Revista de Divulgação Científica em
Ciências Exatas e Tecnológicas
PORANDU Vol. 2, n. 1, 2018, pg.25-38

Manutenção de Software Utilizando Práticas de Codificação Limpa

C. L. SILVA[1,*] M. TORRES[1,] F. FLORIAN[1,]

[1] Universidade de Araraquara - UNIARA, Araraquara-SP, Brasil,

Submetido em 05/09/2017; Aceito em 21/12/2017; Publicado em 16/01/2018.

Resumo. A manutenção de software é a modificação de um produto de software após a sua entrega, visando corrigir defeitos, melhorar desempenho ou outros atributos. Diferente das outras fases presentes no desenvolvimento de software, a manutenção é um processo que acompanha todo o ciclo de vida do produto. Por esse e outros motivos, a manutenção é um requisito cada vez mais valorizado pela indústria de software. A codificação, por sua vez, também é uma etapa que merece atenção, pois ter um código que é apenas funcional não é o bastante, é necessário ter um código autoexplicativo, com módulos coesos e com padrões de design que facilitem a expansão e a reutilização. Partindo do princípio que a codificação é uma das etapas no desenvolvimento de software que tem impacto direto sobre a manutenção, este trabalho evidencia as más práticas adotadas durante a codificação e propõe, embasado em uma pesquisa bibliográfica, ações de melhoria no código que visam contribuir para uma manutenção mais eficiente e menos custosa.

Palavras-chave. Manutenção de Software; Codificação; Código Limpo; Padrões de Design.

Abstract. Software maintenance is the modification of a software product after its delivery, aiming to correct faults, improving performance or other attributes. Unlike other phases in software development, the maintenance process follows the product for all its life cycle. Not only for this reason but also for others, the maintenance is a requirement that software industry has given a great importance. Coding phase also deserves all cares, because just having a functional code is not enough, it is needed to be self-explanatory, with cohesive modules and with design standards which make easier expansion and reuse. Assuming that coding is a step in software development that has direct impact over maintenance, this work shows bad practices adopted during coding phase and suggests, based on a bibliographical research, improvement actions in code that aim at helping for an efficient and less expensive maintenance.

*clebersilva228@gmail.com

1 Introdução

Tendo em vista que o ciclo de vida de um software é mantido por equipes de desenvolvimento diferentes, em épocas distintas e por desenvolvedores com níveis de maturidade divergentes, é previsível que o processo de manutenção deste software se torne uma tarefa cada vez mais complexa [1].

Grande parte do custo de um projeto de software está em sua manutenção. Para um desenvolvedor prover uma manutenção eficiente, é essencial que este entenda rapidamente o que o sistema faz e assim efetuar a intervenção de maneira ágil. Contudo, essa tarefa não é simples, pois além da dificuldade em compreender as funcionalidades do sistema, existe também a preocupação de que alterações no código existente possam resultar em novos defeitos.

Para Martin [2], à medida que a complexidade de um código aumenta, desenvolvedores levam mais tempo para compreendê-lo, com isso, crescem as chances destes desenvolvedores interpretarem o código incorretamente, o que eleva o risco de introdução de erros neste software. Portanto, para amenizar as chances de interpretações precipitadas, o código deve expressar claramente a intenção do seu autor, sendo que, quanto mais claramente o autor expressa o seu código, menos tempo outros desenvolvedores irão demorar em compreendê-lo. Isso reduzirá os defeitos e diminuirá o custo de manutenção.

A utilização de práticas de codificação limpa tem como intuito tornar a manutenção do software mais simples e menos propensa a erros, pois com a utilização dessas práticas, o código tende a ser mais fácil de ser compreendido. Como consequência, um desenvolvedor que efetuar manutenção em um sistema que é beneficiado por essas práticas, poderá ser capaz de realizar esta ação de maneira mais eficiente.

O objetivo deste trabalho consiste em propor a utilização de conceitos e práticas de codificação limpa que tendem a auxiliar no processo de manutenção de software.

Blanchard [3] diz que manutenibilidade é uma característica inerente a um projeto de sistema ou produto, e se refere à facilidade, precisão, segurança e economia na execução de ações de manutenção nesse sistema ou produto.

Considerando a definição de Blanchard para manutenibilidade, torna-se desejável adotar práticas durante o desenvolvimento de software que beneficiem a manutenção.

A principal adversidade quanto a manutenção de software é que à medida que um software cresce sem a utilização de práticas de codificação limpa, torna-se cada vez mais difícil de mantê-lo. Como consequência, o custo e o tempo de criação de novas funcionalidades ou correções de defeitos tornam-se cada vez maiores.

Para Martin [2], a ausência de práticas de codificação limpa pode causar uma desaceleração significante no desenvolvimento de um software. Martin ressalta que times que produzem rapidamente no início de um projeto podem ter sua produtividade consideravelmente afetada, pois em um software que possui alta complexidade no código, cada nova implementação tende a quebrar duas ou três funcionalidades deste software, ou seja, nenhuma mudança se torna trivial. Com o tempo, a complexidade se torna tão grande e tão profunda, que fica difícil encontrar uma solução. Como consequência da desaceleração no desenvolvimento, a gestão de pro-

jetos adiciona mais pessoas no projeto na esperança de aumentar a produtividade. Contudo, novos integrantes não têm conhecimento sobre o design do sistema, não sabem a diferença entre uma mudança que corresponde ao intuito do projeto e uma mudança que frustra o intuito do projeto. Além disso, essas pessoas e todos do time acabam ficando sob uma terrível pressão para aumentar a produtividade. Então, tudo o que fazem é aumentar ainda mais a complexidade do sistema, conduzindo a produtividade próxima de zero.

Martin conclui que todo projeto que não é beneficiado com práticas de codificação limpa tem sua produtividade comprometida. A Figura (1) demonstra a perda de eficiência na produtividade com o passar do tempo em um projeto com essa característica.

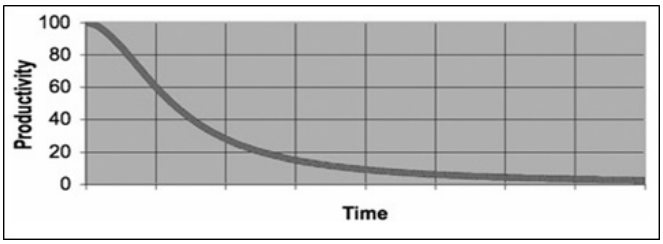

Figura 1: Queda de produtividade em um projeto sem práticas de código limpo [2].

2 Metodologia

A metodologia de pesquisa deste trabalho foi classificada quanto a (os): abordagem, em pesquisa qualitativa; natureza, em pesquisa aplicada; objetivos, em pesquisa exploratória; procedimentos técnicos, em pesquisa bibliográfica [4].

A abordagem é considerada qualitativa, uma vez que é baseada na aceitação do plano de verificação de qualidade a ser proposto, ou seja, valores subjetivos estão envolvidos, caracterizando a abordagem como qualitativa. No que diz respeito à natureza, é aplicada, pois tem como objetivo a elaboração de uma estratégia de verificação da qualidade do produto de software em um contexto prático, ou seja, direciona os conhecimentos para à resolução de problemas específicos [5]. Quanto ao objetivo, o trabalho é caracterizado como exploratório, pois tem como foco a investigação de aspectos de implementação que resultam em qualidade do produto de software. E o principal procedimento técnico adotado será a Pesquisa Bibliográfica.

Todo código utilizado neste trabalho, tanto para ilustração dos exemplos, quanto para realização de testes, utilizam a linguagem Java e o paradigma Orientado a Objetos.

3 Código Limpo

Para Thomas [1], um código limpo pode ser lido e aperfeiçoado por um desenvolvedor diferente do original, possui testes de unidade e de aceitação, contém

nomes significativos, não possui duplicações e tem dependências mínimas que são explicitamente definidas.

Código limpo é um código elegante e eficiente. A lógica deve ser direta para que possíveis defeitos se tornem perceptíveis, as dependências devem ser mínimas a fim de facilitar a manutenção, o tratamento de erros deve seguir uma estratégia bem articulada. O desempenho deve estar próximo do ideal, de modo que outros desenvolvedores não fiquem tentados em realizar melhorias precipitadas [6].

Para Jeffries [7], código limpo tem as seguintes características em ordem de prioridade:

- Executa com sucesso todos os testes;
- Não contém código duplicado;
- Expressa com clareza todas as ideias de design que estão no sistema;
- Minimiza o número de entidades, como classes, métodos e funções.

3.1 Nomes dos Elementos do Código

Em software, os nomes estão por toda parte. Nomes são dados as variáveis, funções, argumentos, classes e pacotes. Dar nomes é um contínuo e exaustivo processo, por isso, é conveniente que essa tarefa seja efetuada de maneira apropriada [8]. Para Martin [2], o nome de uma variável, função ou classe deve responder toda e qualquer pergunta. O nome deve dizer o porquê da existência, o que faz e como o elemento é utilizado.

```
private int d;
```

Figura 2: Nome Impróprio de Variável.

A Figura (2) exibe uma variável de instância que é utilizada em um sistema real de rastreamento, sua finalidade é armazenar o valor do tempo (em dias) que determinado produto permaneceu em transporte.

O nome `d` não revela muita coisa, pois não invoca um senso de tempo de transporte e nem dias. A Figura (3) exemplifica a mesma variável da Figura (2). Desta vez, o nome especifica o que está sendo medido e a unidade de medida utilizada. Dessa maneira, um desenvolvedor que efetuar manutenção neste código, deduzirá com facilidade a função dessa variável.

```
private int tempoDeTransporteEmDias;
```

Figura 3: Nome Próprio de Variável.

Nomear apropriadamente os elementos de um software é uma ação relevante, uma vez que isso ajuda a identificar similaridades no código e auxilia no reconhecimento de padrões de design. Quando um programador esta tentando se familiarizar com o código de um novo projeto, a clareza dos nomes dos elementos contribuirão para variar o grau de dificuldade desta tarefa [2].

3.2 Comentários no Código Fonte

Segundo McConnell [9], um código de boa qualidade é a melhor documentação; é mais adequado melhorar a legibilidade do código a optar pela utilização de comentários.

Contudo, um comentário acaba sendo uma alternativa atraente quando o código escrito não é autoexplicativo, deste modo, um desenvolvedor sente-se tentado em adicionar comentários a fim de compensar a falta de legibilidade em seu código. Para Martin [2], o grande problema dessa abordagem é que um comentário não tende a acompanhar a evolução do código ao qual ele descreve, pois de fato, a própria natureza de um comentário não é vital para as funcionalidades do software e, por conta desta característica, torna-se optativo para um desenvolvedor evoluir, incluir, ou remover um comentário. Com isso, comentários tendem a se tornar desconexos, espalhando falsas informações.

A Figura (4) exibe um comentário em um trecho de código de um sistema real que, a princípio, aparenta descrever a característica da variável `saveLocale`.

```
private MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Figura 4: Comentário Desconexo.

Utilizando o software de controle de versionamento desse sistema, foi possível identificar a origem do comentário da Figura (4). O comentário originalmente descrevia o comportamento da constante `HTTP_DATE_REGEXP`. Com isso, conclui-se que o comentário foi separado da constante a qual originalmente descrevia porque variáveis de instância foram adicionadas posteriormente entre eles. A Figura (5) exibe o trecho de código no qual o comentário foi originalmente posicionado.

```
private MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Figura 5: Origem do Comentário Desconexo.

Segundo McConnell [9], uma das maiores motivações para se escrever um comentário é ter escrito um código ruim; quando desenvolvedores escrevem certo módulo e percebem que este módulo está confuso e desorganizado, sentem-se tentados em adicionar um comentário para compensar a falta de clareza.

A Figura (6) evidencia a posição de McConnell demonstrando um trecho de código de um sistema real. Esse código faz parte de uma função que verifica se um empregado é elegível a ter benefícios totais. Possivelmente, o autor entendeu que o seu código necessitava ser mais transparente e utilizou um comentário para suprir essa necessidade.

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Figura 6: Comentário que Descreve uma Funcionalidade.

Fowler [8] explica que todo trecho de código que não expressa com clareza a sua funcionalidade, deve ser abstraído para uma nova função. Esta nova função deve ter como objetivo clarificar a funcionalidade do código. Desta forma, o trecho de código da Figura (6) pode ser transferido para uma nova função que possua um nome que auto explique seu comportamento, tornando o comentário desnecessário. A Figura (7) exemplifica essa ação.

```
if (employee.isEligibleForFullBenefits())
```

Figura 7: Refatoramento da Função.

3.3 Funções do Código Fonte

Outro conceito essencial em software são as funções. Através das funções são armazenados trechos de código que executam tarefas. E sempre quando necessário, invoca-se essas funções para executar certa tarefa, com isso, evita-se reescrever o mesmo código para solucionar tarefas semelhantes [10].

Martin [2] diz que além de pequena, uma função deve realizar uma única tarefa e de maneira satisfatória, ou seja, o desempenho dessa função deve estar próximo do

ideal e o código intuitivo o bastante para ser facilmente interpretado por qualquer desenvolvedor.

Fowler [8] faz um alerta à utilização de funções extensas, na sua visão, grande partes dos problemas em software são gerados a partir de funções que possuem tamanhos além do ideal. Funções extensas escondem informações valiosas, pois naturalmente possuem lógicas complexas que ocultam detalhes relevantes do código.

A Figura (8) exibe a função `printOwing`, ela possui um tamanho possivelmente redutível. Essa função, além de extensa, realiza tarefas fora do escopo de sua responsabilidade (exibir valores), ferindo o princípio da responsabilidade única, princípio que foca na preocupação de que uma classe tenha seu papel e venha desempenhar somente esse papel de forma eficiente.

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Figura 8: Função Extensa.

A Figura (9) exibe a mesma função `printOwing` refatorada por Fowler [8]. Observa-se que houve uma extração do código fora de escopo para novas funções com responsabilidades bem definidas. Consequentemente, essa função torna-se coesa, menor e adequada ao princípio da responsabilidade única.

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}
```

Figura 9: Função Refatorada.

Kernighan e Plaugher [11] observam que quando o código não é "quebrado"em pequenas partes, os módulos tornam-se mais suscetíveis a erros, pois com este as-

pecto costumam a ter mais responsabilidades do que deveriam. Como consequência, esses módulos se tornam mais difíceis de manter e muito especializados para serem reutilizados.

3.4 Funções de Validação

Segundo Gama [12], um desenvolvedor que cria uma nova função de validação, normalmente escolhe entre duas maneiras: que a função retorne um valor booleano ou um código de erro específico.

Martin [2] aponta que essa abordagem cria problemas para os clientes que invocam essa função, pois cada um deles terá o trabalho de verificar o valor de retorno, logo um sistema que possui funções validadoras com essa característica, tende a conter muitas operações condicionais, o que implica na perda de desempenho e na poluição e duplicação de código.

Uma vez que uma função cliente verifique o retorno de uma validação e também execute outros comandos, este aspecto caracteriza a violação do princípio da separação de comando e consulta. Este princípio preza que toda função deva executar comandos (ações) ou consultas, mas nunca ambos [13]. A violação desse princípio é exemplificada pela Figura (10), onde se percebe que a função cliente realiza uma operação de comando e uma operação de consulta. Isso acontece porque a função validadora retorna um valor booleano como resposta da sua validação.

```
//função cliente
if (isValidCode(100)){
   doSomething();
}

//função validadora
private boolean isValidCode(int code){
    if (code != 100){
       return false;
    }
    return true;
}
```

Figura 10: Violação do Princípio da Separação de Comando e Consulta.

Bugayenko [14] diz que o intuito inicial da inclusão de exceções no paradigma orientado a objetos é retirar do cliente a responsabilidade de checar o retorno de uma função. Assim, ao utilizar exceções em funções validadoras, contribui-se para não infringir o princípio da separação de comando e consulta.

Martin [2] instrui para a utilização de exceções em funções validadoras ao invés de retornos booleanos ou códigos de erro, justificando que esta técnica desacopla função cliente e função validadora. Essa abordagem é exemplificada através da Figura (11). Nota-se que agora a função cliente não viola mais o princípio da separação de comando e consulta, pois esta função executa apenas comandos. A

função cliente agora invoca a função validadora sem a preocupação de checar o seu retorno.

```
//função cliente
validateCode(100);
doSomething();

//função validadora
private void validateCode(int code){
    if (code != 100){
       throw new InvalidCodeException();
    }
}
```

Figura 11: Utilizando Exceções para Efetuar Validações.

3.5 Retorno Nulo de Funções

Segundo Bugayenko [14], retorno nulo no paradigma orientado a objetos é uma péssima prática que deve ser evitada a qualquer custo. Em um sistema cujas funções tenham essa característica, sempre quando objetos são recebidos como parâmetros de entrada é necessário verificar se este objeto é nulo. Como resultado, haverá um código poluído por operações condicionais. Além disso, declarações do tipo `if (employee == null)` atrapalha o "pensamento orientado a objeto", pois introduz conceitos computacionais. Uma declaração que contenha a palavra-chave `null`, é entendida apenas por pessoas que saibam que objeto em Java é um ponteiro e que `null` é um apontamento para nada. A Figura (12) exibe um trecho de código comum encontrado em linguagens orientada a objetos.

```
Employee employee = employees.get("Jeffrey");
if (employee == null) {
  throw new EmployeeNotFoundException();
}
return employee;
```

Figura 12: Checagem por Nulo.

O Quadro (1) representa a Figura 12 em forma de um diálogo, onde a última pergunta soa bem estranho [14].

- Alô, é do dapartamento de tecnologia?
- Sim.
- Posso falar com o empregado, Jeffrey?
- Aguarda na linha, por favor...
- Alô.
- Olá, você é o nulo?

Quadro 1: Simulação de Diálogo [14].

Bugayenko [14] afirma que `null` traz dificuldades em uma linguagem orientada a objeto; utilizar `null` em retornos de funções frustra a utilização do polimorfismo, uma vez que para efetuar chamadas a partir de um objeto, antes será necessário realizar uma checagem a fim de não correr o risco de ocasionar um erro em tempo de execução (`NullPointerException`). Porém, essa checagem repetitiva resulta em duplicação de código, um dos grandes erros no desenvolvimento de software.

Fowler [15] sugere a utilização do padrão "Caso Especial"como uma solução para este problema.

A utilização deste padrão consiste na criação de uma subclasse que implementa a interface que o cliente espera, fornecendo um comportamento especial para casos particulares. Assim, ao invés de retornar nulo, retorna-se essa subclasse. A Figura (13) exibe a implementação do padrão.

```
public interface Animal {
    void makeSound();
}

public class Dog implements Animal {
    public void makeSound() {
        System.out.println("woof!");
    }
}

// Special Case Object
public class NullAnimal implements Animal {
    public void makeSound() {
        throw new AnimalNotFoundException();
    }
}
```

Figura 13: Implementação do Padrão Caso Especial.

Um cliente que espera por `Dog` também pode esperar por `NullAnimal`, pois ambos implementam a interface `Animal`. Com isso, é possível utilizar o padrão Caso Especial conforme demonstra a Figura (14).

```
//client
Animal animal = findAnimal(id);
animal.makeSound();


public Animal findAnimal(int id){
    Animal animal = animalDao.findAnimal(id);

    if (animal == null){
        // returns a Special Case Object if null
        return new NullAnimal();
    }

    return animal;
}
```

Figura 14: Utilização do Padrão Caso Especial.

Nota-se que a função `findAnimal` retorna um objeto especial(`NullAnimal`) quando nenhum animal é encontrado. Por essa razão, o cliente que chama a função `findAnimal` não precisa se preocupar se este objeto é nulo, eliminando a necessidade de checagem por nulo. Percebe-se também que quando o método `makeSound` é chamado em uma instância de `NullAnimal`, será lançado uma `AnimalNotFoundException`, indicando que o `Animal` não foi encontrado.

4 Resultados

Para realizar a análise do resultado da aplicação de práticas de codificação limpa, foram estudas métricas de software relacionadas à manutenibilidade a fim de se obter dados referentes a um sistema. As mesmas métricas foram analisadas no sistema original e no sistema reestruturado com as práticas.

Para avaliar o impacto da adoção de práticas de codificação limpa, realizou-se uma comparação dos dados coletados avaliando quantitativamente se houveram melhorias na manutenibilidade do software estudado.

O SISTEMA analisado é uma aplicação web desenvolvida em Java, utilizando a plataforma Java EE Web Application, e utiliza um banco de dados MySQL. O sistema é executado em um container de aplicações web, Tomcat.

O SISTEMA tem como finalidade possibilitar o gerenciamento e apoio às atividades de um Instituto de Pesquisa, desde o gerenciamento dos dados da equipe e clientes, até o acompanhamento da execução de projetos.

Após a definição e caracterização do software a ser estudado foi realizado uma análise para identificar falhas, deficiências e possibilidades de melhorias, com base em práticas de codificação limpa.

Os itens em foco de melhorias foram:

- Duplicação de código.
- Funcionalidades impactadas indiretamente por alterações no código fonte.
- Complexidade de classes e métodos.

- Números de linhas excessivos.
- Baixa reutilização do código.
- Baixa eficiência na utilização do paradigma orientado a objetos.

As duas versões do sistema passaram pelo mesmo tipo de análise utilizando a ferramenta *CodePro Analityx* (software que permite coletar métricas de código-fonte). Essa ferramenta forneceu um conjunto de medidas que representam diversos aspectos de qualidade.

A Tabela (1) apresenta o resultado das métricas coletadas: característica da métrica, os valores do sistema original (SISTEMA), do Sistema versão2 (SISTEMA 2), e a variação percentual entre as duas medidas.

Métricas	**Característica**	**SISTEMA**	**SISTEMA 2**	**Variação**
N° classes	$\Leftrightarrow$	60	50	-17%
avCYCLO	$\Downarrow$	1,74	1,36	-22%
WMC	$\Downarrow$	1180	644	-45%
avDIH	$\Leftrightarrow$	2,53	2,29	-9%
ABS(%)	$\Uparrow$	0	6	+6%
LOC	$\Downarrow$	7195	3446	-52%
avMLOC	$\Downarrow$	9,21	5,76	-37%
avPAR	$\Downarrow$	0,89	0,75	-16%
MI	$\Uparrow$	-38,213	-33,182	+13%
K3B	$\Downarrow$	1,8738	1,7027	-9%

$\Uparrow$ = quanto maior valor melhor; $\Downarrow$ = quanto menor melhor;
$\Leftrightarrow$ = valor de referência, maior ou menor não tem significado.

Tabela 1: Resultado geral das métricas coletadas.

Os resultados apresentam melhorias em todas as métricas analisadas. As métricas de complexidade (avCYCLO) indicam que a reestruturação possibilitou reduzir em 22% a complexidade dos métodos, o que também pode ser verificado pelo número médio de linhas por método (avMLOC) com uma redução de 37%.

O tamanho total do sistema em linhas de código (LOC) também foi reduzido em 52%. O Nível de abstração (ABS) do SISTEMA era 0%, e após a reestruturação possui 6%. Isso indica um melhor uso da orientação a objetos, o que também é evidenciado pelo indicador avDIH, que teve um aumento de 9%.

Quanto a média de número de parâmetros (avPAR) houve uma redução de 16%. O mesmo acontece para métodos ponderados por classe (WMC), onde se constatou uma redução positiva de 45%.

A métrica K3B, que avalia a propagação de modificações ao se modificar uma determinada classe reduziu 9%. O índice de manutenibilidade (MI), que é um dos índices mais significativos para este trabalho, apresentou um ganho de 13%.

5 Conclusão

Este trabalho propôs a utilização de conceitos e práticas de codificação limpa que auxiliam o processo de manutenção de software.

Para avaliar impacto da utilização de práticas de codificação limpa na manutenibilidade de um software, foram aplicados práticas de codificação limpa em um sistema comercial desenvolvido em Java e mediu-se o nível de manutenibilidade do sistema original e do sistema reestruturado.

Todas as medidas do sistema reestruturado superaram as medidas do sistema original. A reestruturação do sistema proporcionou notáveis melhorias na qualidade do software. Considerando que as métricas de software aplicadas podem ser usadas para mensurar o grau de esforço para realizar a manutenção em um sistema, pode-se afirmar que o sistema reestruturado possui melhor manutenibilidade.

Referências

[1] A. HUNT and D. THOMAS, *The Pragmatic Programmer.* Boston, EUA: Addison-Wesley, 2000.

[2] C. R. MARTIN, *Clean Code: A Handbook of Agile Software Craftsmanship.* Upper Saddle River, EUA: Prentice Hall, Upper Saddle River, 2008.

[3] B. BLANCHARD, *Logistics Engineering and Management.* Upper Saddle River, EUA: Pretince Hall, 1992.

[4] T. E. GERHARDT and D. T. SILVEIRA (orgs.), *Métodos de Pesquisa.* Porto Alegre: Editora da UFRGS, 2009.

[5] A. C. GIL, *Como Elaborar Projetos de Pesquisa.* São Paulo: Atlas, 5 ed. ed., 2010.

[6] B. STROUSTRUP, *The C++ Programming Language.* Upper Saddle River, EUA: Addison-Wesley Professional, 2013.

[7] R. JEFFRIES, *Extreme Programming Installed.* Boston, EUA: Addison-Wesley Professional, 2000.

[8] M. FOWLER, *Refactoring: Improving the Design of Existing Code.* Boston, EUA: Addison-Wesley, 1999.

[9] S. McCONNELL, *Code Complete: A Practical Handbook of Software Construction.* Redmond, EUA: Microsoft Press, 2004.

[10] M. FEATHERS, *Working Effectively with Legacy Code.* Boston, EUA: Addison-Wesley, 2004.

[11] KERNIGHAN and PLAUGHER, *The Elements of Programming Style.* New York, EUA: McGraw-Hill, 2d. ed. ed., 1978.

[12] E. GAMMA, *Design Patterns: Elements of Reusable Object Oriented Software*. Boston, EUA: Addison-Wesley, 1996.

[13] M. FOWLER, "Command Query Separation." `https://martinfowler.com/bliki/CommandQuerySeparation.html`, Dez. 2005. Acesado em 11 abr. 2017.

[14] Y. BUGAYENKO, "Why Null is Bad." `http://www.yegor256.com/2014/05/13/why-null-is-bad.html`, Maio 2014. Acessado em 11 abr. 2017.

[15] M. FOWLER, *Patterns of Enterprise Application Architecture*. Boston, EUA: Addison-Wesley, 2002.