

Maturidade no processo de desenvolvimento de software

Qualidade de código contínua

João Carlos Brasileiro Stefenon de Almeida¹

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

jcbbrasileiro@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, do debug, de novas implementações ou qualquer futura manutenção, diminuindo a efetividade na fase do desenvolvimento. Esse 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 manutenibilidade, 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 manutenibilidade. O presente estudo, utiliza método de pesquisa quantitativa baseada em métricas passíveis de serem coletadas e monitoradas por meio de análises estáticas de código (automatic code-review) , junto com métricas qualitativas por meio 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)

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] surgiu 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 há 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].

Uma 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 exige um aumento na maturidade no desenvolvimento, criando a necessidade de gerar artefatos que evidenciassem e garantissem, desde o início, dois grandes aspectos: (i) Qualidade e (ii) a garantia que cada entregáveis operasse corretamente, conforme evolução do próprio software [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 criar 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 testável, fácil compreensão, alta coesão, desacoplamento, entre outros).

Em um artigo publicado 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 onde praticamente todos os desenvolvedores de software estão usando uma linguagem orientada a objetos de algum tipo, ressaltam a importância da questão, evidenciando um cenário em que a maioria usa essas línguas sem saber o porquê, e sem saber como tirar o máximo proveito delas.

A partir dessas novas perspectivas e preocupações, e visando amadurecer a qualidade contínua 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 com 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 OODP/*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 .

Esse trabalho não incluiu (i) Aprofundar em detalhes nos conceitos relacionados, mas apenas abordar uma 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, maximizando a rapidez de entendimento do código, ou sua possível extensão. O estudo também inclui 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[Max Kanat, 2012]. Também visa, da perspectiva da empresa, evidenciando que diante de um código com qualidade, melhor estruturado, bem testado, de 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 práticas. Utilizando do método de pesquisa estudo de caso, por meio de cenários reais, baseados em cenários, ou simulação do problema, para analisar os resultados obtidos.

O presente trabalho foi estruturado em 7 capítulos. O capítulo um apresenta a introdução, descrevendo o contexto, problema de pesquisa e objetivos. O capítulo dois apresenta a revisão bibliográfica, contendo os principais conceitos relacionados. O capítulo três apresenta os trabalhos relacionados. O capítulo quatro apresenta a metodologia de pesquisa utilizada. O capítulo cinco apresenta o entendimento do problema. O capítulo seis apresenta a concepção da solução proposta. O capítulo sete apresenta a avaliação e os resultados alcançados. O capítulo oito apresenta os trabalhos futuros. E, por final, o capítulo nove é apresentado a conclusão e as considerações finais desse trabalho.

Part II

REVISÃO BIBLIOGRÁFICA

1. *Clean Code*

”Clean code” [Martin, 2008] é um conceito subjetivo que, de modo muito genérico, 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 princípios, 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 raciocínio 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,

¹The 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.

- a importância de objetivar um *design*/sistema testável ²,
- entre outros pontos.

Robert C. Martin [Martin, 2008] também destaca o esforço necessário 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 falhar. 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 princípios 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 apresentados inicialmente 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 tornaram-se conhecidos pelo acrônimo "SOLID", após Michael Feathers reorganizar a sequência 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], **Single responsibility principle (SRP)**⁴ é um conceito genérico, que considera as ideias referente as razões para mudar ou alterar uma classe como uma responsabilidade.

²Um 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

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 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 específica 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 funcionalidades de outras classes.

Listing 1. Exemplo em conformidade ao SRP

```
1 public interface RelatorioCliente{
2     RelatorioClienteDTO clientesPendentes();
3     RelatorioClienteDTO clientesSemMovimentacao(Intervalo ref);
4     RelatorioClienteDTO clientesComMovimentacao(Intervalo ref);
5     RelatorioClienteDTO clientes(Filtro filtro);
6 }
```

Esse princípio mostra a importância de pensar em termos de responsabilidades, ajuda a projetar melhor a aplicação, questionar se lógica ou o comportamento a ser adicionado deverá viver na classe em questão ou não e 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".

Listing 2. Exemplo de violação ao SRP

```
1 public interface Coracao{
2     boolean bombear(Sangue sangue, Destino destino);
3     boolean direcionar(Sangue sangue, Origem origem);
4     SangueArterial enriquecer(Sangue sangue);
5     Oxigenio receberOxigenio();
6     boolean movimentarBracoDireito(Movimento movimento);
7     boolean processar(ImpulsoNervoso impulso);
8     Reacao definirReacao(Fato fato);
9     Collection<Som> falar();
10 }
```

Listing 3. Exemplo 2.1 de conformidade ao SRP

```
1 public interface GenericDAO{
2     <T> T save(T entity);
3     <T> T update(T entity);
4     boolean delete(Long id);
5     <T> Collection<T> queryByExample(T example);
6     <T> Collection<T> findAll();
7     int count();
8 }
```

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".

Listing 4. Exemplo 2.2.1 de conformidade ao SRP

```
1 public interface GenericDAOPersistence{
2     @Transactional( propagation = Propagation.SUPPORTS,readOnly = false )
3     <T> T save(T entity);
4     @Transactional( propagation = Propagation.SUPPORTS,readOnly = false )
5     <T> T update(T entity);
6     @Transactional( propagation = Propagation.SUPPORTS,readOnly = false )
7     boolean delete(Long id);
8 }
```

Listing 5. Exemplo 2.2.2 de conformidade ao SRP

```
1 public interface GenericDAOQuery{
2     @Transactional( propagation = Propagation.NEVER,readOnly = true )
3     <T> Collection<T> queryByExample(T example);
4     @Transactional( propagation = Propagation.NEVER,readOnly = true )
5     <T> Collection<T> findAll();
6     @Transactional( propagation = Propagation.NEVER,readOnly = true )
7     int count();
8 }
```

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 princípio, 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 e 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 das quais as classes concretas implementam seu comportamento. Isso exigirá que as classes concretas estendam algum comportamento específico das abstratas, em vez de alterá-las. Alguns exemplos e casos específicos deste são os *design pattern*: (i) *Template Pattern* e (ii) *Strategy Pattern*.

Listing 6. Exemplo em conformidade ao OCP

```
1 class Pessoa {
2     public boolean export(PessoaExporter exporter) {
3         return exporter.export(this);
4     }
5 }
6
7 interface PessoaExporter { boolean export(Pessoa pessoa); }
```

```

8
9 class PessoaExporterXML implements PessoaExporter {
10     @Override
11     public boolean export(Pessoa pessoa) { /* export to XML */}
12 }

```

Listing 7. Exemplo de violação ao OCP

```

1 public class FornecedorCadastroWS{
2     public boolean processaRequisicao(int id) {
3         Requisicao requisicao = new RequisicaoDAO("dsMSSQL").find(id);
4         new RequisicaoValidator().validate(requisicao);
5         new RequisicaoDAO("dsMSSQL").salvar(requisicao);
6         new RequisicaoFornecedorCadastro().processa(requisicao);
7         Email email = new EmailBuilder(new Reader("\email.properties")
8             .read("destinatario"))
9             .build(requisicao);
10        new EmailSender(new Reader("\email.properties").read("user")
11            , new Reader("\email.properties").read("pwd")
12            .enviarEmail());
13        new NegocioAnalizador.analisar(new FornecedorDAO("dsMSSQL").get(requisicao));
14    }
15 }

```

2.1.3. LSP - Liskov substitution principle

”Subtipos⁶ 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, no qual 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.

Listing 8. Exemplo em conformidade ao LSP

```

1 interface Quackable {String reproduzirSom();}
2
3 abstract class AbstractPato implements Quackable{
4     public final void quack() {System.out.println(reproduzirSom());}
5 }
6 class PatoSemiReal extends AbstractPato{
7     public String reproduzirSom() {return "quack quase real";}
8 }
9 class PatoRobo extends AbstractPato{
10    public String reproduzirSom() {return "quack eletronico";}
11 }

```

⁶Também conhecidas como “Classes derivadas”.

⁷Barbara 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 all programs P defined 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 .”

Listing 9. Exemplo de violação ao LSP

```
1 class Pato{
2     public void quack() {System.out.println("quack");}
3 }
4 class PatoToy extends Pato{
5     public void quack() {System.out.println("barulho");}
6 }
7 class PatoRobo extends PatoToy{
8     public void quack() {System.out.println("quack eletronico");}
9 }
10 class PatoMain{
11     public void run(final String[] args) {
12         Pato pato = getPato();
13         pato.quack();
14     }
15     private Pato getPato() {return new PatoRobo();}
16 }
```

2.1.4. ISP - Interface segregation principle

"Muitas interfaces específicas são melhores do que uma interface com propósito genérico."

Segundo Richard C. Martin, [Martin, 2017], este princípio ensina a forma de criar interfaces. Quando necessário é importante adicionar apenas métodos que são utilizados ou extremamente obrigatórios para o atual comportamento da mesma. Sendo fundamental analisar, se serão adicionados novos métodos ou se é mais adequado mover para uma nova interface, pois as classes que implementam essa interface terão de implementar esses métodos.

Um exemplo educacional, ao criarmos uma interface chamada **Pato** e adicionar um segundo método **voar**, todas 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.

Listing 10. Exemplo de violação ao ISP

```
1 interface Pato {
2     void quack();
```

```

3 |   boolean fly();
4 | }
5 |
6 | class PatoReal implements Pato{
7 |     public void quack() {System.out.println("quack chato real!");}
8 |     public boolean fly() {
9 |         System.out.println("flying...??");
10 |        // ommited code
11 |        return true;
12 |    }
13 | }
14 |
15 | class PatoBorracha implements Pato{
16 |     public void quack() {System.out.println("uaquiiii!!!");}
17 |     public boolean fly() {
18 |         // Pato de borracha nao voa
19 |         //logo nao deveria precisa implementar esse comportamento.
20 |         throw new UnsupportedOperationException();
21 |     }
22 | }

```

Listing 11. Exemplo em conformidade ao ISP

```

1 | interface PatoSonoro {void quack();}
2 | interface PatoVoador {boolean fly();}
3 | interface Pato extends PatoSonoro, PatoVoador {}
4 | class PatoReal implements Pato{
5 |     public void quack() {System.out.println("quack chato real!");}
6 |     public boolean fly() {
7 |         System.out.println("flying...??");
8 |         // ommited code
9 |         return true;
10 |    }
11 | }
12 | class PatoBorracha implements PatoSonoro{
13 |     public void quack() {System.out.println("kuaquiiii!!!");}
14 | }

```

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, 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. Invertendo 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 referência (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 disponibilizam 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 ao 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 por 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 colocar 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 e 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 Northeastern University, 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.

Essa 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 princípio 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 que deverá ser a fonte definitiva. Ainda, segundo o autor [Hunt and Thomas, 2004], idealmente, o sistema deverá automaticamente gerar as cópias 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 especialmente ao código, mas também a qualquer outra parte do sistema e para vida diária dos desenvolvedores - processos de construção, documentação, esquema de banco de dados, código comentários, e assim por diante.

3. *Test-driven development (TDD)*

Kent Beck [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 Kent Beck [Professional, 2002], um principal objetivo do TDD é uma visão da qual 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.

1. Adicionar um teste que falhe
2. Codifique até o teste passar

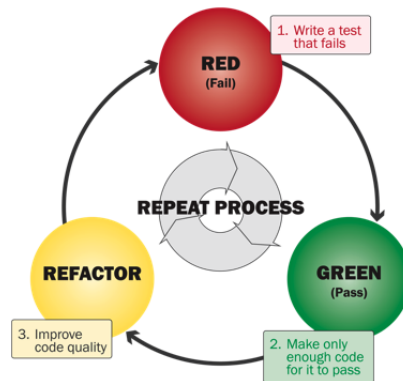


Figure 1. Processo simplificado do TDD

3. Refatoração (analise e melhore o *design* / código).
4. Repetir.

Part III

Trabalhos Relacionados

Com o objetivo de entender e levantar os diversos problemas de design e manutenibilidade de código, foram analisadas diversas iniciativas e estudos relacionados que utilizam ou exploram os conceitos, objectivando 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 apresentar 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 por meio 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, ser 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 encaminhada pelos não especialistas para o conclusão ou entrega do produto, mas é atraída 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, apresentando 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 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.

(vi) [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 IV

Metodologia de Pesquisa

Essa 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 objetivo 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 serão analisados os resultados a fim de determinar uma forma não numericamente representativa as melhores práticas e 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, a mesma 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, os dados serão obtidos a partir de estudo de caso, na qual "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

Existem alguns pontos importantes que tornam difícil trabalhar com software [Fowler et al., 1999]:

- Programas difíceis de ler, são difíceis de modificar.
- Programas com lógica duplicada são difíceis de modificar.
- Programas com lógicas condicionais difíceis tornam o software difícil.
- Programas que requerem comportamentos adicionais e exigem mudanças no código corrente, são difíceis de modificar.

Em consideração aos itens descritos, é possível identificar inúmeros pontos durante o desenvolvimento, itens que possuem uma forte relação com os itens citados, e muitos tendem a representar uma clara violação a muitos dos princípios presentes nesse trabalho, e assim, tornando a manutenção, uma tarefa mais difícil:

- Classe com muitas responsabilidades (GOD CLASS).
- Método muito complexo.
- Design favorecendo implementação ao invés da abstração.
- Duplicidade de código
- Código redundante
- Alto acoplamento
- Métodos adicionados sem necessidade.
- Métodos com muitos laços

Part VI

Concepção da solução proposta

A solução proposta é composta dos seguintes itens:

- Análise de código estático (*Static Code Analysis*).
- Gerenciamento da qualidade de código.
- Utilização de métricas de qualidade de código.

4. Análise de código estático (*Static Code Analysis*)

Brian Chess e Jacob West explicam [Chess and West, 2007] os pontos mais importantes na utilização dessa análise, e as diferentes categorias de análise estática :

- Verificação de tipo
- Verificação de estilo de código (espaços em branco, tabulação nomeação, funções obsoletas, comentários).
- Compreensão do programa (nome de variáveis, análise de divisão de função única para várias funções.)

- Verificação do programa e verificação de propriedades
- Verificação de potenciais BUGS
- Revisão de segurança

A utilização análise da qualidade do código fonte é uma parte essencial no processo de qualidade contínua, juntamente com testes automatizados, é o elemento-chave para fornecer um software confiável e, principalmente, uma base quantitativa em conjunto com uma base histórica de métricas do software.

As ferramentas de análise de código estático (SCA) são amplamente usadas no desenvolvimento [Chess and West, 2007] e cada ferramenta tem seu próprio recurso, propósito e objetivo, o que ajuda a aumentar a qualidade do código.

4.1. PMD

O PMD é uma ferramenta extremamente útil para analisar o código-fonte, podendo verificar o código-fonte à procura de possíveis problemas, possíveis bugs, códigos não usados e sub-ótimos, expressões complicadas e códigos duplicados [PMD, 2018]. A tabela 1 mostra o conjunto de regras, que podemos analisar no código.

4.2. 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 dessa 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 (compilado), enquanto o PMD trabalha no código-fonte.

4.3. Checkstyle

Checkstyle analisa o código-fonte e procura melhorar e alinhar o padrão de formatação do código, verificando o código-fonte conforme convenções de codificação como cabeçalhos, importações, espaços em branco, 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 seguindo um padrão de formatação de códigos. Alguns pontos que o Checkstyle consegue identificar:

- Falta / javadoc impróprio
- Utilização de tabulação e espaço em branco desnecessário.
- Utilização de chaves e parênteses.
- Comprimento da linha

4.4. Fortify

Fortify[MicroFocus, 2003] é um conjunto de analisadores de segurança de software que procura violações de regras de codificação específicas de segurança. Esse SCA fornece

REGRA	DESCRIÇÃO	EXEMPLO
Basic	Regras básicas gerais.	Blocos try ou catch vazios, usar return desnecessariamente, etc.
Braces	Regras relacionadas ao uso de chaves.	Ifs, whiles e loops devem possuir chaves.
Controversial	Regras de aplicação geral, mas de aplicação controversa.	Presença de construtor desnecessário, atribuição de null a uma variável etc.
Coupling	Regras relacionadas ao acoplamento entre objetos e pacotes.	Número excessivo de imports, uso de classe concreta em declaração ao invés da interface (ex.: “ArrayList l” em vez de “List l” etc.).
Design	Regras que avaliam o design do código.	Chamada a métodos que podem ser sobrecarregados no construtor, preservar o stacktrace em exceções, colocar o literal primeiro em comparações etc.
Import statement	Regras relacionadas ao uso de import.	Abuso do recurso de static import.
Naming	Regras que avaliam nomes de variáveis e métodos.	Nomes de métodos ou variáveis curtos demais ou longos demais, nomes de classes devem começar com maiúscula etc.
Optimization	Regras relacionadas à otimização.	Conversão de argumentos de métodos para final, evitar instanciação de objetos em loops etc.
Strict Exception	Regras relacionadas ao lançamento e captura de exceções.	Captura de Throwable, exceções usadas para controle de fluxo etc.
String and StringBuffer	Regras que verificam a correta utilização das classes String e StringBuffer.	Evitar a duplicação de literais, instanciação de Strings etc.
Unused code	Regras que detectam código não utilizado.	Campos privados, variáveis locais e parâmetros de métodos não usados.

Table 1. Alguns dos principais tipos de regras implementadas no PMD[PMD, 2018].

dados que permitem identificar e priorizar violações as quais correções sejam rápidas e precisas. O Fortify informações de análise que ajudam a fornecer softwares mais seguros, além de criar códigos de segurança, avaliações mais eficientes, consistentes e comple-

REGRA	DESCRIÇÃO
CN: Class implements Cloneable but does not define or use clone method	Implementa de classe Clonável, mas não define nem usa o método clone.
CN: Class defines clone() but doesn't implement Cloneable	Essa classe define um método clone (), mas a classe não implementa Cloneable. Existem algumas situações em que isso é aceitável (por exemplo, você quer controlar como as subclasses podem se clonar), mas apenas certifique-se de que é isso que você pretendia.
CN: clone method does not call super.clone()	Essa classe não final define um método clone () que não chama super.clone (). Se esta classe ("A") é estendida por uma subclasse ("B"), e a subclasse B chama super.clone (), então é provável que o método clone () de B retorne um objeto do tipo A, o que viola o contrato padrão para clone (). Se todos os métodos clone () chamarem super.clone (), eles terão a garantia de usar Object.clone (), que sempre retorna um objeto do tipo correto.
Eq: Class defines compareTo(...) and uses Object.equals()	Essa classe define um método compareTo (...), mas herda seu método equals () de java.lang.Object. Geralmente, o valor de compareTo deve retornar zero se e somente se igual for true. Se isso for violado, estranho e imprevisto
HE: Class defines equals() but not hashCode()	Classe substitui equals (Object), mas não substitui o hashCode (). Portanto, a classe pode violar a invariante de que objetos iguais devem ter códigos de hash iguais.
HE: Class defines hashCode() but not equals()	Essa classe define um método hashCode (), mas não um método equals (). Portanto, a classe pode violar a invariante de que objetos iguais devem ter códigos de hash iguais.

Table 2. Exemplo de regras implementadas no FINDBUGS[Pugh and Hovemeyer, 2006].

tas. Seu design permite que você incorpore, rapidamente, novas regras de segurança específicas de terceiros e de clientes.

Alguns pontos que o Fortify consegue identificar[OWASP, 2003]:

- **Mass Assignment: Insecure Binder Configuration**

Para facilitar o desenvolvimento e aumentar a produtividade, a maioria das estruturas modernas permite que um objeto seja automaticamente instanciado e preenchido com os parâmetros de solicitação HTTP, cujos nomes correspondem a um atributo da classe a ser vinculado. A instanciação automática e a população de objetos aceleram o desenvolvimento, mas podem levar a sérios problemas se implementados sem cautela. Qualquer atributo nas classes vinculadas, ou classes aninhadas, será automaticamente vinculado aos parâmetros de solicitação HTTP. Portanto, os usuários mal-intencionados poderão atribuir um valor a qualquer atributo em classes vinculadas ou aninhadas, mesmo que não sejam expostos ao cliente por meio de formulários da Web ou contratos de API.

- **Log Forging**

Os aplicativos geralmente usam arquivos de log para armazenar um histórico de eventos ou transações para revisão posterior, coleta de estatísticas ou depuração. Dependendo da natureza do aplicativo, a tarefa de revisar arquivos de log pode ser executada, manualmente, conforme necessário ou automatizada com uma ferramenta que seleciona automaticamente os logs para eventos importantes ou informações de tendências. A interpretação dos arquivos de log pode ser prejudicada ou mal direcionada se um invasor puder fornecer dados para o aplicativo que é posteriormente registrado na íntegra. No caso mais benigno, um invasor pode inserir entradas falsas no arquivo de log, fornecendo ao aplicativo uma entrada que inclua caracteres apropriados. Se o arquivo de log for processado automaticamente, o invasor poderá tornar o arquivo inutilizável, corrompendo o formato do arquivo ou injetando caracteres inesperados.

- **System Information Leak: Internal**

Dependendo da configuração do sistema, essas informações podem ser despejadas em um console, gravadas em um arquivo de log ou expostas a um usuário. Em alguns casos, a mensagem de erro informa ao invasor, com precisão, em que tipo de ataque o sistema está vulnerável. Por exemplo, uma mensagem de erro do banco de dados pode revelar que o aplicativo é vulnerável a um ataque de injeção de SQL. Outras mensagens de erro podem revelar pistas mais oblíquas sobre o sistema. No exemplo acima, a informação que vazou poderia implicar informações sobre o tipo de sistema operacional, os aplicativos instalados no sistema e a quantidade de cuidados que os administradores dedicaram à configuração do programa.

5. *Manage code quality*

Colocar toda a equipe sincronizada em relação a qualidade é bastante difícil. Para centralizar e escalar em uma visão única a qualidade do código, é necessário um local central para visualizar e definir as regras usadas durante a análise de projetos. Esses conjuntos de regras devem ser organizados, na qual os membros da organização podem ver quais regras são aplicadas ao projeto. O SonarQube[Atlassian, 2018a] fornece um Quality Gate padrão com foco na definição de requisitos únicos, mas aplicados em todos os projetos.

5.1. Sonar

O SonarQube[SonarSource, 2018] 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

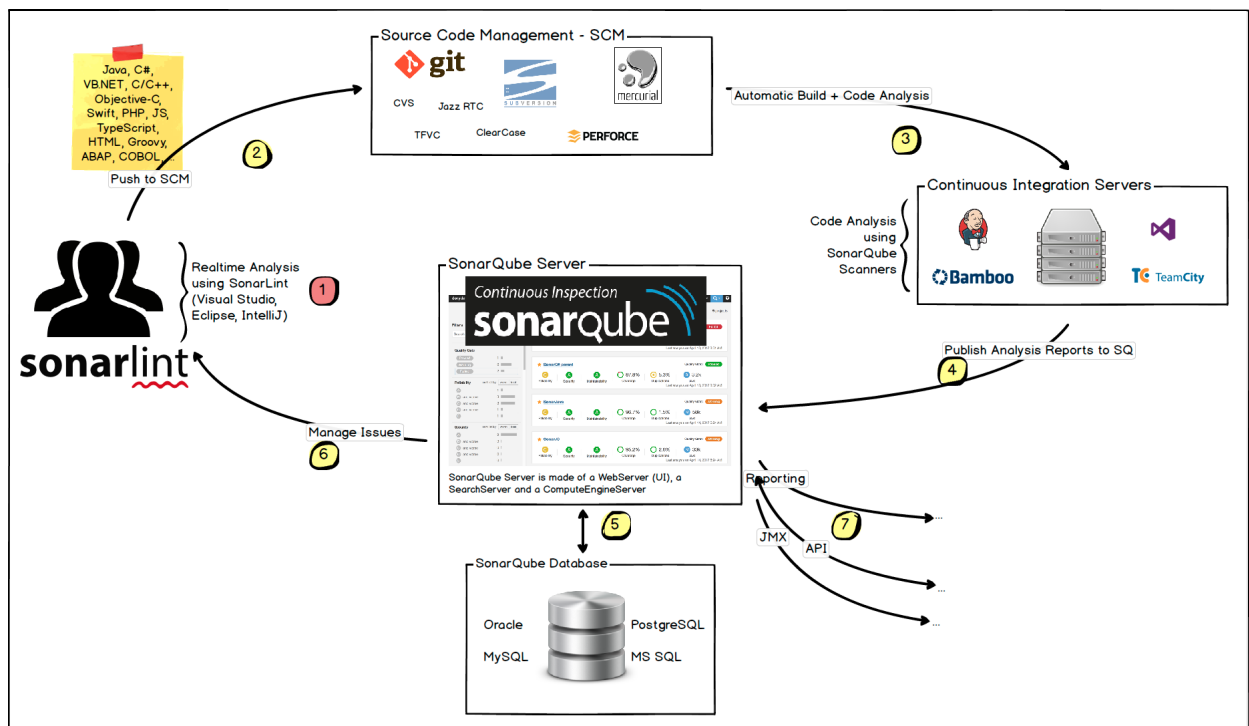


Figure 2. Arquitetura[Atlassian, 2018a] de integração do SonarQube e seus componentes.

idiomas por meio de conjuntos de regras integrados e também pode ser estendido com vários plug-ins, inclusive os citados nesse artigo. As principais vantagens e integrações do SonarQube são as seguintes [Campbell and Papapetrou, 2013]:

- SonarQube Plugins - Possui vários plugins importantes:
 - Findbugs.
 - Checkstyle.
 - PMD.
 - Fortify.
 - PI test.
- SonarQube Scanners - Possível de realizar a análise de várias formas:
 - Maven[Apache, 2018b].
 - Gradle[Apache, 2018c].
 - Jenkins[Jenkins, 2018].
 - Standalone / Ant[Apache, 2018a].

O SonarQube não compete de forma alguma com nenhuma das ferramentas de análise estática acima, mas complementa e funciona muito bem com essas ferramentas, sendo totalmente operante sem essas mesmas ferramentas de análise estática (Checkstyle, PMD e FindBugs).

6. Métricas de qualidade de código.

As métricas definidas foram avaliadas e consideradas com base no seguinte pontos importantes:

- O modelo de qualidade deve ser simples quanto possível.
- Bugs, vulnerabilidades e qualquer issue não devem se perder entre os problemas de manutenibilidade.
- Problemas de manutenção do código são importantes e não podem ser ignorados.

6.0.1. Métricas quantitativas.

Métricas quantitativas foram definidas e configuradas no Sonarqube no quality gate [Atlassian, 2018b]


Conditions


Only project measures are checked against thresholds. Sub-projects, directories and files are ignored.

Metric	Over Leak Period	Operator	Warning	Error		
Blocker Issues	<input type="checkbox"/>	is less than		0	Update	Delete
Classes	<input type="checkbox"/>	is less than		0	Update	Delete
Cognitive Complexity	<input type="checkbox"/>	is less than		0	Update	Delete
Coverage on New Code	Always	is less than		80	Update	Delete
Cyclomatic Complexity	<input type="checkbox"/>	is less than		5	Update	Delete
Duplicated Lines on New Code (%)	Always	is greater than		3	Update	Delete
Maintainability Rating on New Code	Always	is worse than		A X	Update	Delete
Major Issues	<input type="checkbox"/>	is less than		0	Update	Delete
Reliability Rating on New Code	Always	is worse than		A X	Update	Delete
Security Rating on New Code	Always	is worse than		A X	Update	Delete
Unit Test Success (%)	<input type="checkbox"/>	equals		100	Update	Delete


Figure 3. Quality Gates

Em caso de não conformidade, as mudanças não serão aprovadas para revisão manual.







 **Review required**
At least one approved review is required by reviewers with write access. [Learn more.](#)



[Add your review](#)



 **Some checks were not successful**
1 errored, 2 failing, and 1 successful checks

[Hide all checks](#)

  production-services/build-service:PR-linuxx86_64_sles12-linuxx86_64_sles12 — Pu... **Required** [Details](#)

  UXD-Jenkins-Prod/I18NCheck — Jenkins says FAILURE [Details](#)

  UXD-Jenkins-Prod/SONAR — SONAR - QUALITY GATE FAILURE **Required** [Details](#)

  UXD-Jenkins-Prod/WhiteSource — Jenkins says SUCCESS [Details](#)


 **Merging is blocked**
Merging can be performed automatically with one approved review.

Figure 4. Quality Gates

Part VII

Avaliação da solução proposta

A avaliação da solução proposta foi definida em 3 partes:

- Planejando uma estratégia e insights.
- Inspeção contínua.
- Utilizando SonarQube para conduzir revisões de código.

6.1. Planejando uma estratégia e insights.

O planejamento e atualização das estratégias de qualidade de código foram atualizadas, conforme a utilização do sonar e o desenvolvimento das aplicações. Ao final de cada Sprint, avaliações das métricas atuais e inclusão de novas métricas foram realizadas. Sendo a partir do Sprint 0, da utilização do sonar, até a normalização da qualidade do projeto, os objetivos foram definidos em:

- Não diminuição da qualidade atual do projeto.
- Aumento gradual do coverage de código legado.
- Resoluções de issues de nível BLOCKED.

6.1.1. Métricas quantitativas.

Regras, Quality Profiles e Quality Gates são as chave da plataforma SonarQube [Atlassian, 2018a], cada plugin no SonarQube executa uma específica análise de código estático, e nele contém um repositório com as descrições de cada regra, diagnóstico, e em muitos casos, sugestões de melhoria. Violações dessas regras são usadas para determinar e calcular a dívida técnica no projeto e no código, gerando um prévia avaliação com um determinado tempo, também calculado, para corrigir esses problemas.

Quality Gate é um indicador de conformidade (ou inconsistência) do projeto com as métricas de código de limite especificadas[Atlassian, 2018a]. Os projetos adicionados ao SonarQube usaram o gate de qualidade, e ao final desse trabalho as seguintes métricas e seus valores foram definidas:

- Quantidade de issues de qualidade críticas / bloqueantes : nenhuma.
- Horas de trabalho necessárias correção (debito técnico) : <12 horas.
- Tendência de qualidade : A.
- Cobertura de teste de unidade : >85.
- Complexidade por método : <8.
- Package tangle index (dependências cíclicas) : >8.
- Novas issues : 0.
- Novas vulnerabilidades : 0.
- Tolerância de taxa adicional de dívida técnica em código : <3%.

6.1.2. Evolução da qualidade.

Foi possível acompanhar e analisar visualmente a partir dos relatorios a evolução da qualidade:

A partir de uma overview dos projetos :

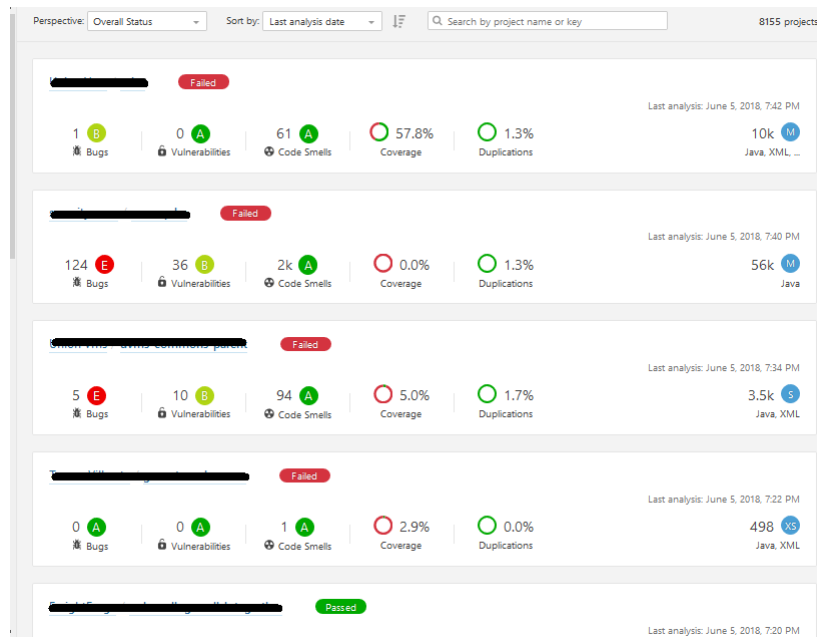


Figure 5. Overview dos projetos

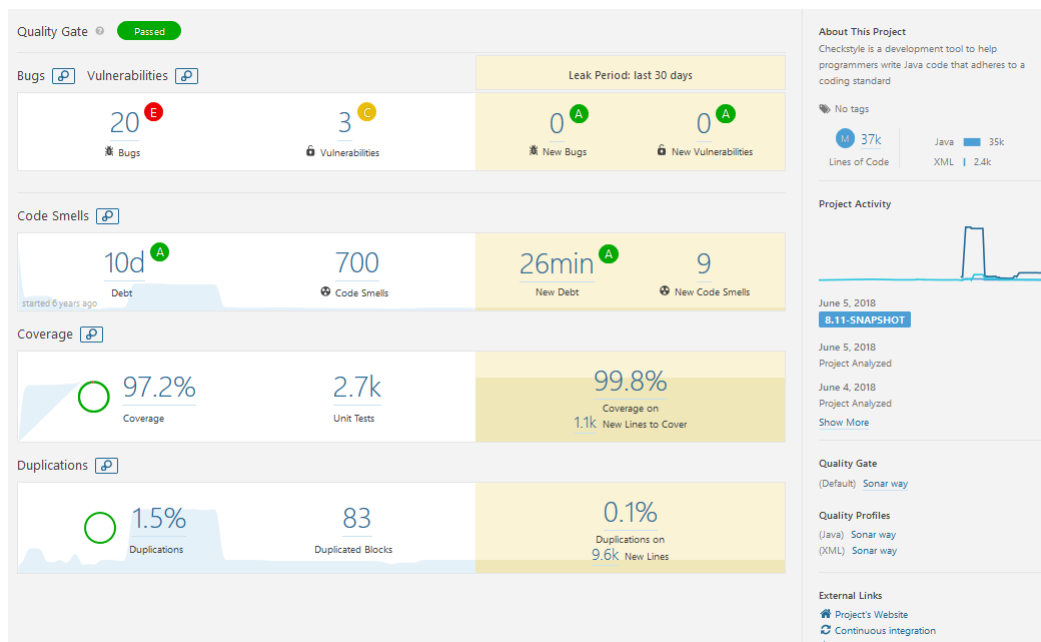


Figure 6. Exemplo 01 de historico de atividades

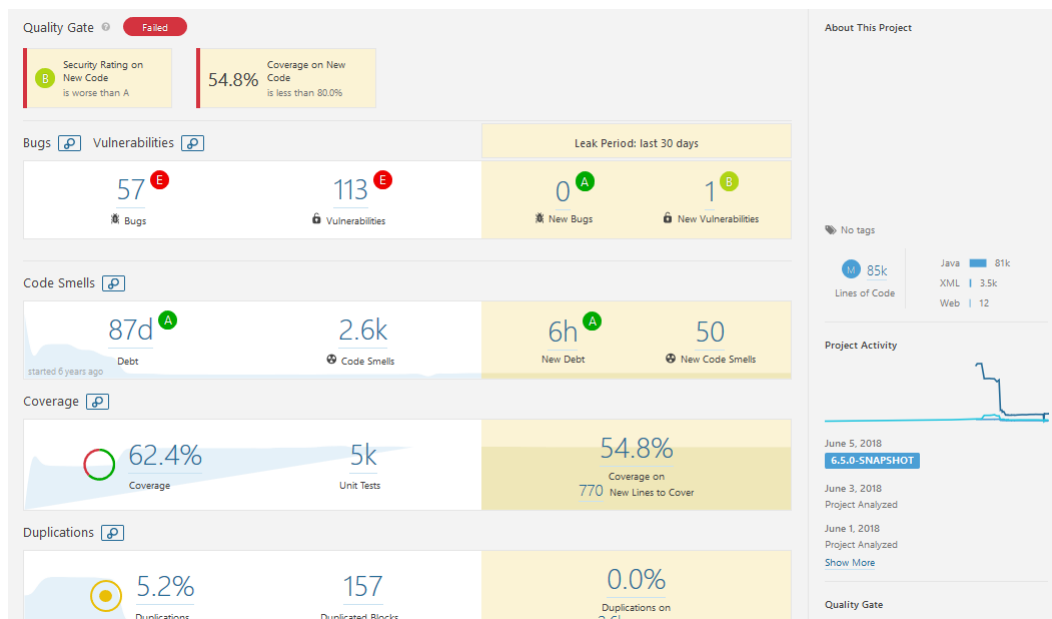


Figure 7. Exemplo 02 de histórico de atividades

6.2. SonarQube - Revisões de código.

SonarQube foi adicionado como ferramenta de revisão de código, sendo implementado e executado a cada nova mudança, e adicionado em cada novo "pull-request", e definido como critério de aceitação a aderência às regras definidas. Sendo possível de aliviar e eliminar muitas das revisões, com base em conceitos pessoais, ajudando o time a organizar e gerenciar esforços de qualidade de código, por meio da confirmação de problemas, comentários, atribuições e ajustes de gravidade.

Part VIII

Trabalhos futuros

Alguns trabalhos futuros podem ser realizados para melhorar a avaliação e entendimento das métricas apresentadas:

- Como pôde ser observado, houve poucos estudos empíricos e referências entre alguns princípios e métricas quantitativas, ou não houve ainda uma métrica sólida e confiável o suficiente para algumas das questões e regras:
 - Polymorphism - Métrica confiável para avaliação do grau de utilização de abstração.
 - SRP nas classes de testes, classe realizando diversos contextos de testes.
 - Law of Demeter principles.
- A conformidade em relação à arquitetura, muitos casos apresenta o problema de ser implementado não aderentes às definições de arquitetura, nas quais em muitos casos, a arquitetura não é refletida no código fonte.
 - Utilização de Classes de domínio (Entidades ORM) sendo utilizadas fora da camada de persistência, sendo propagada e expondo mais informações que o necessário.
 - Sobrecarga de representações em um único objeto (XML, JSON, DTO) não foram possíveis de serem quantificadas.
- Divergências entre a Arquitetura Conceitual (definida) e a Arquitetura Concreta (implementada).
- A necessidade de separação de testes unitário e testes de integração foi um ponto identificado no decorrer do desenvolvimento desse trabalho, tanto para análise de métricas, quanto para um melhor desenvolvimento e sua relação complementar (UT x IT), analisando o problema/anti-pattern conhecido, denominado ice-scream, quando não definido e definido esforço necessário para cada tipo de teste.

Pontos que podem ser avaliados e gerar uma proposta para definição de métrica quantitativas.

Part IX

Conclusão

Por meio do presente trabalho, obteve-se maior entendimento em relação à utilização de métricas no desenvolvimento de software. Houve um indício, de que, a utilização do TDD/BDD, ou de modo geral o test-first resultada em grandes ganhos, em relação a qualidade de código:

- Feedback rápido sobre a funcionalidade em desenvolvimento e em relação ao funcionalidades existentes no sistema
- Design com foco funcional (objetivos claro).

- Código mais limpo
- Realizar refatorações se torna um processo mais seguro.
- Segurança na correção de bugs (reproduzir o bug em relação a unidade afetada)
- Maior produtividade já que o desenvolvedor encontra menos bugs e não desperdiça tempo com depuradores
- Código da aplicação mais flexível, já que para escrever testes temos que separar em pequenos "pedaços" o nosso código, para que sejam testáveis, ou seja, nosso código estará menos acoplado.
- Desenvolve uma equipe mais confiante, pois um possível bug, ou impacto/modificação não planejada, já é mostrado pelos testes.
- Possibilidade de integração contínua, com builds automáticos e feedbacks rápidos de problemas.
- Utilização do TDD, facilita e favorece um coverage alto.

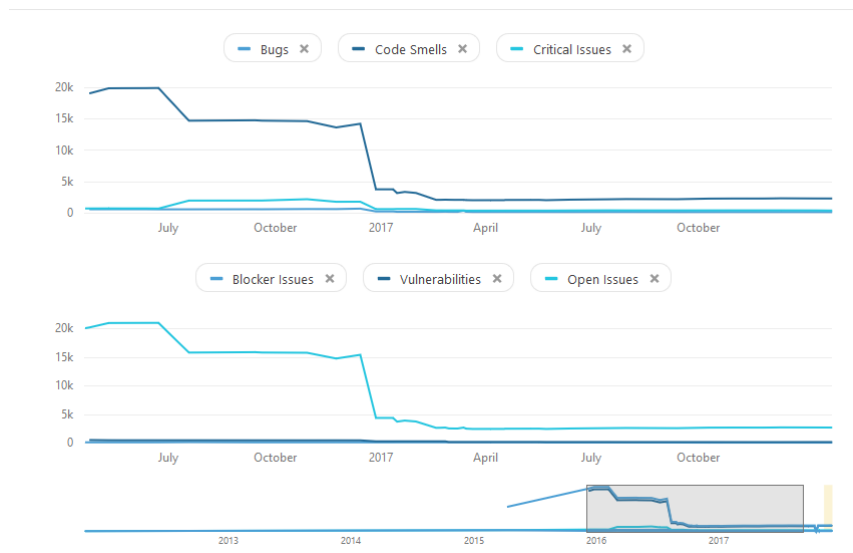


Figure 8. Quality activity history

Configuração das regras e a utilização do dashboard Sonar como guia, permite uma melhor codificação e centralização de definições :

Houve um grande ganho em relação a review, com a utilização do sonar, houve um maior aproveitamento das revisões manuais pelos seguintes motivos.

- Classes e métodos menor são mais rápidos de revisão.
- Utilização do relatórios de testes unitários e de integração como critério de entrega, proporcionou um maior controle em relação a confiabilidade do projeto.
- Houve um esforço menor com alteração em relação à mudança da formatação.
- Time teve um ganho significativo em conhecimento técnico com pouco esforço.

O ponto mais importante na qualidade contínua, incluindo a utilização do Sonar-Qube, é a produção de métricas de uma forma efetiva, criando um repositório confiável o qual não apenas fornece métricas e estatísticas sobre código, mas traduz valores não descritos em valores de negócios reais, como risco e dívida técnica. Não apenas abordando os desenvolvedores, mas também os gerentes de projeto e níveis administrativos ainda mais altos, devido ao aspecto de gerenciamento que o mesmo oferece, e sua visibilidade

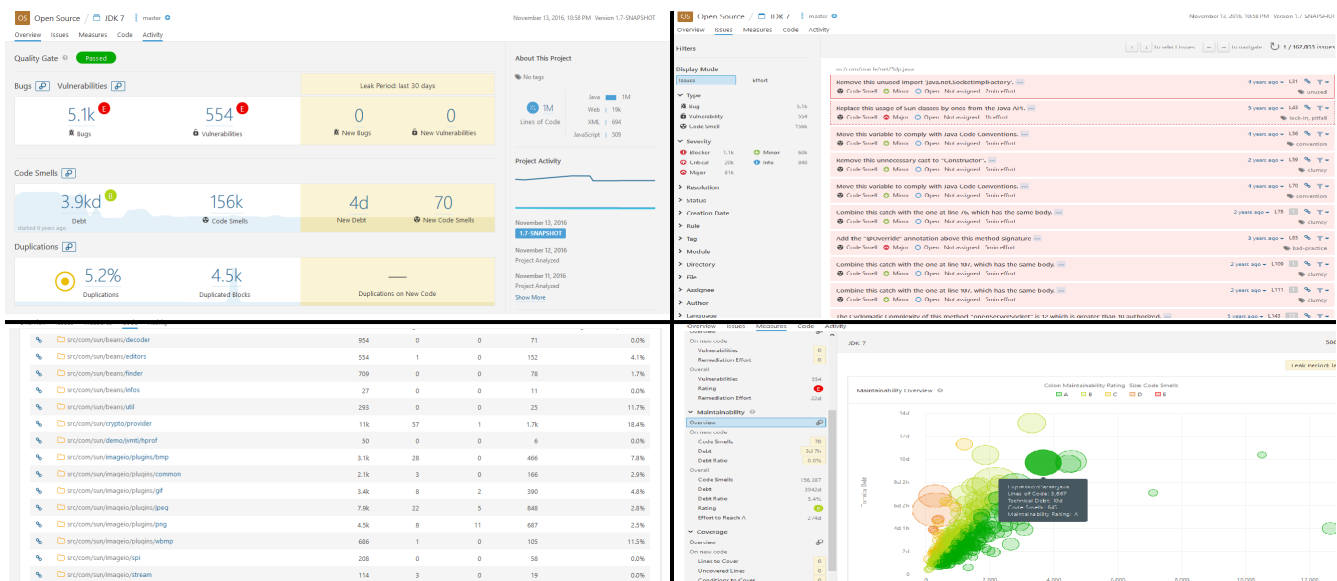


Figure 9. Dashboard do Sonar.

em relação a qualidade com seus relatórios, e principalmente pelas diferentes perspectivas, e de uma perspectiva gerencial, o acesso transparente e contínuo aos dados históricos permite que o gerente, e o time, analise e faça as perguntas certas, auxiliando na tomada de decisão no decorrer do desenvolvimento de software.

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)*.
- Apache (2008-2018b). Apache maven. <https://maven.apache.org/>.
- Apache (2018a). Apache antTM. <https://ant.apache.org/>.
- Apache (2018c). Gradle build tool. <https://gradle.org/>.
- Atlassian (2008-2018a). Sonarqube[®] software. <https://docs.sonarqube.org/display/SONAR/>.
- Atlassian (2008-2018b). Sonarqube[®] software. <https://docs.sonarqube.org/display/SONAR/Quality+Gates>.
- Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in Action*. MANNING.
- Chess, B. and West, J. (2007). *Secure Programming with Static Analysis*. Addison-Wesley Professional.
- 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/>.
- Jenkins (2018). Jenkins. <https://jenkins.io/>.
- 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.

MicroFocus (2003). Fortify inc. <https://software.microfocus.com/en-us/solutions/ap>

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-kanban>

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

OWASP, T. O. W. A. S. P. (2003). Fortify inc. <https://www.owasp.org/index.php>.

PMD (2018). Pmd - an extensible cross-language static code analyzer. <https://pmd.github.io/>.

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.

Pugh, B. and Hovemeyer, D. (2006). Findbugs™ - find bugs in java programs. <http://findbugs.sourceforge.net/>.

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 University - Silicon Valley Campus*.

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

SonarSource (2008-2018). Sonarqube® software. <https://www.sonarqube.org/>.

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)*.