Maturidade no processo de desenvolvimento de software Qualidade de código contínua

João Carlos Brasileiro Stefenon de Almeida[1]

[1] Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo – RS – Brazil

jcbrasileiro@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 exigi um aumento na maturidade no desenvolvimento, criando a necessidade de gerar artefatos que evidenciassem e garantissem, desdo

inicio, 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, sendo uma parte principal, e não uma parte opicional no desenvolvimento, reforçando a importancia dos aspectos de um código bem feito (capacidade de extensão, código testavel, fácil comprensão, alta coesão, desacoplamento, entre outros).

A partir dessas novas perpectivas 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 conjuto 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 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 alçancar o objetivo geral foram definidos os seguindos objetivos especificos:

- Identificar a correta aplicação dos conceitos de OOPD/CLEAN CODE.
- Apontar uma pratica eficiente de códificação.
- 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 possivel extensão, evidenciar a importância e a necessidade de conhecer, estudar e aplicar os conceitos de OOPD, 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 cenarios, ou simulção 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 tecnicas 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 analisa 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 BIBLIOGRAFICA

1. *Clean Code*

”Clean code” [Martin 2008] é um conceito subjetivo, que de modo muito generico, significa código bem feito, incluindo saber transformar um ”código ruim” em um ”código bom”, ou o conhecimento da diferença entre ambos, assim como escrever um ”código bom”. Robert C. Martin [Martin 2008] debate os conceitos do que é ”certo” e o que é ”errado” na codificação. Destacando pontos impotantes para alcançar o nível de qualidade como principios, padrões, e boas práticas, passando por estudos de casos de diversos tipos e complexidades, uma base de conhecimento que descreve e explicar o racícinio usado durante a leitura, a escrita e durante a aplicação do ”clean code”, como a utilização de nomes significativos, escritas de funções pequenas e com objetivos claros, The Stepdown Rule [1], formatação de código, importancia de um sistema testavel [2], entre outros pontos.

Também destaca o esforço necessario para absorver o conhecimento, e conhecer o ”clean code” [Martin 2008]:

> *”Aprender a escrever um código limpo é um trabalho árduo. Exige mais do que apenas o conhecimento de princípios e padrões. Você deve suar sobre ele. Você deve praticá-lo você mesmo, e assistir você falha. Você deve observar os outros praticando e falhar. Você deve vê-los tropeçar e siga seus passos. Você deve vê-los agonizantes sobre as decisões e ver o preço que pagam fazendo essas decisões do jeito errado.”*

[1] The Stepdown Rule [Martin 2008], um código deve ser escrito de modo ”TOPDOWN”, aonde o programa fosse um conjunto des parágrafos ”TO-DO”, cada um descrevendo o nível atual de abstração e referenciando subseqüentemente outro parágrafos ”TO-DO” no próximo nível de abstração abaixo.

[2] Um sistema que é testado de forma abrangente e passa todos os seus testes o tempo todo é um sistema testável. Essa é uma declaração óbvia, mas importante. Sistemas que não são testáveis não são verificáveis. Provavelmente, um sistema que não pode ser verificado nunca deve ser implantado

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

Os principios de design orientado a objeto, tem como objetivo ajudar os desenvolvedores a eliminar ”design smell” [3] e construir melhores soluções para o atual problema/”feature”.

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

Conceitos apresentendo iniciamente em 2002 [Martin 2002], como parte do processo ”agile design”:

> *” É a aplicação contínua de princípios, padrões e práticas para melhorar a estrutura e a legibilidade do software. É a dedicação de permanecer o design do sistema tão simples, limpa e expressiva quanto possivel a todo hora. ”*

Posteriormente, em 2004, estes cinco princípios tornou conhecido pelo acrônimo ”SOLID”, após Michael Feathers reorganizar as sequencia dos itens.

- **SRP** - Single responsibility principle
- **OCP** - Open closed principle
- **LSP** - Liskov substitution principle
- **ISP** - Interface segregation principle
- **DIP** - Dependency Inversion principle

2.1.1. SRP - Single responsibility principle

> *”A classe deve ter apenas uma responsabilidade.”*

2.1.2. OCP - Open closed principle

> *”Entidades de software (classes, modulos, fuções..) devem estar abertas para extensão, mas fechadas para modificação.”*

2.1.3. LSP - Liskov substitution principle

> *”Subtipos [4] devem ser substituíveis por suas classes base.”*[5]

2.1.4. ISP - Interface segregation principle

> *”Muitas interfaces específicas são melhores do que uma interface com propósito generico.”*

[3] ”Desing smell” [Martin 2002] é um sintoma, algo mensuravel, subjetivamente se não objetivamanete, que normalmente, é o resultado de uma ou mais violações aos principios.

[4] Também conhecidas como ”Classes derivadas”.

[5] Barbara Liskov escreveu em seu primeiro artigo em 1988, *”What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for allprograms Pdefined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.”*

2.1.5. DIP - Dependency Inversion principle

> *A - Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.*
> *B - Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.*

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.*

> *"Uma relação "HAS-A" pode ser melhor do que "IS-A"."*[Sierra et al. 2004]

Criar sistemas usando a composição permite muito mais flexibilidade, permite uma maior facilidade na implementação, permite que você altere o comportamento em tempo de execução, desde que o objeto que você está compondo implementa a interface de comportamento correta.

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.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 1th edition.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 1th edition.

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 Univeristy - Silicon Valley Campus*.

Sierra, K., Bates, B., Robson, E., and Freeman, E. (2004). *Head First Design Patterns*. O'Reilly Media, Inc.

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)*.