

# Maturidade no processo de desenvolvimento de software

## Qualidade de código contínua

João Carlos Brasileiro Stefenon de Almeida<sup>1</sup>

<sup>1</sup> Universidade do Vale do Rio dos Sinos (UNISINOS)  
São Leopoldo – RS – Brazil

`jcbbrasileiro@hotmail.com`

*Abstract.* [WIP]

*Resumo.* [WIP]

## Part I

# INTRODUÇÃO

Uma das mudanças mais impactantes no processo de desenvolvimento de software das últimas décadas, foi a concepção e absorção das metodologias ágeis de software, seja SCRUM, Extreme Programming (XP), Test Driven Development (TDD), Lean Software Development, Kanban, etc, todas foram criadas a partir da filosofia AGILE [Agile 2001], surgido a partir do esforço de várias pessoas que lidavam com o processo de software na década de 1990, com o objetivo de definir uma abordagem mais efetiva e eficiente para o desenvolvimento de software. Apesar dos conceitos já existirem a quase 20 anos, a importância e a implementação dessas ideias ainda são muito pouco exploradas no Brasil, segundo Coleman Parkes Research [Research and Technologies 2017], durante uma pesquisa, envolvendo cerca de 1.770 executivos de tecnologia da informação em 21 países, incluindo 76 brasileiros, mostra que somente 6% das empresas têm utilizado fortemente essa filosofia com o objetivo de "transformar toda a organização para abranger os princípios de agilidade".

A partir dessa nova abordagem de gerenciar software, a importância da qualidade evoluiu e alcançou novos pontos de vistas, aumentando significativamente tanto na concepção, quanto no decorrer das outras fases, incluindo durante o desenvolvimento, e não mais apenas no final com um simples objetivo de execução do software. Um das grandes mudanças foi a inclusão e a utilização do conceito de **MPV**, sigla de *Minimum Viable Product*, que significa Produto Mínimo Viável – conceito popularizado por Eric Ries [Ries 2011] - trazendo a importância de entregar e assegurar o sistema de modo incremental, incluindo progressivamente funcionalidades importantes para o cliente, ou o negócio, ressaltando não apenas garantir um software útil, mas também visando minimizar a maleabilidade do ecossistema do negócio, seja as possibilidades de mudanças, tanto de prioridades de funções, técnicas ou mesmo do próprio objetivo fim. A adoção desses pequenos entregáveis, gerou e exigiu um aumento na maturidade no desenvolvimento, criando a necessidade de gerar artefatos que evidenciassem e garantissem, desde

início, dois grandes aspectos: (i) Qualidade e (ii) a garantia que cada entregáveis operasse corretamente.

Max Kanat-Alexander [Max Kanat 2012] resume sinteticamente uma interpretação da importância da qualidade do código/design:

*”É mais importante reduzir o esforço de manutenção do que reduzir o esforço de implementação”.*

Também reforça as consequências de ignorar o fato de existir um futuro, e cair no erro de criamos coisas que ”apenas funcionam no presente”, a partir disso menciona uma regra importante para a qualidade:

*”O nível de qualidade do seu projeto deve ser proporcional ao tempo do futuro em que seu sistema continuará a ajudar as pessoas”.*

Steve McConnell [McConnell 2004] deixa claro a ideia de qualidade de um código limpo, não é uma parte opcional no desenvolvimento, reforçando a importância dos aspectos (capacidade de extensão, código testável, fácil compreensão, alta coesão, desacoplamento, entre outros).

A partir dessas novas perspectivas e preocupações, e visando amadurecer a qualidade continua no processo de desenvolvimento foi identificado um grande problema na quantidade de esforço para manutenção de códigos com baixa extensibilidade e alto acoplamento, junto com uma alta taxa de “code smells”, termo criado por Kent Beck em conjunto de Martin Fowler [Fowler et al. 1999], e níveis muito baixos de qualidade de design/código.

Acreditando na importância da qualidade do código, o presente trabalho visou desenvolver, aplicar e avaliar uma estratégia para minimizar as adversidades na fase de um desenvolvimento eficiente, obter uma melhor produtividade através da codificação de códigos com alta manutenibilidade e alta confiabilidade, desacoplados e coeso, aplicando as ideias do CLEAN CODE, em conjunto com os mais difundidos princípios da programação orientada a objeto.

A finalidade desse trabalho foi aplicar os conceitos do OODP (Object Oriented Design Principles), CLEAN CODE e “AGILE”, sendo assim, o objetivo geral foi analisar uma estratégia para produção de códigos com qualidade. A partir disso para alcançar o objetivo geral foram definidos os seguintes objetivos específicos:

- Identificar a correta aplicação dos conceitos de OODP/CLEAN CODE.
- Apontar uma prática eficiente de codificação.
- Definir métricas qualitativas e quantitativas para monitoramento da qualidade de código.

O presente trabalho justifica-se por uma forte necessidade de ressaltar ao desenvolvedor, a responsabilidade direta pela produção do artefato final da fase de desenvolvimento (código-fonte), não apenas em relação ao presente, mas também quanto ao seu futuro, maximizar a rapidez de entendimento do código, ou sua possível extensão, evidenciar a importância e a necessidade de conhecer, estudar e aplicar os conceitos de OODP, e/ou do CLEAN CODE, que por sua vez, trouxe uma mudança de paradigma de como codificar software, como assegurar, minimizar e identificar impactos decorrentes de mudanças. Também visa, da perspectiva da empresa, evidenciando que diante um código

com qualidade, código melhor estruturado, bem testado, fácil absorção e entendimento, a sua manutenção será mais eficiente e rápida, fazendo do desenvolvimento mais eficaz e eficiente, assim, gerando mais produtividade.

O plano de pesquisa desse trabalho utiliza uma abordagem mista utilizando de métricas quantitativas e qualitativas, de natureza aplicada, afim de determinar uma estratégia para maximizar a qualidade de código, explorando técnicas, conceitos e praticas. Utilizando do método de pesquisa estudo de caso, através de cenários, reais, baseados em cenários, ou simulação do problema, para analisar os resultados objetivos a partir.

O presente trabalho foi estruturado em 7 capítulos. O capítulo 2 apresenta os trabalhos relacionados. O capítulo 3 apresenta a revisão bibliografia. O capítulo 4 apresenta a Metodologia de Pesquisa utilizada. O capítulo 5,apresenta a avaliação da solução proposta. O capítulo 6,apresenta os trabalhos futuros. E, o capítulo 7, apresenta a conclusão deste trabalho

## **Part II**

# **Trabalhos Relacionados**

Com o objetivo de entender e levantar os diversos problema de design e manutenabilidade de código, foram analisadas diversas inciativas e estudos relacionados que utilizam ou exploram os conceito e objetivam a qualidade de código referente a esse trabalho. Com isso, os principais trabalhos analisados foram: (i) trabalho de [Thomaz Almeida and Machini de Miranda 2015] que explora os conceitos do CLEAN CODE, explicar e apresenta com mais detalhes algumas técnicas a serem utilizadas.

(ii) [Diniz Junior and Domingos da Silva 2015] enfatiza 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. Destacando ainda o quanto um código ruim pode custar às empresas e diminuir drasticamente a produtividade dos desenvolvedores através de um pequeno experimento, por fim, analisa estatisticamente as vantagens do código limpo comparado a um código convencional, concluindo que a partir dos resultados observados, os mesmos, sugerem que as técnicas, quando aplicadas disciplinadamente, podem aumentar a produtividade dos desenvolvedores, visto que o índice de manutenibilidade, alegibilidade e o tempo de manutenção são melhores.

() [Niralem et al. 2017] fala sobre a dívida técnica, se referir a qualquer projeto de sistema, arquitetura, desenvolvimento dentro da base de código, é uma solução de curto prazo para qualquer trabalho específico, que é aplicado antes da solução completa ou adequada para qualquer trabalho, como dizendo que não é uma solução a longo prazo para qualquer trabalho específico. É uma espécie de solução que é encaminhado pelos não especialistas para o conclusão ou entrega do produto, mas é atraídos pelos especialistas que podem comprometer a qualidade do produto.

(iii) [Almeida and de Miranda 2010] um mapeamento entre um conjunto de métricas de código-fonte, com o objetivo de facilitar a detecção de trechos de código

com potencial de melhorias, apresenta uma maneira de interpretar as métricas.

(iv) [Sedano 2016] demonstra como o teste de legibilidade do código melhora a capacidade dos programadores para escrever código legível, e identificar correções. Apresenta uma comparação de técnicas e conclui com resultados positivos, relatando que as técnicas valem seu tempo investido e articula como os testes podem alterar positivamente seus hábitos de programação.

(v) [Yamashita and Moonen 2012] esse artigo apresenta uma importante análise sobre "code smell", sendo apresentado um relatório sobre um estudo empírico que investiga a extensão que os "code smell" refletem e afetam a capacidade de manutenção.

## **Part III**

# **REVISÃO BIBLIOGRÁFICA**

### **1. *Clean Code***

[WIP]

Learning to write clean code is hard work. It requires more than just the knowledge of principles and patterns. You must sweat over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace their steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way.

### **2. *Object-Oriented Design Principles (OODP)***

[WIP]

#### **2.1. *S.O.L.I.D principles***

- Single responsibility principle
- Open closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency Inversion principle

[WIP]

#### **2.2. *General Responsibility Assignment Software Principles (GRASP)***

[WIP]

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling

- Polymorphism
- Protected Variations
- Pure Fabrication

[WIP]

### **2.3. *Favor Composition over Inheritance.***

[WIP]

### **2.4. *Law of Demeter principles***

[WIP]

### **2.5. *Don't Repeat Yourself (DRY)***

[WIP]

### **2.6. *Keep it Simple, Stupid. (KISS)***

[WIP]

## **3. *Test-driven development (TDD)***

[WIP] Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

The process can be defined as such:

1. Write a failing unit test
2. Make the unit test pass
3. Refactor

Repeat this process for every feature, as is necessary.

## **Part IV**

# **Conclusão**

[WIP]

## **Part V**

# **Trabalhos futuros**

[WIP]

## References

- Agile (2001). Manifesto for agile software development. <http://agilemanifesto.org/>.
- Almeida, L. T. and de Miranda, J. M. (2010). Código limpo e seu mapeamento para métricas de código fonte. *Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)*.
- Diniz Junior, J. and Domingos da Silva, D. (2015). A importância do código limpo na perspectiva dos desenvolvedores e empresas de software. *USP Digital*.
- Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (July 8, 1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, New York.
- Max Kanat, A. (2012). *Code Simplicity*. O'Reilly Media.
- McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2th edition.
- Niralem, S., Kawati, V., and G.R, S. (2017). Quality code: Eliminating technical debt. *Imperial Journal of Interdisciplinary Research (IJIR)*.
- Research, C. P. and Technologies, C. (2017). Accelerating velocity and customer value with agile and devops. Technical report, CA Technologies.
- Ries, E. (c2011.). *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, New York, 1st ed. edition.
- Sedano, T. (2016). Code readability testing, an empirical study. *Carnegie Mellon University - Silicon Valley Campus*.
- Thomaz Almeida, L. and Machini de Miranda, J. (2015). Visão introdutória sobre os conceitos de código limpo. *Revista semana acadêmica*.
- Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects?. *28th IEEE International Conference on Software Maintenance (ICSM)*.