponível em: <http://struts.apache.org/primer.html>.