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

***Djalma Domingos da Silva, Joberto Diniz Junior***

FATEC - Faculdade de Tecnologia de São José do Rio Preto

me@jobertodiniz.com

**1. Introdução**

Desde os anos noventa uma tarefa não parece ter mudado muito no desenvolvimento de sistemas: 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]. Dois grandes problemas emergem do código ruim: *bugs*, que segundo [Bird 2011] em um projeto de manutenção de 50.000 linhas, encontram-se 1.080 *bugs*; o outro problema é a baixa produtividade dos desenvolvedores.

Mas 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?

**2. Objetivos**

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 se expressa 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].

**3. Fundamentação Teórica**

**3.1 Código 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”. 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. 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, mas depois de muitos *bugs* a empresa saiu do mercado.

**3.2 Definição de Código Limpo**

Segundo [Martin 2009], Código Limpo é uma arte produzida por uma miríade de pequenas técnicas aplicadas disciplinadamente. Definir Código Limpo por 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 como agradabilidade, elegância e legibilidade são tão importantes quanto.

**3.3 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.

[Fowler 2002] 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, evitar desinformação, adicionar contexto significativo e usar nomes do domínio do problema.

**3.4 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.

Para evitar esse tipo de cenário, [Martin 2009] sugere que métodos devem ser pequenos, com no máximo cinco linhas, e ter apenas uma única responsabilidade. [Beck 2007] compartilha do mesmo pensamento ao relatar que um código é melhor lido quando quebrado em métodos relativamente pequenos.

**3.5 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), Princípio Aberto-fechado (OCP), Princípio da Substituição de Liskov (LSP), Princípio da Segregação de Interface (ISP), e Princípio da Inversão de Dependência (DIP).

Esses princípios quando aplicados juntos tendem a construir um sistema que é fácil de manter e evoluir com o tempo.

Em síntese, o SRP afirma que uma classe deve ter uma, e apenas uma, razão para mudar. O OCP diz que deve ser possível estender o comportamento de uma classe sem modificá-la. O LSP aborda o conceito de herança, em que classes derivadas devem ser substituíveis por suas classes bases. O ISP diz para criar interfaces enxutas para cada cliente específico. Já o DIP relata para depender de abstrações e não de implementações concretas.

**3.6 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 poderosos Ambientes de Desenvolvimento Integrado (IDE – *Integrated Development Enviroment*), 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. Para evitar isso, pode-se usar o padrão de projeto Objeto Nulo, isto é, um objeto oco, sem funcionalidade, que substitui a referência nula.

**4. Métodos**

**4.1 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.

**4.2 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 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, almejando simular um cenário de adição de novas funcionalidades em um sistema já pronto.

Os resultados, como índice de manutenibilidade e tempo gasto, foram analisados estatisticamente por meio do teste *t* independente.

**5. Resultados**

O índice de manutenibilidade representa um valor de 0 a 100 que indica o quão fácil é modificar o código; um valor maior significa uma melhor manutenibilidade.

Estatisticamente, como mostra a Figura 1, observa-se que a hipótese nula foi rejeitada para essa métrica, visto que o valor-p (3,6%) é menor que 5%, revelando que se as técnicas do Código Limpo forem seguidas na construção de um código, ele certamente será mais fácil de se modificar.



Figura 1: Valores do índice de manutenibilidade

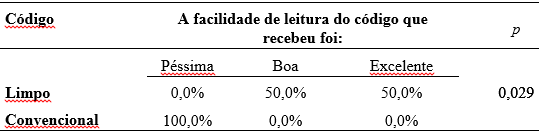
Diante desse alto índice, o tempo gasto para modificar o Código Limpo (Figura 2) foi menor (16,50±2,25 vs 49,00±10,97), o que mostra que o código estava de fácil compreensão e bem estruturado para receber novas funcionalidades.



Figura 1: Valores do tempo gasto

A Tabela 1 mostra o resultado de uma pergunta do questionário, demonstrando que para os indivíduos que participaram do experimento, o Código Limpo estava mais fácil de ler.

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



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 Em contrapartida, o Código Limpo introduziu o conceito de faixas para representar os valores de intervalos de salários das tabelas do INSS. Usando um conceito do domínio do problema, foi possível mostrar a real intenção do código.

**6. Conclusões**

Os resultados sugerem que as técnicas do Código Limpo podem aumentar a produtividade dos desenvolvedores, visto que o índice de manutenibilidade e o tempo de manutenção são melhores que de um código convencional. A importância do Código Limpo é notória, e as empresas de software podem investir em estratégias que visem o treinamento de seus desenvolvedores.

**7. Referências Bibliográficas**

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.

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

Hoare C. 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 R. Clean Code: A Handbook of Agile Software Craftsmanship. Westford (MA): Prentice Hall; 2009.