FACULDADE DE TECNOLOGIA DE  
SÃO JOSÉ DO RIO PRETO

A Importância do Código Limpo na Perspectiva dos Desenvolvedores e Empresas de Software

JOBERTO DINIZ JUNIOR

Orientador

Prof. Dr. Djalma Domingos da Silva

São José do Rio Preto

2013

SUMÁRIO

[CAPÍTULO 1. INTRODUÇÃO 1](#_Toc351397352)

[1.1 Apresentação do Tema 1](#_Toc351397353)

[1.2 Justificativa 2](#_Toc351397354)

[1.3 Problema 2](#_Toc351397355)

[1.4 Objetivos 3](#_Toc351397356)

[CAPÍTULO 2. FUNDAMENTAÇÃO TEÓRICA 4](#_Toc351397357)

[2.1 O custo de um código-fonte ruim 4](#_Toc351397358)

[2.2 Definição de Código Limpo 4](#_Toc351397359)

[2.3 Nomes significativos 5](#_Toc351397360)

[CAPÍTULO 3. METODOLOGIA 7](#_Toc351397361)

[3.1 Tipo do trabalho 7](#_Toc351397362)

[3.2 Coleta de dados 7](#_Toc351397363)

[3.3 Desenvolvimento 7](#_Toc351397364)

[3.4 Cronograma 7](#_Toc351397365)

[REFERÊNCIAS 9](#_Toc351397366)

LISTA DE ABREVIATURAS

IDE Ambiente Integrado de Desenvolvimento

SRP Princípio da Responsabilidade Única

SOLID Princípio da Responsabilidade Única, Princípio Aberto-Fechado, Princípio da Substituição de Liskov, Principio da Segregação de Interface, Principio da Inversão de Dependência

1. INTRODUÇÃO

O desenvolvimento de sistemas é uma área em que a tecnologia está em constante e acelerada mudança, fato que dificulta o profissional especializado a manter-se atualizado. A busca pelo aprendizado é importante, e deve ser continuo para, assim, assegurar a vaga no mercado de trabalho. Diante disso, desenvolvedores se veem na obrigação de estarem aprendendo algo novo diariamente, seja por uma notícia de uma nova tecnologia, ou por vontade de aprender mais. Sem dúvida há sempre novas habilidades para adquirir, treinar e dominar. O Código Limpo é uma delas.

Dos anos noventa para cá, observa-se diversas mudanças no dia-a-dia de desenvolvimento de sistemas. Poderosos Ambientes de Desenvolvimento Integrado (IDE) surgiram, controles de versões de arquivos foram criados, novos padrões de projetos nasceram, processos de desenvolvimento e paradigmas de programação amadureceram, grande parte para facilitar e melhorar o ciclo de vida de sistemas.

Entretanto uma tarefa parece não ter mudado muito: a manutenção do código-fonte. Diferentemente do senso comum, programas são lidos mais frequentemente do que eles são escritos (Beck, 2007). Constantemente lemos código antigo como parte do esforço para criar um novo. Isso se deve, principalmente, ao atraso que o código ruim proporciona (Martin, 2008).

Se certas técnicas de Código Limpo fossem aplicadas na concepção do código, esse cenário poderia ser diferente. Desenvolvedores não ficariam tão frustrados em participar de projetos que eles não fizeram parte, gerentes ficariam entusiasmados com a velocidade da equipe e clientes mais satisfeitos com a qualidade dos resultados.

Este estudo espera apresentar a importância do Código Limpo tanto para os desenvolvedores quanto para as empresas de software, visto que o código é a linguagem na qual expressamos os requisitos e particularidades do sistema (Martin, 2008). Ele deve expressar a sua real intenção, deve ser flexível, deve ser cuidado com responsabilidade (Martin, 2008).

## Apresentação do Tema

Em um sistema sempre haverá código, e diante desse código sempre haverá manutenções e novos requerimentos. Um código escrito hoje será, indubitavelmente, modificado amanhã. Demore semanas, meses ou anos, o fato é que um código não perdurará na forma que foi concebido. Um código bem escrito pode facilitar a sua manutenção, aumentar a produtividade dos desenvolvedores que o encontram, diminuir o tempo despendido para adicionar uma nova funcionalidade ou substituir uma regra antiga, e consequentemente, diminuir os recursos financeiros necessários a realização dessas tarefas.

Um bom profissional deve conhecer e aplicar constantemente as técnicas do Código Limpo; estas serão apresentadas no decorrer deste trabalho. Escrever Código Limpo é muito importante e poucos desenvolvedores parecem conhecê-lo; este trabalho vai propor uma mudança nesse paradigma, mostrando o que é um Código Limpo e sua importância.

## Justificativa

Do ponto de vista do desenvolvedor, conhecer as técnicas do Código Limpo trará uma mudança de paradigma de como escrever código e melhorar profissionalmente. De acordo com Martin (2008) “escrever código limpo é o que você deve fazer a fim de se intitular um profissional. Não há nenhuma desculpa razoável para fazer nada menos que o seu melhor”. Ficará evidente que ler um código limpo é melhor do que qualquer outro código já lido. Espera-se que esse sentimento seja passado adiante, pois o desenvolvedor não ficará realizado diante de um código mal escrito por ele e nem por mais ninguém.

Da perspectiva da empresa, contratar profissionais que conheçam as técnicas diminuirá os bugs e aumentará a qualidade do código produzido. Os benefícios estendem-se para o financeiro, pois diante de um código melhor estruturado e de fácil absorção, a sua manutenção será mais eficiente e rápida. Outro ponto em decorrência disso serão clientes mais satisfeitos.

## Problema

De acordo com os cálculos de BIRD (2011), em um novo projeto estima-se que ao final do desenvolvimento com 50.000 linhas de código existam 750 bugs. Esse número aumenta na manutenção: avalia-se que em 50.000 linhas de código encontrem-se 1.080 bugs.

O tempo de familiarização com um sistema já desenvolvido é muito alto. Desenvolvedores que não participaram do projeto possuem grande dificuldade em entender o código, qual o problema ele tenta solucionar. Perdem muito tempo navegando pelo código, depurando-o a fim de encontrar alguma dica, um vislumbre, que lhes permita entender a total bagunça em que se encontram.

“Equipes que trabalharam rapidamente no início de um projeto podem perceber mais tarde que estão indo a passos de tartaruga” (Martin, 2008, p. 4). Nenhuma mudança é trivial, opta-se por remendos, o que piora cada vez mais o código.

Será que precisa ser sempre assim? Será que não existem melhores formas de escrever um código que facilite o entendimento dos desenvolvedores atuais e futuros?

.

## Objetivos

O objetivo desse estudo é apresentar as técnicas e boas práticas para se escrever um Código Limpo. Desejamos também mostrar, por meio de um experimento e análise estatística, a vantagem de se escrever um Código Limpo, em termos de tempo de manutenção e métricas de código-fonte, como quantidade de linhas de código.

1. FUNDAMENTAÇÃO TEÓRICA

## Código-fonte ruim custa caro

De acordo com Beck (2007, p. 11), “a maior parte do custo do software é constituída após a primeira vez que ele é implantado [...], portanto, se eu quero fazer meu código barato, eu deveria torna-lo fácil de ler”. Em outro trecho e pela mesma premissa, Beck (2007, p. 12) propõe que o programa deveria ser fácil de modificar também. Simples afirmações como essas nos remete à argumentação de que possuir um código compreensível e de fácil manutenção pode diminuir os dispêndios financeiros futuros.

Além disso, Martin (2008, p. 4) argumenta que, diante de um código confuso, a produtividade de uma equipe caí drasticamente com o passar do tempo, aproximando-se de zero. Com a redução da produtividade, a gerência adiciona mais membros na esperança de aumentar a eficiência da equipe, até que o código-fonte torne-se abominável e um replanejamento seja exigido. Ainda segundo Martin, esse grande replanejamento pode levar anos; os mesmos erros são cometidos, e depois de pronto, a bagunça é a mesma e a nova equipe demanda outro replanejamento.

Mas será que um código-fonte ruim influencia tanto assim o financeiro de uma empresa? De acordo com Martin (2008, p. 3) sim: “Foi o código ruim que acabou com a empresa”, em referência a um fato dos anos oitenta, em que uma empresa lançou um aplicativo extraordinário que se tornou muito popular. Depois de vários bugs não consertados nas novas versões, a empresa saiu do mercado. O motivo de tantos erros foi que o código estava um caos devido ao lançamento precipitado, e à medida que novos recursos eram adicionados o código piorava até que não era possível mais gerenciá-lo.

Diante do supracitado, um código ruim não pode ser negligenciado, visto que ele diminui a produtividade dos desenvolvedores, afetando a empresa financeiramente, podendo culminar em sua falência.

## Definição de Código Limpo

Antes de definirmos o que é um Código Limpo, vejamos o que é necessário para escrevê-lo segundo Martin (2008, p. 7):

Escrever um código limpo exige o uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de uma sensibilidade meticulosamente adquirida sobre “limpeza”. A “sensibilidade ao código” é o segredo. Alguns de nós já nascemos com ela. Outros precisam se esforçar para adquiri-la.

Em suma, um desenvolvedor que escreve Código Limpo é um artista ou por natureza ou por obstinação.

Mas afinal, o que é um Código Limpo? Podemos assegurar que sua definição não é tão simples. Existem variáveis subjetivas como a elegância, agradabilidade e legibilidade. Escrever Código Limpo é uma arte (Martin, 2008), portanto defini-lo segundo parâmetros lógicos e mensuráveis é complicado. Podemos considerar aspectos técnicos como testabilidade, índice de manutenibilidade e complexidade ciclomática, entretanto, temos que ter em mente que os aspectos subjetivos são tão importantes quanto.

Martin (2008) entrevistou renomados especialistas em desenvolvimento de software acerca da definição de Código Limpo. A partir disso, ele infere que ler um Código Limpo “deve fazer você sorrir”, deve ser tão bem legível quanto uma prosa bem escrita e deve ser cercado por testes. Outra característica notável é a afirmação de Michael Feathers, autor do livro Trabalhando Efetivamente com Código Legado: “um código limpo sempre parece que foi escrito por alguém que se importava”. Essa palavra “importar” tem um significado incisivo, e nos remete ao profissionalismo, visto que, ao não se importar, o desenvolvedor está agindo de forma não profissional.

## Nomes significativos

A maioria dos desenvolvedores carece de expressar a real intenção ao nomear uma variável, método ou classe, simplesmente porque escolher bons nomes requer boas habilidades de descrição (Martin, 2008, p. 30). Normalmente, o foco deles está em resolver o problema usando toda a expertise em lógica de programação e não em nomear adequadamente, e isso é um problema.

Nomear a fim de mostrar a real intenção é uma tarefa árdua, por isso, é necessário disciplina e ser destemido ao renomear as coisas. Não é por acaso que Fowler (2002, p. 22) diz que se o intuito é melhorar a clareza, renomear vale a pena. Não ter medo também é a chave, pois é difícil acertar o nome na primeira vez, porém, as IDEs atuais facilitam muito essa tarefa.

Fowler (2002, p. 22) ainda afirma que “um código que comunica o seu propósito é muito importante”. Evans (2004) compartilha o mesmo pensamento reconhecendo que um código bem escrito pode ser bastante comunicativo. Mas como conseguir uma boa comunicação através do código? Martin (2008) sugere usar nomes pronunciáveis que revelem a sua real intenção, não usar prefixos ou notação húngara[[1]](#footnote-1), evitar desinformação[[2]](#footnote-2), adicionar contexto significativo e usar nomes do domínio do problema.

## Quebrando o código em pedaços pequenos

Abrir um código-fonte e perceber que a barra de rolagem parece não ter fim é uma injeção de desânimo para os desenvolvedores. Em códigos como esse, o tempo gasto para entender e achar o ponto que precisa ser modificado é uma tarefa árdua e enfadonha. Outra característica de códigos assim é a duplicação e repetição de lógicas semelhantes ou até mesmo idênticas.

Para evitar esses tipos de códigos devemos seguir algumas regras. Martin (2008, p. 34) assegura que a primeira regra para se escrever métodos é que eles devem ser pequenos; e a segunda regra é que eles devem ser menores que isso. Outro ponto apontado por Martin (2008) é que métodos devem fazer uma coisa, e fazê-la bem.

Fowler (2002, p. 18) afirma que “pedaços menores de código tendem a fazerem as coisas mais manejáveis. Eles são fáceis de trabalhar”. Beck (2007, p. 78) segue o mesmo raciocínio ao relatar que um código é melhor lido quando quebrado em métodos relativamente pequenos.

A fim de obtermos classes com alta coesão e métodos enxutos é importante conhecermos também os princípios S.O.L.I.D.[[3]](#footnote-3), especialmente o Princípio da Responsabilidade Única (SRP), que declara que uma classe ou módulo deve ter uma, e apenas uma razão para mudar. Segundo Martin (2008, p. 139) o SRP é um dos mais importantes conceitos em orientação a objetos, um dos mais simples de entender, e apesar disso, parece ser um dos menos seguidos.

## Referência nula

Hoare (2009) relatou em uma conferência em Londres o que ele chama de um dos erros mais caros de sua carreira: “Eu chamo de meu erro de bilhões de dólares. Foi a invenção da referência nula em 1965”.

Quantos bugs não foram introduzidos por causa de uma referência nula? Até hoje, com todos os avançados compiladores e poderosas IDEs, os erros *NullPointerException* (Java) e *NullReferenceException* (C#) são ainda muito comuns.

Mas por que isso acontece? Normalmente porque o desenvolvedor ou passou nulo para um método ou retornou nulo, e não fez a devida checagem. Erros assim são comuns, mas podem ser evitados se o padrão Objeto Nulo for seguido. Martin (2006, p. 440) e Fowler (2002, p. 209) preferem o uso desse padrão ao invés de retornar ou passar referências nulas. Eles relatam que utilizando esse padrão nós podemos assegurar que os métodos sempre retornarão objetos válidos, mesmo quando eles falharem, evitando a necessidade de comparar por nulo.

## Testes unitários

Por que escrever testes é importante? Fowler (2002, p. 73) responde essa pergunta relatando como o desenvolvedor gasta seu tempo. Segundo Fowler, escrever código é uma pequena fração do tempo gasto. Na verdade, o desenvolvedor perde seu tempo tentando descobrir o que está acontecendo com aquele código, depurando para encontrar exatamente onde o erro está ocorrendo. Consertar o bug na maioria das vezes é simples, entretanto, descobrir onde ele está é um pesadelo. Por isso, a introdução de testes unitários é importante, pois auxilia muito na detecção de erros antes de eles acontecerem em produção, além de evitar o dispêndio com horas de depuração de programas.

Outro benefício advindo com os testes segundo Martin (2006, p. 71) é que eles servem como uma forma de documentação, isto é, basta olhar para o teste escrito para saber como uma funcionalidade foi implementada. Além disso, talvez o benefício mais importante que os testes proveem é o impacto na arquitetura e design da aplicação, pois para escrever um código testável requer que ele seja desacoplado e que não dependa de implementações concretas e sim de abstrações.

Todavia, não basta escrever testes a esmo. Eles devem ter um propósito e, acima de tudo, devem ser limpos. Martin (2009, p. 124) diz que código de teste é tão importante quanto código de produção e por isso devem ser tão limpos quanto. Testes limpos são aqueles que prezam pela legibilidade, simplicidade e clareza, por isso é interessante escrever testes que expressem em poucas linhas a complexidade do sistema que estão testando.

1. METODOLOGIA

Este capítulo aponta os métodos adotados e materiais usados na execução do trabalho. Há também um cronograma das atividades e os recursos utilizados.

## Tipo do trabalho

Este trabalho terá caráter exploratório aplicado.

## Coleta de dados

A amostra deste trabalho serão desenvolvedores na linguagem C# empregados em São José do Rio Preto, SP, com no mínimo um ano de experiência no mercado de trabalho.

Os dados serão coletados de fontes primárias, aplicando algumas métricas nos códigos elaborados pelos desenvolvedores.

## Desenvolvimento

As técnicas do Código Limpo serão utilizadas para resolver um problema proposto. O mesmo problema será resolvido sem as técnicas por outra pessoa. Cinco pessoas irão dar manutenção no Código Limpo para resolver outro problema proposto e outras cinco pessoas farão a mesma coisa no código sem as técnicas.

A ferramenta usada será o Visual Studio 2012, a linguagem de programação C#, usando o paradigma de orientação a objetos. Os códigos serão subidos no site <http://github.com/JobaDiniz/TCC>, que utiliza o GIT como controle de versão.

Dados como tempo gasto, número de linhas de código, complexidade ciclomática, índice de manutenibilidade e acoplamento de classes serão extraídos com o auxílio do Visual Studio 2012.

Os dados passaram por uma análise estatística usando o Teste de Hipótese T, que baseia-se nas médias dos dados.

## Cronograma

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Atividades | Jan | | | | Fev | | | | Mar | | | | Abr | | | | Mai | | | | Jun | | | |
| 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| Definição do tema |  |  |  |  |  | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Levantamento conteúdo teórico |  |  |  |  |  |  | X | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Anteprojeto |  |  |  |  |  |  |  |  | X | X | X |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Tabela 1 - Cronograma do trabalho.

REFERÊNCIAS

BECK, Kent. **Implementation Patterns**. Westford, Ma: Addison Wesley, 2007.

BIRD, Jim. **Bugs and Numbers**: How many bugs do you have in your code? Calgary, Canada, 24 ago. 2011. Disponível em: <http://swreflections.blogspot.com.br/2011/08/bugs-and-numbers-how-many-bugs-do-you.html>. Acesso em: 18 mar. 2013.

EVANS, Eric. **Domain Driven Design**: Tackling Complexity in Software. Westford, Ma: Addison Wesley, 2004.

FOWLER, Martin; BECK, Kent; BRANT, Jonh; OPDYKE, William; ROBERTS, Don. **Refactoring**: Improving the Design of Existing Code. Westford, Ma: Addison Wesley, 2002.

HOARE, Charles A. R. **Null References:** The Billion Dollar Mistake. Londres, Inglaterra. 2009. Disponível em: <http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake>. Acesso em: 28 abr. 2013.

MARTIN, Robert C.; MARTIN, Micah. **Agile Principles, Patterns, and Practices in C#**. Westford, Ma: Prentice Hall, 2006.

MARTIN, Robert C.. **Clean Code**: A Handbook of Agile Software Craftsmanship. Westford, Ma: Prentice Hall, 2009.

1. Prefixar ou sufixar ao nome o tipo que aquela variável representa. [↑](#footnote-ref-1)
2. Nomear variáveis com a letra L minúscula ou a letra O maiúscula. Ambas desinformam o leitor pois se assemelham aos números um e zero, respectivamente. [↑](#footnote-ref-2)
3. Acrônimo para Single Responsability Principle, Open-closed Principle, Liskov Substitution Principle, Interface Segregation Principle e Dependency Inversion Principle. [↑](#footnote-ref-3)