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

Joberto Diniz Junior[1], Djalma Domingos da Silva[1]

[1]FATEC – Faculdade de Tecnologia de São José do Rio Preto

me@jobertodiniz.com, djalma@fatecriopreto.edu.br

Abstract. *This paper discusses the Clean Code's science through some of its techniques, such as meaningful names, SOLID principles and unit testing, emphasizing through examples the importance of Clean Code's enforcement in order to obtain a robust system with few errors and high maintainability. Also, noteworthy is how much bad code can cost to companies and dramatically decrease developers' productivity. Using a small experiment, this study also statistically analyzes the Clean Code's advantages compared to a conventional code.*

Resumo. *Este trabalho aborda a ciência do Código Limpo através de algumas de suas técnicas, tais como nomes significativos, príncipios SOLID e testes unitários, enfatizando por meio de exemplos a importância da aplicação do Código Limpo com a finalidade de se obter um sistema robusto com poucos erros e alta manutenibilidade. Destaca-se ainda o quanto um código ruim pode custar às empresas e diminuir drasticamente a produtividade dos desenvolvedores. Recorrendo a um pequeno experimento, o referido estudo também analisa estatisticamente as vantagens do Código Limpo comparado a um código convencional.*

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 – *Integrated Development Enviroment*) 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 2009].

Pode-se citar dois grandes problemas que o código ruim apresenta. O primeiro são os *bugs*. 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 segundo está relacionado à baixa produtividade dos desenvolvedores. "Equipes que trabalharam rapidamente no início de um projeto podem perceber mais tarde que estão indo a passos de tartaruga" [Martin 2009]. Em um código ruim 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, que minimize os *bugs* e aumente a produtividade?

Este estudo tem por objetivo apresentar e colocar à prova por meio de um pequeno experimento 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 2009]. Ele deve expressar a sua real intenção, deve ser flexível, deve ser cuidado com responsabilidade [Martin 2009].

O que se segue é a justificativa desse trabalho, o preço que se paga por um código ruim, o que é um Código Limpo e quais são as suas técnicas, e este trabalho é concluído com a apresentação e discussão de um experimento.

1.1 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 2009] "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 contente 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 será clientes mais satisfeitos.

2. Fundamentação Teórica

2.1 Código-fonte ruim custa caro

De acordo com [Beck 2007], "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] propõe que o programa deveria ser fácil de modificar também. Simples afirmações

como essas remetem à argumentação de que possuir um código compreensível e de fácil manutenção pode diminuir os dispêndios financeiros futuros.

Além disso, [Martin 2009] argumenta que, diante de um código confuso, a produtividade de uma equipe cai 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 2009], 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 2009] 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 e introduz *bugs*, afetando a empresa financeiramente, podendo culminar em sua falência.

2.2 Definição de Código Limpo

Antes de definir o que é um Código Limpo, apresenta-se o que é necessário para escrevê-lo segundo [Martin 2009]:

> *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? Pode-se assegurar que sua definição não é tão simples. Existem variáveis subjetivas como a elegância, agradabilidade e legibilidade. Escrever Código Limpo é uma arte [Martin 2009], portanto defini-lo segundo parâmetros lógicos e mensuráveis é complicado. Pode-se considerar aspectos técnicos como testabilidade, índice de manutenibilidade e complexidade ciclomática, entretanto, deve-se ter em mente que os aspectos subjetivos são tão importantes quanto.

[Martin 2009] 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 [Feathers 2004 apud Martin 2009]: "um código limpo sempre parece que foi escrito por alguém que se importava". Essa palavra "importar" tem um significado incisivo, e

remete ao profissionalismo, visto que, ao não se importar, o desenvolvedor está agindo de forma não profissional.

2.3 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 2009]. 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] 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] 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 2009] sugere usar nomes pronunciáveis que revelem a sua real intenção, não usar prefixos ou notação húngara[1], evitar desinformação[2], adicionar contexto significativo e usar nomes do domínio do problema.

2.4 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, principalmente porque os desenvolvedores 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. Duplicações de lógicas semelhantes ou até mesmo idênticas são constantes.

Para evitar esse cenário deve-se seguir algumas regras propostas por [Martin 2009]. A primeira regra é que métodos devem ser pequenos; e a segunda regra é que eles devem ser menores que isso. Segundo [Martin 2009] um método pequeno caracteriza-se por no máximo cinco linhas. Além disso, métodos devem fazer uma única coisa, e devem fazê-la bem.

[Fowler 2002] afirma que "pedaços menores de código tendem a fazerem as coisas mais manejáveis. Eles são fáceis de trabalhar". [Beck 2007] segue o mesmo raciocínio ao relatar que um código é melhor lido quando quebrado em métodos relativamente pequenos.

---

[1] ***Notação Húngara***: prefixar ou sufixar ao nome o tipo que aquela variável representa.

[2] ***Exemplo Desinformação***: 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.

2.5 SOLID

A fim de se obter classes com alta coesão e métodos enxutos é importante conhecer também os princípios SOLID de programação orientada a objetos.

SOLID é um acrônimo para Princípio da Responsabilidade Única (SRP - *Single Responsability Principle*), Princípio Aberto-fechado (OCP - *Open-closed Principle*), Princípio da Substituição de Liskov (LSP - *Liskov Substitution Principle*), Princípio da Segregação de Interface (ISP - *Interface Segregation Principle*),e Princípio da Inversão de Dependência (DIP - *Dependency Inversion Principle*).

2.5.1 SRP

O Princípio da Responsabilidade Única declara que uma classe ou módulo deve ter uma, e apenas uma razão para mudar. Segundo [Martin 2009] 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. Percebe-se que esse princípio não está sendo seguido quando uma classe tem muitas responsabilidades, como mostrado na Figura 1.

```
public IEnumerable<Transmissao> ListarTransmissoes()
{
    try
    {
        var transmissoesXml = LerDoXml();
        var transmissoes = new List<Transmissao>();
        foreach (var transmissaoXml in transmissoesXml)
        {
            long espaco = 0;
            var dir = new DirectoryInfo(transmissaoXml.Pasta);
            if (!dir.Exists)
                espaco = 0;

            Parallel.ForEach(dir.EnumerateFiles("*.*", SearchOption.AllDirectories), file =>
            {
                espaco += file.Length;
            });

            var transmissao = new Transmissao { EspacoEmDiscoUsado = espaco };
            transmissoes.Add(transmissao);
        }

        return transmissoes;
    }
    catch (Exception ex)
    {
        var smtpClient = new SmtpClient();
        var email = new MailMessage("joba@joba.com", "erros@webcasters.com.br", "Erro", ex.ToString());

        smtpClient.Send(email);
    }
```

Figura 1. Método com muitas responsabilidades

O método ListarTransmissoes deveria apenas ler as transmissões de um arquivo *Xml*, porém ele está encarregado de outras responsabilidades como conversão de dados e envio de email. As Figuras 2 e 3 demonstram essas responsabilidades encapsuladas em outras classes (ControladorEmail, ConversorTransmisao, e ControladorDiscoRigido), tornando o método mais simples, legível e pequeno.

```
public IEnumerable<Transmissao> ListarTransmissoes()
{
    try
    {
        var conversor = new ConversorTransmissao();
        var transmissoesXml = LerDoXml();
        return conversor.Converter(transmissoesXml);
    }
    catch (Exception ex)
    {
        var controladorEmail = new ControladorEmail();
        controladorEmail.EnviarErro(ex);
    }
```

Figura 2. Método ListarTransmissoes refatorado

```
private Transmissao Converter(TransmissaoXml transmissaoXml)
{
    var controlador = new ControladorDiscoRigido();
    return new Transmissao
    {
        Codigo = transmissaoXml.CodigoTransmissao,
        Nome = transmissaoXml.NomeTransmissao,
        Pasta = transmissaoXml.Pasta,
        EspacoEmDiscoUsado = controlador.CalcularEspacoEmDiscoUsado(transmissaoXml.Pasta)
    };
}
```

Figura 3. Método Converter da classe ConversorTransmissao

2.5.2 OCP

De acordo com [Martin 2006], um sintoma de que o Princípio Aberto-fechado não está sendo seguido é quando uma simples mudança no programa resulta em uma mudança em cascata por toda parte dos módulos dependentes. Ao aplicar o OCP o sistema fica aberto para extensão mas fechado para modificação. O exemplo da Figura 4 mostra a rigidez do código, visto que se desejado adicionar o cálculo de mais um ano, o método terá que ser modificado.

```
public decimal Calcular(int ano, decimal salario)
{
    if (salario <= 0)
        return 0;
    else if (ano == 2010)
    {
        if (salario <= 1040.22M)
            return Math.Round(salario * 8 / 100, 2);
        else if (salario >= 1040.23M && salario <= 1733.70M)
            return Math.Round(salario * 9 / 100, 2);
        else if (salario >= 1733.71M && salario <= 3467.40M)
            return Math.Round(salario * 11 / 100, 2);
        else
            return 381.41M;
    }
    else if (ano == 2011)
    {
        if (salario <= 1106.90M)
            return Math.Round(salario * 8 / 100, 2);
        else if (salario >= 1106.91M && salario <= 1844.43M)
            return Math.Round(salario * 9 / 100, 2);
        else if (salario >= 1844.44M && salario <= 3689.66M)
            return Math.Round(salario * 11 / 100, 2);
        else
            return 405.86M;
```

Figura 4. Exemplo de rigidez no código

Aplicando o padrão de projeto Estratégia [Padrões de Projeto 2005] , pode-se refatorar o código para ficar em conformidade com o OCP, como mostrado na Figura 5. Dessa forma, para adicionar um novo ano, basta criar uma nova classe que implemente a interface ITabelaAliquota, não interferindo no código do método Calcular.

```
private IList<ITabelaAliquota> tabelas;

public CalculadorINSS()
{
    tabelas = new List<ITabelaAliquota>();
    tabelas.Add(new TabelaAliquota2010());
    tabelas.Add(new TabelaAliquota2011());
}

public decimal Calcular(int ano, decimal salario)
{
    foreach (var tabela in tabelas)
    {
        if (tabela.PodeCalcularParaOAno(ano))
            return tabela.CalcularDesconto(salario);
    }

    return decimal.Zero;
}
```

Figura 5. Refatoração para estar em conformidade com OCP

2.5.3 LSP

O Princípio de Substituição de Liskov afirma que subtipos devem ser substituíveis pelo seus tipos base, ou seja, está relacionado com regras de como usar herança corretamente. O princípio não está sendo seguido quando se faz necessário verificar se um determinado tipo base é um subtipo, como evidenciado na Figura 6.

```
public static void DobrarTamanho(Retangulo retangulo)
{
    if (retangulo is Quadrado)
        retangulo.Altura *= 2;
    else
    {
        retangulo.Altura *= 2;
        retangulo.Largura *= 2;
    }
}
```

Figura 6. Violação do LSP

A violação desse princípio nesse caso está em projetar um Quadrado como sendo um subtipo de um Retângulo. Este possui largura e altura, porém o Quadrado na verdade só se importa com o lado, então coisas estranhas podem acontecer se o método DobrarTamanho não verificar se a instância passada é um Quadrado. Como pode-se perceber, o Quadrado não é substituível pelo Retângulo, quebrando o LSP. Um melhor *design* para as classes citadas seria ambas herdarem de Forma.

2.5.4 ISP

O Princípio de Segregação de Interface, segundo [Martin 2006], lida com as desvantagens de interfaces gordas, isto é, os métodos da interface podem ser quebrados em grupos de métodos, cada grupo servindo diferentes clientes. Por exemplo, suponha as classes apresentadas nas Figuras 7 e 8:

```
class Contato
{
    public string Nome { get; set; }
    public string Endereco { get; set; }
    public string Email { get; set; }
    public string Telefone { get; set; }
}
```

Figura 7. Classe Contato

```
class ControladorEmail
{
    public void EnviarMensagem(Contato contato, string assunto, string corpo)
    {
        // Código para mandar o email usando o endereço de email e nome do contato.
    }
}
```

Figura 8. Classe ControladorEmail

O problema com essa solução é o fato de que o método EnviarMensagem conhece o Telefone e o Endereço do Contato sem necessidade, pois ele não se importa com esses dados. Permitir que a classe ControladorEmail tenha acesso a esses dados está violando o ISP. As figuras 9, 10 e 11 demonstram a refatoração do código citado para ficar em conformidade com o ISP.

```
interface IEmail
{
    string Nome { get; set; }
    string Email { get; set; }
}
```

Figura 9. Interface IEmail

```
class Contato : IEmail
{
    public string Nome { get; set; }
    public string Endereco { get; set; }
    public string Email { get; set; }
    public string Telefone { get; set; }
}
```

Figura 10. Classe Contato refatorada

```
public void EnviarMensagem(IEmail destino, string assunto, string corpo)
{
    // Código para mandar o email usando o endereço de email e nome da interface.
}
```

Figura 11. Método EnviarMensagem refatorado

Dessa maneira, o método EnviarMensagem conhece apenas o necessário para ele enviar um email.

2.5.5 DIP

O Princípio da Inversão de Dependência é importante para diminuir o acoplamento de classes [Martin 2006]. DIP destaca que módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Um exemplo clássico é módulos de interface com o usuário depender do módulos de acesso a dados. Outro exemplo, mais sutil, pode-se ver na refatoração do código exemplo do SRP (Figura 2 e 3). O método ListarTransmissoes depende explicitamente de ConversorTransmissao e ControladorEmail. Para respeitar o DIP, é necessário inverter essa dependência, introduzindo abstrações, como mostra a Figura 12:

```
class TransmissaoRepositorio
{
    private IConversorTransmissao conversor;
    private IControladorEmail controladorEmail;

    public TransmissaoRepositorio(IConversorTransmissao conversor,
        IControladorEmail controladorEmail)
    {
        this.conversor = conversor;
        this.controladorEmail = controladorEmail;
    }

    public IEnumerable<Transmissao> ListarTransmissoes()
    {
        try
        {
            var transmissoesXml = LerDoXml();
            return conversor.Converter(transmissoesXml);
        }
        catch (Exception ex)
        {
            controladorEmail.EnviarErro(ex);
        }
```

Figura 12. Classe TransmissaoRepositorio respeitando o DIP

2.6 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] e [Fowler 2002] preferem o uso desse padrão ao invés de retornar ou passar referências nulas. Eles relatam que utilizando esse padrão pode-se assegurar que os métodos sempre retornarão objetos válidos, mesmo quando eles falharem, evitando a necessidade de comparar por nulo.

Objeto Nulo é um objeto oco, sem funcionalidade, que substitui a referência nula, muito simples de ser implementado, como ilustrado na Figura 13:

```
public class Transmissao
{
    public static readonly Transmissao Vazia = new Transmissao();

    public int Codigo { get; set; }
    public string Nome { get; set; }
    public string Pasta { get; set; }
    public long EspacoEmDiscoUsado { get; set; }
}
```

Figura 13. Exemplo Objeto Nulo

Um exemplo de uso é mostrado na Figura 14. O método ObterPorCodigo retorna o Objeto Nulo ao invés de *null* caso não for encontrado a transmissão com o código especificado:

```
public Transmissao ObterPorCodigo(int codigo)
{
    var transmissao = LerDoXml().FirstOrDefault(t => t.Codigo == codigo);
    return transmissao ?? Transmissao.Vazia;
}
```

Figura 14. Uso do Objeto Nulo

2.7 Testes unitários

Por que escrever testes é importante? [Fowler 2002] 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] 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] 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.

3. Métodos

3.1 Amostra

Participaram do presente estudo 9 voluntários, residentes em São José do Rio Preto e São Paulo. Para serem incluídos no estudo os participantes deveriam possuir experiência profissional mínima de 1 ano em desenvolvimento de sistemas.

3.2 Experimento

Foi proposto a resolução de um problema de cálculo de desconto do Instituto Nacional do Seguro Social (INSS) para os anos 2010 e 2011 (Apêndice A) para 2 desenvolvedores, um que conhece as técnicas do Código Limpo e outro que não. Posteriormente, foi proposto para outros 7 desenvolvedores uma manutenção em um dos códigos anteriores para calcular o desconto dos anos 2012, 2013 e 2014 (Apêndice B), almejando simular um cenário de adição de novas funcionalidades em um sistema já pronto.

Com a resolução do segundo problema, pretende-se aferir características objetivas do Código Limpo, tais como índice de manutenibilidade, complexidade ciclomática, quantidade de linhas de código e tempo gasto. Essas métricas foram extraídas com o auxílio do *Visual Studio 2012*.

Também foi elaborado um pequeno questionário sobre alguns pontos das técnicas do Código Limpo a fim de aferir algumas características subjetivas.

3.3 Análise estatística

Os dados foram verificados quanto à normalidade da distribuição (teste de *Kolmogorov-Smirnov*), e apresentados em média e desvio-padrão. Parte da amostra foi dividida em 2 grupos, os que aplicaram manutenção no Código Limpo (n=4), e os que aplicaram no convencional (n=3). As comparações das médias de ambos os grupos, referentes as variáveis analisadas no estudo, foram realizadas por meio do teste *t* independente. Para analisar a associação entre as variáveis categóricas do estudo, provenientes do questionário, foi utilizado o Qui-Quadrado. Todas as análises foram realizadas por meio do pacote estatístico SPSS versão 13.0 com significância estabelecida em 5%.

4. Resultado e discussão

O referido estudo teve como objetivo comparar alguns parâmetros do Código Limpo em relação ao convencional, afim de testar sua eficácia.

4.1 Características objetivas

Dentre as características objetivas do Código Limpo, as que mais se destacaram foram o tempo gasto e o índice de manutenibilidade. Este último representa um valor entre 0 e 100 que indica o quão fácil é modificar o código; um valor maior significa uma melhor manutenibilidade.

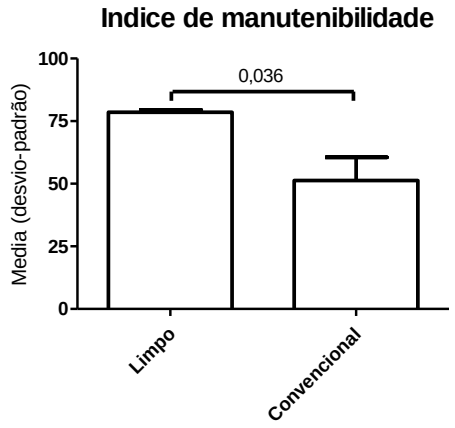

Figura 15. Valores do índice de manutenibilidade entre os voluntários do Código Limpo e convencional.

Estatisticamente, como mostra a Figura 15, observa-se que a hipótese nula foi rejeitada para essa métrica, visto que o valor-p (3,6%) é menor que 5%, o que indica que o índice de manutenibilidade do Código Limpo é superior ao convencional (78,50±0,28 vs 51,33±5,33). Isso revela que se tais técnicas forem seguidas na construção de um código, ele certamente será mais fácil de se modificar futuramente.

Diante de um alto índice de manutenibilidade, o tempo gasto para se modificar o código foi menor, como evidenciado na Figura 16.

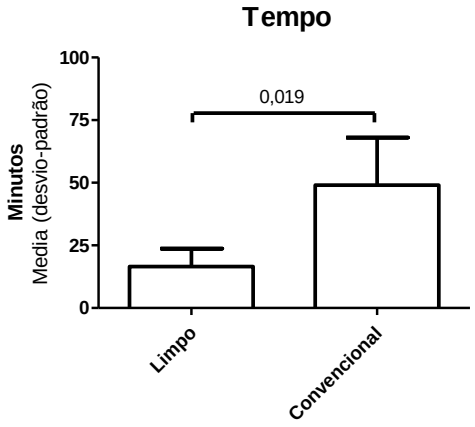

Figura 16. Valores do tempos (min) gasto para resolver o problema entre os voluntários do Código Limpo e convencional.

A média de tempo dispendido na execução da manutenção no código convencional foi mais que o dobro em relação ao Código Limpo (16,50±2,25 vs 49,00±10,97), revelando que o Código Limpo foi de fácil compreensão e estava estruturado de modo a fácil recepção de novas funcionalidades.

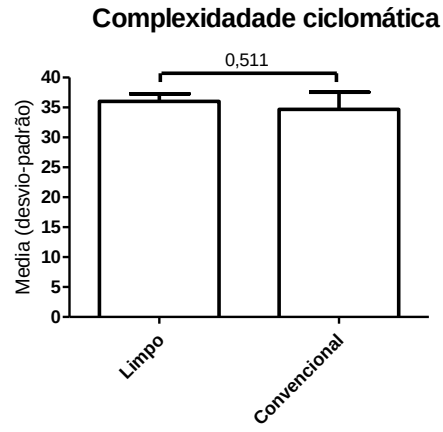

Figura 17. Valores da complexidade ciclomática entre os voluntários do Código Limpo e convencional.

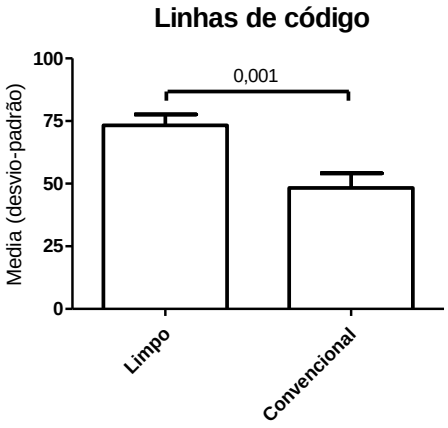

Figura 18. Quantidade de linhas utilizadas pelos voluntários do Código Limpo e convencional.

Essencialmente, complexidade ciclomática é usada para se obter um senso de quão difícil é testar, manter ou solucionar problemas de um código, além de indicar a probabilidade do código produzir erros. Valores menores que 10 são desejáveis segundo [Gustafson 2003]. Essa métrica, como mostrado na Figura 17, teve resultado praticamente igual nos dois tipos de códigos, ou seja, estatisticamente ela não teve significância.

Um ponto negativo do Código Limpo no experimento proposto foi a quantidade de linhas de código, que foi relativamente maior a do código convencional, como pode ser observado na Figura 18. Isso aconteceu pelo fato de que foi usado o padrão de projeto Estratégia, que segundo [Martin 2006] é flexível, mas envolve mais classes e mais código para ligar os objetos no sistema. Com esse padrão o código ficou mais adaptável, porém custou algumas linhas de código a mais, o que fundamentalmente não é ruim.

4.2 Características subjetivas

Um pequeno questionário foi elaborado para aferir pontos subjetivos do Código Limpo, aqueles que não são possíveis obter por meio de métricas. As questões que puderam ser analisadas estatisticamente são apresentadas na tabela 1, 2 e 3.

Tabela 1. Associação entre tipo de código e facilidade de leitura

Código	**A facilidade de leitura do código que recebeu foi:**			*p*
	Péssima	Boa	Excelente	
Limpo	0,0%	50,0%	50,0%	0,029
Convencional	100,0%	0,0%	0,0%	

Tabela 2. Associação entre tipo de código e facilidade para adicionar nova funcionalidade

Código	**A facilidade para adicionar a nova funcionalidade no código que recebeu foi:**			*p*
	Péssima	Boa	Excelente	
Limpo	0,0%	25,0%	75,0%	0,061
Convencional	33,3%	66,7%	0,0%	

Tabela 3. Associação entre tipo de código e tamanho das classes e métodos

Código	**Em media você achou o tamanho das classes e métodos:**			*p*
	Extensos	Médios	Pequenos	
Limpo	0,0%	25,0%	75,0%	0,039
Convencional	66,7%	33,3%	0,0%	

A totalidade dos indivíduos que aplicaram a manutenção no Código Limpo (100%) relataram que obtiveram boa e excelente facilidade de leitura do código, já os que aplicaram no código convencional informaram, em sua totalidade, que a leitura foi péssima, como mostra a Tabela 1. Analisando o código convencional em comparação ao Código Limpo, um dos motivos do código convencional receber péssima facilidade de leitura foi utilizar *if-else* encadeados, juntamente com números mágicos que não revelam sua real intenção, como mostrado na Figura 19. Em contrapartida, o Código Limpo introduziu o conceito de faixas para representar os valores de intervalos de salários das tabelas do INSS, como apresentado na Figura 20. Com isso, foi possível mostrar a real intenção do código, usando um conceito do domínio do problema, permitindo fácil leitura e entendimento.

```
else if (ano == 2010)
{
    if (salario <= 1040.22M)
        return Math.Round(salario * 8 / 100, 2);
    else if (salario >= 1040.23M && salario <= 1733.70M)
        return Math.Round(salario * 9 / 100, 2);
    else if (salario >= 1733.71M && salario <= 3467.40M)
        return Math.Round(salario * 11 / 100, 2);
    else
        return 381.41M;
}
else if (ano == 2011)
{
    if (salario <= 1106.90M)
        return Math.Round(salario * 8 / 100, 2);
    else if (salario >= 1106.91M && salario <= 1844.43M)
```

Figura 19. Código convencional extraído de um dos voluntários

```
public TabelaAliquota2010()
{
    Faixas.Add(new Faixa(1040.22M, Aliquotas.OitoPorcento));
    Faixas.Add(new Faixa(1040.23M, 1733.70M, Aliquotas.NovePorcento));
    Faixas.Add(new Faixa(1733.71M, 3467.40M, Aliquotas.OnzePorcento));
```

Figura 20. Faixas extraídas do Código Limpo

Analisando a Tabela 2, apesar de a maioria que aplicou manutenção no Código Limpo ter achado excelente a facilidade de adicionar a nova funcionalidade, estatisticamente não houve diferença em relação ao código convencional.

Embora a média do número de linhas do Código Limpo ter sido maior que a do código convencional, como mostrado na Figura 18, a Tabela 3 demonstra que as classes e métodos estavam pequenos, o que pode ser um indício de que o número final de linhas não importe tanto assim na construção de um Código Limpo.

O Gráfico 1 realça que todos os voluntários, seja do grupo do Código Limpo ou do código convencional, sentem-se seguros para alterar um código coberto por testes unitários. Não obstante, o grupo do código convencional não criou nenhum tipo de teste para o código produzido, mostrando que às vezes as técnicas não são feitas por simples displicência ou preguiça.

Gráfico 1. Porcentagens sobre sentimento de segurança com código coberto de testes

Um ponto controvérsio em desenvolvimento de sistemas, abordado pela pergunta do Gráfico 2, é a ação de comentar ou não o código produzido.

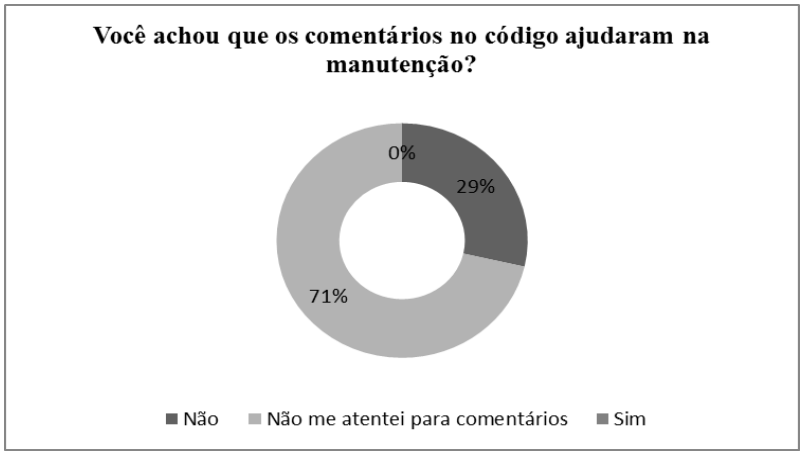

Gráfico 2. Porcentagem sobre se os comentários ajudaram na manutenção

O Gráfico 2 mostra que mais da metade dos voluntários nem sequer notaram a presença de comentários no código. Mas será que comentários realmente ajudam a entender o código? [Martin 2009] argumenta que às vezes é um mal necessário, entretanto devem ser evitados ao máximo, pois eles normalmente não são atualizados quando mudanças no código são feitas, podendo enganar futuros desenvolvedores. [Martin 2009] propõe sempre preferir a expressividade pelo código ao invés de por comentários quando possível; e a maioria das vezes é possível.

5. Conclusão

Apesar da baixa amostra e dos problemas propostos serem de baixíssima complexidade, em síntese, os resultados observados na presente investigação sugerem que as técnicas do Código Limpo, quando aplicadas disciplinadamente, podem aumentar a produtividade dos desenvolvedores, visto que o índice de manutenibilidade, a legibilidade e o tempo de manutenção são melhores que de um código convencional. Dessa forma, a importância do Código Limpo é notória, e as empresas de *software* podem investir em estratégias que visem o treinamento e conscientização dos seus desenvolvedores para obterem um produto de qualidade, com baixos índices de erros e alta extensibilidade.

6. Referências

Beck K. Implementation Patterns. Westford (MA): Addison Wesley; 2007.

Bird J. Bugs and Numbers: How many bugs do you have in your code? [base de dados na internet]. Calgary (Canada). [atualizada em 2011 Ago 24; acesso em 2013 Mar 18]. Disponível em: http://swreflections.blogspot.com.br/2011/08/bugs-and-numbers-how-many-bugs-do-you.html.

Evans E. Domain Driven Design: Tackling Complexity in Software. Westford (MA): Addison Wesley; 2004.

Feathers M apud Martin RC. Working Effectively with Legacy Code. Prentice Hall; 2004.

Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. Westford (MA): Addison Wesley; 2002.

Gustafson DA. Engenharia de Software. São Paulo (SP): Bookman; 2003.

Hoare CA. Null References: The Billion Dollar Mistake. [base de dados na internet]. Londres (Inglaterra). [atualizada em 2009; aceso em 2013 Abr 28]. Disponível em: http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake.

Martin RC, Martin M. Agile Principles, Patterns, and Practices in C#. Westford (MA): Prentice Hall; 2006.

Martin RC. Clean Code: A Handbook of Agile Software Craftsmanship. Westford (MA): Prentice Hall; 2009.

Vlissides J, Helm R, Gamma R, Johnson R. Padrões de Projeto. São Paulo (SP): Bookman; 2005.

Apêndice A

Uma empresa precisa calcular o desconto do INSS sobre o salário dos seus empregados dos anos 2010 e 2011. A sua tarefa é criar uma rotina para calcular esse desconto. Para isso, você deve criar uma classe que implemente uma interface pré-estabelecida chamada *ICalculadorINSS*, que possui um único método chamado *Calcular*, que recebe dois parâmetros, o ano e o salário, respectivamente, e retorna o valor do desconto. O projeto do *Visual Studio* incluindo essa interface será entregue a você. Atente-se que você não está restrito somente a essa classe; outras podem ser criadas, mas apenas uma que implemente a interface dada.

O desconto do INSS segue uma tabela anual que contém alíquotas para determinadas faixas de salário. O valor do desconto é igual à porcentagem da alíquota sobre o salário. Entretanto, quando o empregado tiver um salário superior ao limite máximo da tabela, o desconto é um valor pré-estabelecido, chamado de teto.

Abaixo segue as tabelas de 2010 e 2011, respectivamente.

2010	
Salário-de-contribuição (R$)	**Alíquota para fins de recolhimento ao INSS (%)**
até R$ 1.040,22	8,00
de R$ 1.040,23 a R$ 1.733,70	9,00
de R$ 1.733,71 até R$ 3.467,40	11,00
Teto	**381,41**

2011	
Salário-de-contribuição (R$)	**Alíquota para fins de recolhimento ao INSS (%)**
até R$ 1.106,90	8,00
de R$ 1.106,91 a R$ 1.844,83	9,00
de R$ 1.844,84 até R$ 3.689,66	11,00
Teto	**405,86**

Para melhor entendimento, vejamos dois exemplos se o ano for 2011:

1. Se o salário de João for R$ 1.000,00, o desconto do INSS será R$ 1.000,00 multiplicado por 8%, resultando em R$ 80,00.
2. Se o salário de Maria for R$ 4.000,00, o desconto do INSS será o Teto R$ 405,86.

Além disso, a empresa relatou que pode haver lixo em sua base de dados e por isso é importante validar os parâmetros de entrada. Para isso, ela disponibilizou a seguinte tabela:

Parâmetro	Condição	Desconto INSS
Salário	<= 0	0
Ano	!= 2010 e 2011	0

Apêndice B

Uma empresa precisa calcular o desconto do INSS sobre o salário dos seus empregados dos anos 2012, 2013 e 2014. Ela já possui um código pronto para o cálculo dos anos 2010 e 2011. A sua tarefa é acrescentar o cálculo desses novos três anos. Usando o projeto dado, adicione essa nova funcionalidade. Não é necessário ficar restrito as classes já existentes.

O desconto do INSS segue uma tabela anual que contém alíquotas para determinadas faixas de salário. O valor do desconto é igual à porcentagem da alíquota sobre o salário. Entretanto, quando o empregado tiver um salário superior ao limite máximo da tabela, o desconto é um valor pré-estabelecido, chamado de teto.

Abaixo segue as tabelas de 2012, 2013 e 2014 (tabela fictícia), respectivamente.

2012	
Salário-de-contribuição (R$)	**Alíquota para fins de recolhimento ao INSS (%)**
até 1.174,86	8,00
de 1.174,87 até 1.958,10	9,00
de 1.958,11 até 3.916,20	11,00
Teto	**430,78**

2013	
Salário-de-contribuição (R$)	**Alíquota para fins de recolhimento ao INSS (%)**
até 1.247,70	8,00
de 1.247,71 até 2.079,50	9,00
de 2.079,51 até 4.159,00	11,00
Teto	**457,49**

2014	
Salário-de-contribuição (R$)	**Alíquota para fins de recolhimento ao INSS (%)**
até 1.300,00	8,00
de 1.300,01 até 2.300,00	9,00
de 2.300,01 até 4.300,00	12,00
Teto	**516,00**

Para melhor entendimento, vejamos dois exemplos se o ano for 2012:

1. Se o salário de João for R$ 1.000,00, o desconto do INSS será R$ 1.000,00 multiplicado por 8%, resultando em R$ 80,00.
2. Se o salário de Maria for R$ 4.000,00, o desconto do INSS será o Teto R$ 430,78.

Além disso, a empresa relatou que pode haver lixo em sua base de dados e por isso é importante validar os parâmetros de entrada. Para isso, ela disponibilizou a seguinte tabela:

Parâmetro	Condição	Desconto INSS
Salário	<= 0	0
Ano	!= 2010, 2011, 2012, 2013 e 2014	0