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

Resumo. *Os métodos de desenvolvimento de software ágil têm se tornado uma das principais abordagens no desenvolvimento de software, trazendo um grande impacto na engenharia de software, aumentando e alterando a percepção e o conceito de qualidade na codificação e na entrega de códigos. Saber como utilizar conceitos e princípios do design orientado a objeto (OOPD), CLEAN CODE - não excludente, produção de testes unitários e/ou de integração - para aumentar a produtividade são extremamente importantes. Muitas das falhas são por falta de boas práticas mais básicas de programação, tais problemas, embora não correspondam necessariamente a bugs, dificultam a leitura do código, o debug, novas implementações ou qualquer futura manutenção, diminuindo a efetividade na fase do desenvolvimento. Este artigo corrobora com a aplicação desses conceitos e define uma melhor estratégia para alcançar um desenvolvimento mais eficaz e eficiente, produzindo códigos com alta qualidade e alto grau de manutenabilidade, com o objetivo geral de definir uma estratégia de codificação, contendo ações necessárias para maximizar e monitorar a codificação, visando um código com alta qualidade e manunenabilidade, utilizando método de pesquisa quantitativas baseadas em métricas passiveis de serem coletadas e monitoradas através de analises estáticas de código (automatic code-review) junto com métricas qualitativas através de lições aprendidas, projetos bem sucedidos e alguns insights sobre as melhores soluções e designs, aplicado a alguns estudos de casos, simulações de erros ou de problemas, trechos ou demais exemplos de código a serem utilizados nesse artigo.*

Abstract. *Agile software development methods have become a major approach in software development, bringing a major impact on software engineering, increasing and changing the perception and concept of quality in coding and code delivery. Knowing how to use object-oriented (OOPD), CLEAN CODE concepts and principles - not excluding, producing unit and / or integration tests - to increase productivity are extremely important. Many of the flaws are due to lack of basic programming best practices, such problems, although they do not necessarily correspond to bugs, make it difficult to read code, debug, new implementations or any future maintenance, reducing effectiveness in the development phase. This article corroborates with the application of these concepts and defines a better strategy to achieve a more efficient and efficient development, producing codes with high quality and high degree of maintainability, with the general objective of defining a coding strategy, containing actions necessary to maximize and to monitor coding, aiming at a code with high quality*

and manunenability, using quantitative research method based on metrics that can be collected and monitored through static code analysis (automatic code-review) along with qualitative metrics through lessons successful projects, and some insights on best solutions and textures, applied to some case studies, error or problem simulations, snippets, or other code examples to use in this article.

Palavras-chaves: clean-code, software development, object-oriented design principles, test driven development (TDD)

Contents

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* [Nedre, 2011].

O manifesto *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 o instituto 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 [Alliance, 2006].

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 [Wells, 2009].

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”.

Max Kanat-Alexander [Max Kanat, 2012] também reforça as consequências de ignorar o fato de existir um futuro, e cair no erro de criado software 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 opcional no desenvolvimento, reforçando a importância dos aspectos de um código bem feito (capacidade de extensão, código testavel, fácil compreensão, alta coesão, desacoplamento, entre outros).

Em um artigo ainda em 2005, Richard C. Martin [Martin, 2000] aborda os conceitos e a importância do que é *design* orientado a objetos, seus benefícios e ainda seus custos. Ainda, segundo o author Richard C. Martin [Martin, 2000], essas simples perguntas por mais óbvias e bobas que possam parecer em tempos aonde praticamente todos os desenvolvedores de software estão usando uma linguagem orientada a objetos de algum tipo, ressaltam a importância da questão, evidenciando um cenário aonde a maioria usa essas línguas sem saber o por quê, e sem saber como tirar o máximo proveito delas.

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.

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: (i) Identificar a correta aplicação dos conceitos de OOPD/*CLEAN CODE* ,(ii) identificar as melhores práticas para a produção de código eficiente ,(iii) aplicar as melhores práticas identificadas ,(iv) propor uma estratégia para aumentar a eficiência na codificação .

Este trabalho não incluiu (i) aprodundar nos conceitos relacionados, mas apenas uma apresentação rápida do seu entendimento.

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)[Diniz Junior and Domingos da Silva, 2015], não apenas em relação ao presente, mas também quanto ao seu futuro, maximizar a rapidez de entendimento do código, ou sua possivel extensão. Também inclui 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[Max Kanat, 2012]. 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 1, apresenta a introdução, descrevendo o contexto, problema de pesquisa e objetivos. O capítulo 2, apresenta a revisão bibliografia, contêm os pricipais conceitos relacionados. O capítulo 3, apresenta os trabalhos relacionados. O capítulo 4, apresenta a metodologia de pesquisa utilizada. O capítulo 5, apresenta o entendimento do problema. O capítulo 6, apresenta a concepção da solução proposta. O capítulo 7, apresenta a avaliação e os resultados alcançados. O capítulo 8, apresenta os trabalhos futuros. Por final, o capítulo 9, é apresentado a conclusão e as considerações finais deste trabalho.

Part II

REVISÃO BIBLIOGRÁFICA

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.

É destacado pontos importantes para alcançar o nível de qualidade como principios, padrões, e boas práticas, passando por estudos de casos de diversos tipos e complexidades.

Inclui também uma base de conhecimento que descreve e explica o racícinio utilizado durante as leituras, as escritas e durante a aplicação do ”clean code”, assim como:

- utilização de nomes significativos,
- codificação de funções pequenas e com objetivos claros,
- The Stepdown Rule [1],
- formatação de código,
- a importância de objetivar um *design*/sistema testavel [2],
- entre outros pontos.

Robert C. Martin [Martin, 2008] 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 pratica-lo você mesmo, e assistir você*

[1] The Stepdown Rule [Martin, 2008], um código deve ser escrito de modo ”*TOPDOWN*”, aonde o programa fosse um conjunto de parágrafos ”TO-DO”, cada um descrevendo o nível atual de abstração e referenciando subsequentemente 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

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."

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

Richard C. Marti [Martin, 2002] afirma que 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 possível a todo hora. "

Posteriormente, em 2004, estes cinco princípios tornou conhecido pelo acrónimo "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 razão para mudar."

Segundo Richard C. Martin, [Martin, 2017], **OCP**[4] sendo um conceito genérico, sendo considerado uma das razões para mudar ou alterar uma classe como uma responsabilidade.

Este princípio afirma que, se tivermos duas ou mais razões, ou responsabilidades, para mudar uma classe, há uma grande probabilidade da necessidade de dividir essa funcionalidade em duas classes distintas. Sendo assim, cada classe irá lidar com apenas uma responsabilidade e no futuro, se precisarmos adicionar um novo comportamento, deverá ser adicionado na classe especifica que mais se adequa a essa mudança.

Quando é necessário realizar uma alteração em uma classe que possuiu uma sobrecarga comportamentos, a alteração pode afetar inúmeras outras funcionalidade de outras classes.

[3] *"Desing smell"* [Martin, 2002] é um sintoma, algo mensurável, subjetivamente se não objetivamente, que normalmente, é o resultado de uma ou mais violações aos princípios.

[4] **SRP** foi inicialmente introduzida por Tom DeMarco [DeMarco, 1979].

Esse principio mostra a importância de pensar em termos de responsabilidades, ajuda a projetar melhor a aplicação, questionar a lógica ou o comportamento a ser adicionado deverá viver na classe em questão ou não, adicionar uma nova classe para implementar as novas necessidades.

Dividir as grandes classes em menores evita o problema conhecido como ”*God Class*”[5][Riel J, 1996], para o português ”classe deus”.

2.1.2. OCP - Open closed principle

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

Segundo Richard C. Martin, [Martin, 2017], com base nesse principio, é necessário considerar ao criar as classes, certificar de que quando precisar estender seu comportamento, não precisará mudar a classe, mas estende-la. O mesmo princípio pode ser aplicado para módulos, pacotes, bibliotecas.

Se você tem uma biblioteca contendo um conjunto de classes, há muitos motivos pelos quais você preferirá estende-la sem alterar o código que já estava escrito (compatibilidade com versões anteriores, teste de regressão, etc...).

Ao se referir às classes, **OCP** pode ser assegurado pelo uso de classes abstratas, aonde as classes concretas implementam seu comportamento. Isso exigirá que as classes concretas estendam algum comportamento especifico das abstratas em vez de altera-las. Alguns exemplos e casos específicos deste são os *design pattern*: (i) *Template Pattern* e (ii) *Strategy Pattern*.

2.1.3. LSP - Liskov substitution principle

> *”Subtipos* [6] *devem ser substituíveis por suas classes base.”*[7]

Segundo Richard C. Martin, [Martin, 2017], este princípio é apenas uma extensão do **OCP** em termos de comportamento, aonde deverá ser garantido que novas classes derivadas estejam estendendo as classes base sem alterar seu comportamento. As novas classes derivadas devem ser capazes de substituir as classes base sem qualquer alteração no código.

2.1.4. ISP - Interface segregation principle

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

[5] ”*God class*”, sendo descrito como um objeto que controla muito outros objetos, que cresceu muito além de qualquer lógica, se tornando ”uma classe que controla tudo”.

[6] Também conhecidas como ”Classes derivadas”.

[7] 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.”*

Segundo Richard C. Martin, [Martin, 2017], este princípio ensina a forma de criar interfaces. Quando é necessário criar interfaces, é necessário adicionar apenas métodos que são utilizados ou extremamente obrigatórios. É necessário analisar, ao adicionar métodos incorretamente, ou que não deveriam estar na interface, as classes que implementam a interface terão que implementar esses métodos.

Um exemplo educacional, ao criarmos uma interface chamada **Pato** e adicionar um segundo método **voar**, todos as classes terão que implementar esse novo comportamento, porém se um **Pato** é um robô ou um pato de brinquedo não obrigatoriamente eles irão precisar desse método.

Conclusão para o exemplo, interfaces que contêm métodos que não são específicos são chamadas *polluted interfaces* ou *fat interfaces*, e essa característica deve ser evitada, classes não devem implementar métodos que não são utilizados. A aplicação do **ISP** proporciona um baixo acoplamento e alta coesão.

Ao falar sobre o acoplamento, a coesão também é mencionada. A alta coesão significa manter coisas similares e relacionadas. A união de coesão e acoplamento é o design ortogonal.A ideia é manter seus componentes focados e tentar minimizar as dependências entre eles.

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.*

Segundo Richard C. Martin, [Martin, 2017], afirma que devemos desacoplar módulos de alto nível de módulos de baixo nível, introduzindo uma camada de abstração entre classes de alto nível e classes de baixo nível. Ivertendo a dependência: em vez de escrever nossas abstrações com base nos detalhes, devemos escrever os detalhes com base em abstrações.

Inversão de Dependência ou Inversão de Controle, são melhores termos de conhecimento referentes à forma como as dependências são realizadas. Na maneira clássica, quando um módulo de software (classe, estrutura,) precisa de algum outro módulo, ele inicializa e mantém uma referência direta a ele. Isso fará com que os 2 módulos estejam altamente acoplados. Para desacoplá-los, o primeiro módulo fornecerá uma referencia (uma propriedade, parâmetro,) e um módulo externo que controle as dependências injetará a referência ao segundo.

Ao aplicar **DIP**, os módulos podem ser facilmente alterados por outros módulos apenas mudando o módulo de dependência (implementação).

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

Segundo [Larman, 2004] os padrões GRASP englobam uma série de princípios baseados em conceitos de Orientação a Objetos. Partindo de análises que procuram definir quais as obrigações dos diferentes tipos de objetos em uma aplicação, estes patterns disponibi-

lizam uma série de recomendações que procuram favorecer a obtenção de sistemas melhor estruturados. Ainda segundo o autor GRASP procuram fornecer diretrizes para a construção de aplicações bem estruturadas e que possam ser facilmente adaptáveis diante da necessidade de mudanças. A consequência direta das recomendações propostas por estes patterns é um código melhor organizado, de fácil manutenção e ainda, capaz de ser compreendido por diferentes desenvolvedores sem grandes dificuldades.

Craig Larman, afirma que ”a ferramenta crucial de projeto para desenvolvimento de software é uma mente bem educada em princípios de projeto. Não é UML ou qualquer outra tecnologia”. [Larman, 2004] Assim, GRASP é definido como um conjunto de ferramentas mentais, um auxílio de aprendizagem para ajudar no projeto de software orientado a objetos.

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

2.2.1. Controller

Segundo [Larman, 2004] Craig Larman, o padrão controlador atribui a responsabilidade de manipular eventos do sistema para uma classe que não seja de interface do usuário (UI) que representa o cenário global ou cenário de caso de uso. Um objeto controlador é um objeto de interface não-usuário, responsável por receber ou manipular um evento do sistema.

Um caso de uso controlador deve ser usado para lidar com todos os eventos de casos de uso e pode ser usado para mais de um caso de uso (por exemplo, para casos de uso como Criar usuário e Excluir usuário, pode ter um único UserController, em vez de dois casos de uso controllers separados).

É definido como o primeiro objeto além da camada UI que recebe e coordena (”controla”) operações do sistema. O controlador deve delegar o trabalho que precisa ser feito para outros objetos; ele coordena ou controla a atividade. Ele não deve fazer muito trabalho por si próprio. O Controller GRASP pode ser considerado uma parte da camada de aplicação/serviço [2] (assumindo que a aplicação tenha feito uma distinção explícita entre a camada de aplicativo/serviço e a camada de domínio em um sistema orientado a objetos com camadas comuns em uma arquitetura lógica do sistema de informações).

2.2.2. Creator

Segundo [Larman, 2004], criação de objetos é uma das mais comuns atividades em um sistema orientado a objetos. Descobrir qual classe é responsável por criar objetos é uma propriedade fundamental da relação entre objetos de classes particulares.

Em geral, uma classe B deve ser responsável por criar instâncias de classe A se uma, ou preferencialmente mais, das seguintes afirmações se aplicam:

Instâncias de B contêm ou agregam instâncias de A; Instâncias de B gravam instâncias de A; Instâncias de B utilizam de perto instâncias de A; Instâncias de B têm as informações de inicialização das instâncias de A e passam isso na criação.

2.2.3. Indirection

Segundo [Larman, 2004], indireção suporta baixo acoplamento (e potencial de reutilização) entre dois elementos, atribuindo a objeto intermediário a responsabilidade de ser o mediador entre eles. Um exemplo é a introdução do componente controlador para mediação entre dados (modelo) e sua representação (visualização) no padrão MVC.

2.2.4. Information Expert

Segundo [Larman, 2004], especialista na informação é um princípio utilizado para determinar onde delegar responsabilidades. Essas responsabilidades incluem métodos, campos computados, e assim em diante.

Usando o princípio information expert, uma abordagem geral para atribuir responsabilidades é olhar para uma determinada responsabilidade, determinar a informação necessária para cumpri-la e depois determinar onde essa informação está armazenada.

Information expert guia a colocará a responsabilidade na classe com a maioria das informações necessárias para cumpri-la.

2.2.5. High Cohesion

Segundo [Larman, 2004] a alta coesão é um padrão avaliativo que tenta manter os objetos adequadamente focados, gerenciáveis e compreensíveis. A alta coesão é geralmente utilizada em suporte de baixo acoplamento. A alta coesão significa que as responsabilidades de um determinado elemento estão fortemente relacionadas e altamente focadas. A quebra de programas em classes e subsistemas é um exemplo de atividades que aumentam as propriedades coesivas de um sistema. Alternativamente, a baixa coesão é uma situação em que um determinado elemento tem muitas responsabilidades distintas, não relacionadas. Elementos com baixa coesão muitas vezes sofrem de ser difíceis de entender, reutilizar, manter e são avessos à mudança.

2.2.6. Low Coupling

Segundo [Larman, 2004] o acoplamento é uma medida de quão forte um elemento está conectado, tem conhecimento ou depende de outros elementos. O baixo acoplamento é um padrão de avaliação que determina como atribuir responsabilidades de suporte:

menor dependência entre as classes, mudança em uma classe com menor impacto em outras, maior potencial de reutilização.

2.2.7. Polymorphism

Segundo [Larman, 2004] o princípio do polimorfismo, a responsabilidade de definir a variação dos comportamentos com base no tipo é atribuída ao tipo para o qual essa variação ocorre. Isto é conseguido utilizando operações polimórficas. O usuário do tipo deve usar operações polimórficas em vez de ramificações explícitas com base no tipo.

2.2.8. Protected Variations

Segundo [Larman, 2004] o padrão variações protegidas protege elementos das variações em outros elementos (objetos, sistemas, subsistemas) envolvendo o foco de instabilidade com uma interface e usando polimorfismo para criar várias implementações desta interface.

2.2.9. Pure Fabrication

Segundo [Larman, 2004] Uma fabricação/invenção pura é uma classe artificial que não representa um conceito no domínio do problema, especialmente feito para conseguir baixo acoplamento, alta coesão e o potencial de reutilização derivado (quando uma solução apresentada pelo padrão information expert não é). Esse tipo de classe é chamado de ”serviço” em padrão orientado a domínio.

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*

A lei de Demeter foi desenvolvida em 1988 por Karl Lieberherr e Ian Holland, da Northeastem Univerity, com uma ideia extremamente simples: organizar e reduzir dependências entre classes.

Na classe C, para todos os métodos M definidos em C, todos os objetos com o qual M se comunica deve ser:

Argumento de M Um membro de C

Objetos criados por M, por métodos que M invoca ou objetos de escopo global na classe são considerados argumentos de M.

Esta lei tem dois propósitos primários:

Simplificar modificações; Simplificar a complexidade da programação.

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

> *Cada parte do conhecimento deve ter uma representação única, não ambígua e definitiva dentro do sistema.*

Segundo os autores Andy Hunt e Dave Thomas [Hunt and Thomas, 2004] o principio DRY [Hunt and Thomas, 1999], abreviação do inglês *"Don't Repeat Yourself"* ("Não se Repita"), tem como ideal manter as representações de qualquer ideia, qualquer pedaço de conhecimento de um sistema em apenas um lugar. Define centralizar, não necessariamente, por acabar com cópias físicas de código, mas visa existir apenas uma representação deverá ser a fonte definitiva. Ainda, segundo o autor [Hunt and Thomas, 2004], idealmente, o sistema deverá automaticamente gerar as copias a partir da fonte definitiva, desse modo quando houver uma mudança de código, só precisará realizar em único ponto, minimizando o desastre de inconsistências e potenciais bugs difíceis de encontrar por ambiguidade no código.

DRY aplica-se ao especialmente ao código, mas também a qualquer outra parte do sistema e paravida diária dos desenvolvedores - processos de construção, documentação, esquema de banco de dados, código comentários, e assim por diante.

3. *Test-driven development (TDD)*

Kwent Back [Professional, 2002] define *test-driven development* (TDD) como uma abordagem evolutiva para o desenvolvimento, combinando o *test-first development* onde escreve um teste primeiro, e em seguida, produz um código de produção que contemple o objetivo do teste, e realiza refatorações. Ainda segundo Kwent Back um principal objetivo do TDD é uma visão aonde o objetivo é a especificação e não a validação, uma forma de pensar em seus requisitos ou no *design* antes de escrever seu código funcional (implica e determina que o TDD como um importante requisito ágil e uma técnica de *design* ágil). Outra visão é que TDD é uma técnica de programação, Como diz Ron Jeffries [Jeffries, 2017], o objetivo do TDD é escrever um código limpo que funciona.

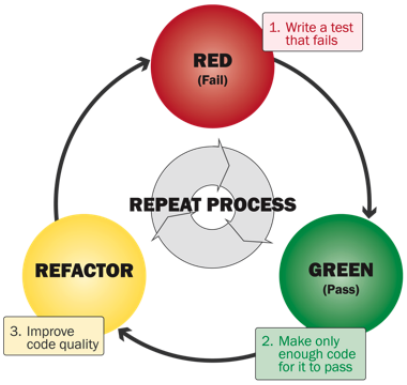

Figure 1. Processo simplificado do TDD

1. Adicionar um teste que falhe
2. Codifique até o teste passar
3. Refatoração (analise e melhore o *design* / código).

4. Repetir.

Part III

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, a legibilidade e o tempo de manutenção são melhores.

(iii) [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.

(iv) [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.

(v) [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.

(vi) [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 IV

Metodologia de Pesquisa

Esta pesquisa possui caráter qualitativo, e quantitativo em relação a sua abordagem e coleta de dados, utilizando de métricas em ambos os tipos. Conforme Prodanov e Freitas [PRODANOV, 2013], a pesquisa quantitativa considera que tudo pode ser quantificável, traduzido em números opiniões e informações para classificá-las e analisá-las. Dessa forma, buscando métricas para serem monitoradas afim de determinar a qualidade do código em relação ao objetiva desse trabalho.

Também considerando um caráter quantitativo como abordagem e coleta de dados conforme autores Prodanov e Freitas [PRODANOV, 2013] considerando a relação dinâmica entre o mundo real e um vínculo indissociável entre o mundo objetivo e a subjetividade do sujeito que não pode ser traduzido em números exploratório. Com base nisso será analisados os resultados a fim de determinar uma forma não numericamente representativa as melhores práticas, recomendações para a codificação. Assim seguindo o autor, como pesquisa qualitativa há um contato direto do pesquisador com a situação estudada, esse trabalho visa entender a perspectiva no decorrer da aplicação, e apresentar as experiências e percepções.

Em relação a natureza, o mesmo será aplicada, conforme Prodanov e Freitas [PRODANOV, 2013], objetivando gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos. Com objetivo exploratório descritivo, do ponto de vista dos procedimentos técnicos, o dados serão obtidos a partir de estudo de caso, aonde ”envolve o estudo profundo e exaustivo de um ou poucos objetos de maneira que permita o seu amplo e detalhado conhecimento.” [PRODANOV, 2013]. Utilizando de trechos de códigos ou *design* e/ou problemas que representam o problema citado nesse trabalho.

Part V

Entendimento do problema

Levando em consideração pontos importantes que tornam díficil trabalhar com software [Fowler et al., 1999]:

- Programas dificies de ler, são dificeis de modificar.
- Programas com lógica dupilcada são dificies de modificar.
- Programas com lógicas condicionais díficeis tornam o software díficil.
- Programas que requerem comportamentos adicionais e exigem mudanças no código corrente, são díficeis de modificar.

Podemos identificar inúmeros pontos durante o desenvolvimento que representam violações aos principios, e assim, tornam a manutenção uma tarefa mais dificil:

- Classe com muitas responsabilidades (GOD CLASS).

- Método muito complexo.
- Design favorecendo implementação ao invés da abstração.
- Duplicidade de código
- Alto acoplamento
- Métodos adicionados sem necessidade.

Part VI

Concepção da solução proposta

4. *x*

5. *Static Code Analysis*

A análise da qualidade do código fonte é uma parte essencial do processo de Integração Contínua. Juntamente com testes automatizados, é o elemento-chave para fornecer software confiável sem muitos bugs, vulnerabilidades de segurança ou vazamentos de desempenho. Provavelmente, o melhor analisador de código estático que você pode encontrar no mercado é o SonarQube. Tem suporte para mais de 20 linguagens de programação. Ele pode ser facilmente integrado aos mecanismos de Integração Contínua mais populares, como o Jenkins ou o TeamCity. Finalmente, ele possui muitos recursos e plug-ins que podem ser gerenciados facilmente a partir de um extenso painel da web.

No entanto, antes de prosseguirmos para discutir sobre as capacidades mais poderosas desta solução, vale a pena perguntar Por que fazemos isso? Seria produtivo forçar os desenvolvedores a se concentrarem na qualidade do código? Provavelmente a maioria de nós é programadora e sabemos exatamente que todo mundo espera de nós para entregar código que atenda às demandas de negócios ao invés de parecer legal Afinal, nós realmente queremos quebrar a construção por não cumprir regra não importante como o comprimento máximo da linha? um pouco de prazer. Por outro lado, assumir o código-fonte de alguém que não estava prestando atenção a qualquer boa prática de programação também não é bem-vindo se você sabe o que quero dizer. Mas fique calmo, o SonarQube é a solução certa para você. Neste artigo, mostrarei a você que carregar uma alta qualidade de código pode ser uma boa diversão e, acima de tudo, você pode aprender mais como desenvolver um código melhor, enquanto outros membros da equipe gastam tempo corrigindo seus bugs

As ferramentas de análise de código estático são amplamente usadas no desenvolvimento de Java para melhorar a base de código e identificar possíveis vulnerabilidades junto com falhas de design. Cada ferramenta tem seu próprio recurso, propósito e força, o que ajuda a aumentar a qualidade do código e faz de você um desenvolvedor melhor. Eu irei me referir a Static Analysis Tools como SCA a partir de agora.

5.1. PMD

Analisa a Árvore de Sintaxe Abstrata (AST) gerada pelo JavaCC e não requer a compilação real. Identifica problemas potenciais principalmente código morto e dupli-

cado, complexidade ciclomática, expressões complicadas e quase tudo de que o Checkstyle é capaz. O PMD é uma ferramenta extremamente útil para analisar o código-fonte. De acordo com o site do projeto, ele ”verifica o código-fonte e procura possíveis problemas, possíveis bugs, códigos não usados e sub-ótimos, expressões complicadas e códigos duplicados”. O PMD vem com um enorme conjunto de regras que podem analisar muitas coisas diferentes no código java. Para nomear alguns:

- Esvaziar blocos try / catch

- Expressões complicadas

- Usando .equals () em vez de ”==”

- Variáveis e importações não utilizadas

- loops desnecessários e instruções if

- Aplicar convenções de nomenclatura

Além disso, o PMD vem com um detector de copiar e colar para localizar blocos de códigos copiados e colados. O melhor de tudo, as regras PMD personalizadas são facilmente escritas com XPath e uma GUI incluída no software.

5.2. FindBugs

Ele analisa o código de byte Java, principalmente .classes para encontrar qualquer falha de projeto e possíveis bugs. Ele precisa de código compilado para contornar e, eventualmente, será rápido, já que funciona em nível de código de bytes. As principais categorias desta ferramenta são: Exatidão, Má prática, Código desonesto, Corrigidez multissegmentada, Desempenho malicioso, Vulnerabilidade de código, Segurança experimental e internacionalização O FindBugs é outro analisador de código estático muito semelhante ao PMD. A maior diferença entre o PMD e o FindBugs é que o FindBugs funciona no código de bytes, enquanto o PMD trabalha no código-fonte. FindBugs pode encontrar coisas como:

- Uso indevido de .equals () e .hashCode () - Conjuntos inseguros - Quando algo sempre será nulo - Possíveis StackOverflows - Possíveis exceções ignoradas

5.3. Checkstyle

Ele basicamente analisa o código-fonte e procura melhorar o padrão de codificação ao atravessar o simples AST gerado pelo Checkstyle. Ele verifica o código-fonte para convenções de codificação como cabeçalhos, importações, espaços em branco, formatação etc. Checkstyle é uma ferramenta para analisar estilo e convenções de codificação. Ele não interromperá as exceções de rouge, mas fornecerá feedback sobre como o código é organizado. Checkstyle é útil para garantir que o código Java está sendo escrito corretamente. Aqui estão algumas coisas que o Checkstyle vai pegar:

- Falta / javadoc impróprio - espaço em branco - Colocação de chaves e parênteses - Comprimento da linha - Convenções de nomenclatura

O Checkstyle é muito diferente do PMD e do FindBugs. Embora tenha verificações de itens como blocos de captura vazios e .equals () versus ‘==’, o foco principal do projeto é garantir que o estilo de codificação esteja de acordo com um conjunto de convenções.

6. *Manage code quality*

Hora de respirar! Na verdade, não há necessidade de tomar essa posição, uma vez que essas ferramentas não competem, mas são complementares e devem ser usadas simultaneamente, como é o caso do Sonar. Cada um deles está direcionando principalmente um certo tipo de regras de codificação: convenções (Checkstyle), práticas ruins (PMD) e possíveis bugs (FindBugs).

O tipo de convenção abrange nomenclatura, comentários e convenções de formato. Aqui estão alguns exemplos :

Existe javadoc em métodos públicos? O projeto está seguindo as convenções de nomenclatura da Sun? O código é escrito com um formato consistente?

O tipo de convenção tem frequentemente a reputação de ser bastante inútil, pois as regras são muito simples. Como explicar então que a maioria dos projetos de código aberto fornece um arquivo checkstyle em seu guia de desenvolvimento, quando os mesmos projetos geralmente descartam algo inútil? É verdade que as regras de convenção não têm impacto na estabilidade, desempenho ou confiabilidade de um aplicativo. No entanto, o tipo de convenção é a cola que permite que as pessoas trabalhem juntas e liberem sua criatividade, em vez de gastar tempo e energia na compreensão de código inconsistente.

O tipo de más práticas consiste em comportamentos bem conhecidos que quase sistematicamente levam a dificuldades ao longo do tempo. Aqui estão alguns exemplos de más práticas:

Pegando uma exceção sem fazer nada Ter código morto Muitos métodos complexos Uso direto de implementações em vez de interfaces Implementando o método hashcode () sem o método não igual (Object object)

O PMD é um tipo de anjo que sempre olha por cima do seu ombro para lembrá-lo de práticas ruins, da mesma forma que seu senso comum o lembra de interagir com seu cliente ao desenvolver uma funcionalidade completa e responder a perguntas de seus colegas de trabalho.

O tipo de bugs em potencial ajuda a detectar o que não está claramente visível no código e a entender por que sequências de código podem levar a possíveis bugs. Aqui estão alguns exemplos de possíveis bugs:

Sincronização em Boolean pode levar a deadlock Pode expor a representação interna retornando a referência ao objeto mutável O método usa o mesmo código para dois ramos

Bugs são como relações humanas, nem sempre é fácil entender o problema, pois há muitos parâmetros a serem levados em conta. Pode ser uma boa ideia, às vezes, consultar um analista para ajudar a resolvê-los :-). Findbugs é um tipo de analista para o seu código-fonte!

O que há com o Macker? Enquanto o Checkstyle, o PMD e o Findbugs concentram sua atenção na análise de fontes e na aplicação de regras, a Macker dá um grande passo atrás para questões de arquitetura de identidade. Aqui estão alguns exemplos de regras arquitetônicas:

Classes na camada de interface do usuário podem não acessar diretamente a ca-

mada de objeto de dados ou usar classes em java.sql Sistemas externos não podem acessar classes de implementação internas (com sufixo 'Impl') Um módulo funcional pode acessar outro somente por meio de sua API Somente classes implementando interfaces em javax.ejb e determinados pacotes de frameworks podem usar as APIs EJB

Macker olha para a sua aplicação da mesma forma que um homem na lua olha para a terra: ei, o que está acontecendo? O oceano pacífico está muito perto do continente europeu! Uma vez que você tenha uma ideia clara de como deve ser sua arquitetura, você pode facilmente modelá-la com o Macker para manter sua arquitetura consistente ao longo do tempo. Com a Macker, você pode definir convenções arquitetônicas e identificar práticas ruins de arquitetura.

7. Sonar

The following schema shows how SonarQube integrates with other ALM tools and where the various components of SonarQube are used.

O SonarQube é uma plataforma de código aberto para inspeção contínua da qualidade do código. Usando a análise de código estático, ele tenta detectar bugs, códigos cheiros e vulnerabilidades de segurança. O SonarQube suporta vários idiomas por meio de conjuntos de regras integrados e também pode ser estendido com vários plug-ins.

Neste artigo, estamos particularmente interessados em questões de segurança. Muitas ferramentas de análise estática existem para a linguagem Java, incluindo aquelas de fonte livre e de código aberto. Algumas vantagens do SonarQube são as seguintes:

É ativamente desenvolvido e bem integrado. Muitos plugins estão disponíveis para uso como parte de pipelines de integração contínua, incluindo Maven, Jenkins e GitHub. Seus conjuntos de regras internos podem ser estendidos com plug-ins que são mais orientados à segurança. Por exemplo, usaremos o plugin FindBugs para aproveitar as regras do FindBugs. Ele também pode relatar coisas como código duplicado, cobertura de código ou padrões de codificação.

https://sonarcloud.io/dashboard?id=net.sourceforge.pmdhttps://sonarcloud.io/dashboard?id=net.jav

Part VII
Avaliação da solução proposta

Part VIII
Trabalhos futuros

Part IX
Conclusão

```
public class Abstract {}
```

```
1     public class Abstract {}
```

References

Agile (2001). Manifesto for agile software development. http://agilemanifesto.org/.

Alliance, T. W. (2006). Waterfall. http://www.waterfall2006.com/.

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)*.

DeMarco, T. (1979). *Structured Analysis and System Specification*. Prentice Hall, 1st edition edition.

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.

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1 edition edition.

Hunt, A. and Thomas, D. (2004). Keep it dry, shy, and tell the other guy. Technical report, The Pragmatic Programmers.

Jeffries, R. E. (2017). Clean code: Too much of a good thing? https://ronjeffries.com/xprog/articles/too-much-of-a-good-thing/.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 3rd edition edition.

Martin, R. C. (2000). The principles of ood. http://butunclebob.com//ArticleS.UncleBob.PrinciplesOfOod.

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.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)*. 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.

Nedre, N. (2011). How to choose between agile and lean, scrum and kanban — which methodology is the best? "https://realtimeboard.com/blog/choose-between-agile-lean-scrum-kanba

Niralem, S., Kawati, V., and G.R, S. (2017). Quality code: Eliminating technical debt. *Imperial Journal of Interdisciplinary Research (IJIR)*.

PRODANOV, Cleber C.; FREITAS, E. C. (2013). Metodologia do trabalho científico: Métodos e técnicas de pesquisa e do trabalho acadêmico. Novo Hamburgo: Ed. Feevale, 2013. Livro eletrônico.

Professional, A.-W. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional, 1st edition edition.

Research, C. P. and Technologies, C. (2017). Accelerating velocity and customer value with agile and devops. Technical report, CA Technologies.

Riel J, A. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 1 edition edition.

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 Unveristy - 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*.

Wells, D. (2009). Working software. Technical report, Agile-Process.org. http://www.agile-process.org/working.html.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects?. *28th IEEE International Conference on Software Maintenance (ICSM)*.