

|  |
| --- |
| **­­­­­­­­­­­­­FACULDADE DE TECNOLOGIA**  **DE SÃO CAETANO DO SUL** |

**VINÍCIUS TEMÓTEO FERRARI**

**TEST-DRIVEN DEVELOPMENT COMO ALTERNATIVA DE GARANTIA DE QUALIDADE NO DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS**

**SÃO CAETANO DO SUL/SP  
2015**

**VINÍCIUS TEMÓTEO FERRARI**

**TEST-DRIVEN DEVELOPMENT COMO ALTERNATIVA DE GARANTIA DE QUALIDADE NO DESENVOLVIMENTO DE SOFTWARE**

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia de São Caetano do Sul, sob a orientação do Professor Msc. Wilson Vendramel, como requisito parcial para a obtenção do diploma de Graduação no Curso de Analise e Desenvolvimento de Sistemas

**SÃO CAETANO DO SUL/SP  
2015**

Dedico este trabalho aos desenvolvedores apaixonados por código e seu *design*.

AGRADECIMENTOS

Agradeço meu pai Reinaldo, minha mãe Amara e minha irmã Viviane por terem me proporcionado o suporte e aguentado todo meu estresse durante a conclusão desse trabalho.

Agradeço a meus amigos e parentes que ajudaram na revisão deste trabalho, e principalmente meu orientador Prof. Me. Wilson Vendramel, por ter me dado o suporte necessário durante todo o processo de desenvolvimento deste trabalho e não ter desistido de mim.

Qualquer tolo consegue escrever código que um computador entenda. Bons programadores conseguem escrever código que humanos conseguem entender.

Martin Fowler

RESUMO

FERRARI, Vinicius T. Test-Driven Development como alternativa de garantia de qualidade no desenvolvimento de software orientado a objetos. 64 f. Trabalho de Graduação – Faculdade de Tecnologia de São Caetano do Sul, São Caetano do Sul, 2015.

Com a intenção de facilitar tarefas repetitivas, demoradas ou que necessite de um grande volume de informações a serem processadas, foram criadas linguagens que computadores possam entender, porém escrever para os computadores e tornar a leitura de fácil entendimento para os desenvolvedores foi um desafio da engenharia de software. No processo de implementação de software é comum o programador cometer problemas de código e *design* devido a, por exemplo, falta de experiência na linguagem, excesso de informação ou pressão para que o prazo da entrega possa ser cumprido. Na tentativa de minimizar as más práticas, diferentes linguagens, paradigmas e técnicas de desenvolvimento são criadas. Este trabalho tem por objetivo mostrar que é possível alcançar qualidade através da técnica de Test-Driven Development (TDD) unido ao paradigma da orientação a objetos, com auxílio de um *framework* de testes na linguagem Java que irá guiar na utilização do TDD. Mostrar o processo do TDD aplicando um ciclo da técnica no desenvolvimento de uma aplicação Java Web, capaz de interpretar um texto marcado por caracteres especiais e gerar um arquivo ePub (livro digital). Avaliar o código escrito através de métricas de qualidade utilizando ferramentas de análise de qualidade, a fim de obter um *design* adequado, desacoplado, e com um bom alcance de cobertura de testes.

Palavras-chave: Teste; Métricas de qualidade; Design de código; Qualidade de software; TDD.

ABSTRACT

FERRARI, Vinicius T. Test-Driven Development como alternativa de garantia de qualidade no desenvolvimento de software orientado a objetos. 64 f. Trabalho de Graduação – Faculdade de Tecnologia de São Caetano do Sul, São Caetano do Sul, 2015.

In order to facilitate repetitive tasks, time consuming or requires a large volume of information to be processed, languages ​​have been created that computers can understand, but write to the computers and make reading easy to understand for developers was an engineering challenge software. In the software implementation process is common programmer commit code and design problems due to, for example, lack of experience in language, information overload or pressure on delivery date can be met. In an attempt to minimize the bad practices, different languages, paradigms and development techniques are created. This work aims to show what can be achieved through technical quality of Test-Driven Development (TDD) attached to the paradigm of object orientation with the aid of a testing framework in Java that will guide the use of TDD. Show the process of applying a TDD technique in the development cycle of a Java web application capable of interpreting a text marked by special characters and generate a file epub (digital book). Evaluate the written code through quality metrics using quality analysis tools in order to get a proper design, uncoupled, and with good test coverage range.

Keywords: Test; Quality metrics; Code design; Quality software; TDD.

LISTA DE ILUSTRAÇÕES

[Figura 1: Implementação interativa tradicional do algoritmo de Fibonacci 25](#_Toc419872276)

[Figura 2: Grafo de caminhos do algoritmo interativo de Fibonacci 26](#_Toc419872277)

[Figura 3: Implementação recursiva tradicional do algoritmo de Fibonacci 27](#_Toc419872278)

[Figura 4: Grafo de caminhos do algoritmo recursivo de Fibonacci 27](#_Toc419872279)

[Figura 5: Linhas de códigos de em Java 28](#_Toc419872280)

[Figura 6: Código com duplicação 30](#_Toc419872281)

[Figura 7: Código refatorados da Figura 6 31](#_Toc419872282)

[Figura 8: Mantra do TDD 32](#_Toc419872283)

[Figura 9: Teste não compila 34](#_Toc419872284)

[Figura 10: JUnit em modo gráfico junto da IDE NetBeans. 35](#_Toc419872285)

[Figura 11: Resultado dos testes após a implementação de *stubs*. 36](#_Toc419872286)

[Figura 12: Forçando o resultado de um teste 36](#_Toc419872287)

[Figura 13: Refatoração da classe Dollar e execução do teste. 38](#_Toc419872288)

[Figura 14: *Feedback* dos testes em relação ao código 39](#_Toc419872289)

[Figura 15: Lista de apoio 41](#_Toc419872290)

[Figura 16: Lista subdividida 42](#_Toc419872291)

[Figura 17: Primeiro teste do Ciclo de TDD. 43](#_Toc419872292)

[Figura 18: Erro de compilação 44](#_Toc419872293)

[Figura 19: Classe com método *stub* com retorno nullo 44](#_Toc419872294)

[Figura 20: IDE compilando a classe corretamente 45](#_Toc419872295)

[Figura 21: Erro de asserção 45](#_Toc419872296)

[Figura 22: Método *stub* com retorno da String esperada 46](#_Toc419872297)

[Figura 23: Teste passa 46](#_Toc419872298)

[Figura 24: Novo teste 47](#_Toc419872299)

[Figura 25: Erro de asserção 47](#_Toc419872300)

[Figura 26: Código funcional 48](#_Toc419872301)

[Figura 27: Passam todos os testes 48](#_Toc419872302)

[Figura 28: Refatoração do método para HTML 49](#_Toc419872303)

[Figura 29: Refatoração dos testes 49](#_Toc419872304)

[Figura 30: Execução de todos os testes do sistema 50](#_Toc419872305)

[Figura 31: Diagrama de classes. 52](#_Toc419872306)

[Figura 32: Implementação do Chain of Responsibility 53](#_Toc419872307)

[Figura 33: Dashboard da ferramenta de qualidade SonarQube 54](#_Toc419872308)

[Figura 34: Linhas de código do projeto 55](#_Toc419872309)

[Figura 35: Divida Técnica do projeto 56](#_Toc419872310)

[Figura 36:Sugestão de *issue*. 57](#_Toc419872311)

[Figura 37: Duplicidade de código no projeto 57](#_Toc419872312)

[Figura 38: Quadro de cobertura de testes 58](#_Toc419872313)

[Figura 39: Gráfico de cobertura. 59](#_Toc419872314)

[Figura 40: Quadro de complexidade 60](#_Toc419872315)

LISTA DE ABREVIATURAS E SIGLAS

CC – Complexidade Clicomática

CMMI - Capability Maturity Model - Integration

HML - Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

JSF – Java Server Faces

LOC – Line Of Code

QA – Quality Assurance

SQALE - Software Quality Assessment based on Lifecycle Expectations

SW-CMM – Software-Capability Maturity Model

SE-CMM – Systems Engineering-Capability Maturity Model

TDD - Test-driven development

UML - Unified Modeling Language

XML – eXtensible Markup Language

XP – eXtreme Programing

SUMÁRIO

[INTRODUÇÃO 13](#_Toc423189832)

[**1.** **FUNDAMENTAÇÃO TEÓRICA** 17](#_Toc423189833)

[**1.1.** **Qualidade** 17](#_Toc423189834)

[**1.2.** **Qualidade de produto de software** 23](#_Toc423189835)

[**1.3.** **Qualidade de código** 23](#_Toc423189836)

[**1.3.1.** **Complexidade Ciclomática (CC)** 24](#_Toc423189837)

[**1.3.2.** **Número de Linhas De Código (LOC – *Lines of Code*)** 28](#_Toc423189838)

[**1.3.3.** **Dívida Técnica (*Technical Debt*)** 28](#_Toc423189839)

[**1.3.4.** **Cobertura de código.** 29](#_Toc423189840)

[**1.3.5.** **Duplicação de código.** 29](#_Toc423189841)

[**2.** **TDD – TEST-DRIVEN DEVELOPMENT** 32](#_Toc423189842)

[**2.1.** **JUnit: *Framework* para testes automáticos** 33](#_Toc423189843)

[**2.2.** **Passos do TDD** 33](#_Toc423189844)

[**2.2.1.** **Vermelho: Criando um novo teste** 34](#_Toc423189845)

[**2.2.2.** **Verde: Fazendo o teste passar** 35](#_Toc423189846)

[**2.2.3.** **Refatorar: Generalizar o método** 37](#_Toc423189847)

[**2.2.4.** **Teste** 38](#_Toc423189848)

[***2.3.*** ***Baby-Steps*** 38](#_Toc423189849)

[**3.** **FERRAMENTA DE PUBLICAÇÃO DE LIVRO DIGITAL NO FORMATO EPUB: IMPLEMENTAÇÃO UTILIZANDO OS CONCEITOS DE TDD** 40](#_Toc423189850)

[**3.1.** **Configuração do projeto** 40](#_Toc423189851)

[**3.2.** **Ferramentas utilizadas** 40](#_Toc423189852)

[**3.3.** **Demonstração do TDD no desenvolvimento do módulo** 40](#_Toc423189853)

[**3.3.1.** **Criar listas de apoio** 41](#_Toc423189854)

[**3.3.2.** **Passo 1: Teste falha (vermelho)** 42](#_Toc423189855)

[**3.3.3.** **Passo 2: Teste passa (verde)** 45](#_Toc423189856)

[**4.1.** **Diagrama de classes** 51](#_Toc423189857)

[**4.2.** **Análise de qualidade com a ferramenta SonarQube** 51](#_Toc423189858)

[**4.2.1.** **Linhas de Código (LOC)** 52](#_Toc423189859)

[**4.2.2.** **Dívida Técnica (*Technical Debt*)** 52](#_Toc423189860)

[**4.2.3.** **Duplicação de Código** 52](#_Toc423189861)

[**4.2.4.** **Cobertura** 52](#_Toc423189862)

[**4.2.5.** **Complexidade Ciclomática (CC)** 52](#_Toc423189863)

[**CONSIDERAÇÕES FINAIS** 52](#_Toc423189864)

[**REFERÊNCIAS** 54](#_Toc423189865)

**INTRODUÇÃO**

O desenvolvimento de software passou por incríveis mudanças desde o início da era da informática, o avanço de linguagens e formas de como trabalhar a favor das linguagens e as novas metodologias trouxeram reuso e reaproveitamento de linhas de código, transformou a linguagem estruturada, que era difícil de se compreender, e também deu suporte a sistemas antigos ou sistemas de outros programadores, trazendo ao mundo real o modelo de objetos.

Os sistemas orientados a objetos deveriam ser mais fáceis de entender, ler, dar manutenção, porém problemas parecidos com o paradigma estruturado tornaram a aparecer como: códigos extensos, inconsistentes e duplicados, obrigando especialistas a criar maneiras de tornar os sistemas padrões mais fáceis de serem modificados e compreendidos, criando linguagens de modelagem como a *Unified Modeling Language* (UML) e até ferramentas para melhorar a qualidade no decorrer do desenvolvimento do software. Mesmo assim o tempo, recursos e custos despendidos em direção a qualidade ainda é grande, e há o risco de no final o projeto tornar-se inviavel.

Estudiosos de engenharia de software buscam diminuir o risco tanto da obsolescência quanto dos altos custos e tempo. Pensando nisso, Beck (2010) defende o Desenvolvimento Guiado por Testes, do inglês Test-Driven Developement (TDD).

Resumidamente, o TDD guia por meio de pequenos passos para que se alcance o todo, que é a conclusão do projeto, baseando-se em especificações que foram previamente definidas. Dessa forma, caso precise fazer qualquer mudança, basta rodar os testes e assim garantir que o todo continue a funcionar.

TDD não é somente isso, tem muito mais a agregar e a colaborar com a engenharia de software, entretanto, mesmo com a popularização do TDD. A técnica é ainda subutilizada pelos desenvolvedores, já que ela não é somente uma técnica de desenvolvimento, mas também uma técnica de projeto. (GUERRA ,2012)

Beck (2010), apresenta um conjunto de técnicas com uma linguagem acessível para qualquer engenheiro de software, com a intenção de encorajar a aplicabilidade de TDD em qualquer projeto e assim proporcionar oportunidades para que os profissionais do ramo possam trabalhar mais próximos do potencial real.

Com base na importância e colaboração que o TDD pode agregar à engenharia de software, esta pesquisa apresenta uma aplicação de TDD por meio de um exemplo prático.

Este trabalho tem por objetivo desenvolver uma aplicação Web aplicando Test-Driven Development (TDD) como alternativa de garantia de qualidade no desenvolvimento de software. A aplicação deve ser capaz de interpretar texto sinalizado com caracteres especiais para marcar e transformar em um arquivo e-Pub (e-book, livro digital), além de estudar as métricas de qualidade na produção da aplicação, garantir a legibilidade do código produzido, analisar a tolerância à mudança de requisitos do sistema, e também a influência do TDD no *design* do projeto.

O desenvolvimento de um software é algo complexo e delicado, mesmo compondo uma significante parcela do nosso dia-a-dia. Profissionais de todos os ramos munem-se de aplicações de software para facilitar atividades profissionais. À medida que a complexidade do problema aumenta, a qualidade tende a diminuir, isso porque é difícil documentar todas as necessidades. Com isso surgiu uma revolução nos métodos de desenvolvimento por meio do manifesto ágil, que aconteceu em 2001. Foram definidos vários padrões tanto de modelagem quanto de desenvolvimento ou, como também métodos de melhores práticas para tentar solucionar os principais problemas de qualidade de software e entre produtos de software que satisfazem os *stakeholders*. (BECK et al., 2001)

Crispin (2006) mostra que durante um processo de desenvolvimento de software, a qualidade e a satisfação dos *stakeholders*, são garantidas ao utilizar o desenvolvimento ágil somado à técnica de TDD.

*Of course, defect rate is just one small measure of quality. If you could talk to our business stakeholders, you’d find they’re satisfied, often even delighted, that we deliver just what they asked for each iteration. We’ve implemented features that competing companies thought were too complex to automate. Many factors come into play, including other agile practices such as short iterations, continuous integration, and refactoring. However, I think our ability to understand and capture business requirements, and then code to meet them, is a critical reason for our success.* (CRISPIN, 2006, p. 71)

Claro, a taxa de defeito é apenas uma pequena medida de qualidade. Se você pudesse falar com nossos *stakeholders* do negócio, você acharia que eles estão satisfeitos, muitas vezes até, encantados, que nós entregamos exatamente o que eles pediram para cada iteração. Nós implementamos características que as empresas concorrentes pensavam que eram demasiado complexas para automatizar. Muitos fatores entram em jogo, incluindo outras práticas ágeis como iterações curtas, integração contínua e refatoração. No entanto, eu acho que a nossa capacidade de compreender e capturar os requisitos de negócio e, em seguida, o código para encontrá-los, é uma razão fundamental para o nosso sucesso. (Tradução nossa).

Após o manifesto ágil, métodos de desenvolvimento foram criados como Scrum e eXtreme Programing (XP), que tentam estabelecer parâmetros de qualidade e permitem que o desenvolvimento seja versátil e fácil de ser modificado, passiveis à mudança de requisitos. Partindo do XP, o TDD é só um pedaço do *framework* e por esse motivo pode ser adaptado para qualquer modelo de desenvolvimento de software. (BECK et al., 2001)

Escrever código que possa ser compreendido por outro desenvolvedor ou por uma equipe de desenvolvimento é um desafio para a engenharia de software e o TDD tenta solucionar o problema obrigando cada unidade de código, cada método ou função a ser testada antes de ser alterada, impedindo falhas que podem ser ocasionadas durante uma refatoração.

Supondo por um momento que tal estilo de programação é possível, ainda pode-se reduzir significativamente a densidade de defeitos de código e fazer o tema de trabalho claro como cristal para todos os envolvidos. Se assim for, então escrever apenas aquele código que é requerido por testes que falham também tem implicações sociais. (BECK, 2010, p. x).

O TDD pode também diminuir o tempo de desenvolvimento do software, bem como dos testes, pois o tempo de depuração do código diminui devido à técnica aplicada e exigida, pois os passos de implementação são pequenos.

*If you like numbers, many studies have shown that TDD does improve quality. According to David Janzen’s paper, “Software Architecture Improvement through Test-Driven Development” (ACM Press, 2005), industry studies show that programmers using TDD produced code that passed between 18 and 50 percent more external test cases than code produced by control groups not using TDD. Additionally, the TDD programmers spent less time debugging. Janzen also cites academic studies that demonstrated significant improvements in external software quality and programmer productivity.* (CRISPIN, 2006, p. 71)

Se você gosta de números, muitos estudos têm mostrado que TDD faz melhorar a qualidade. De acordo com o artigo de David Janzen, "Software Improvement Arquitetura através de Test-Driven Development" (ACM Press, 2005), os estudos da indústria mostram que os programadores que usam TDD produzem código que passam entre 18 e 50 por cento mais em casos de teste externos do que código produzido por grupos de controle que não usam o TDD. Além disso, os programadores TDD gastam menos tempo de depuração. Janzen também cita estudos acadêmicos que demonstraram melhorias significativas na qualidade de software externo e produtividade do programador. (Tradução nossa)

Durante a programação, os desenvolvedores sentem-se mais seguros em seguir adiante, pois os passos anteriores estão cobertos por teste. A criação de novos códigos associados fica mais eficaz, devido aos testes de unidade garantirem que caso alguma alteração quebre as associações entre os métodos ou classes, os testes informarão rapidamente ao desenvolvedor ou ao time que está desenvolvendo o sistema de software, permitindo rápida correção.

O TDD obriga o desenvolvedor a atender as necessidades do produto, diminuindo a quantidade de implementações desnecessárias, ajuda o programador a escrever só o que será utilizado, para então prosseguir para a próxima implementação do sistema de software. Com testes já estabelecidos pelo programador, qualquer alteração no sistema que gere falha, é imediatamente capturada pela ferramenta de teste garantindo que o software funcione completamente na entrega do produto final, módulo ou de um protótipo do sistema.

Para tanto, o pesquisador adota a metodologia da pesquisa quantitativa de caráter exploratório, guiando-se por meio de livros, artigos idôneos, e utiliza imagens e vídeos para compor a apresentação teórica e prática de sua pesquisa acadêmica. (SEVERINO, 2007)

O projeto utiliza ferramentas livres de código aberto, como: Maven – ferramenta de *build* e dependência de códigos; Netbeans – IDE para desenvolvimento de código; Java Server Faces (JSF) – *framework* Web e também, é utilizado um *framework* para testes (JUnit), próprio para linguagem Java, além do SonarQube para análise de qualidade de código. O projeto foi desenvolvido linguagem de programação Java, guiado pelo TDD.

1. **FUNDAMENTAÇÃO TEÓRICA**

Este capítulo aborta os principais tópicos de qualidade que foram utilizados no trabalho.

* 1. **Qualidade**

Weinberg (1993) aponta a questão da relatividade quanto à qualidade, pois para ele é preciso examinar com muito cuidado porque definições como “a qualidade é a conformidade às exigências” se transformam em “às exigências de alguma pessoa” isto porque o mesmo produto detém aspectos de qualidade diferentes de acordo com cada ponto de vista. Assim, Weinberg (1993, p. 6) quanto às definições de qualidade de software ele se pergunta “Quem é a pessoa por trás dessa definição de qualidade?”

Weinberg (1993, p. 6) cita algumas “ideias familiares”, porém conflitantes sobre qualidade de software:

Defeito zero é a alta qualidade

a) Para os usuários cujo trabalho é afetado pelos defeitos.

b) Para os gerentes que são criticados pelos defeitos.

Ter um grande número de funções é alta qualidade

a) Para os usuários cujo trabalho pode tirar proveito dessas funções – se eles as conhecerem.

b) Para os distribuidores que acreditam que as funções vendem produtos.

Codificação elegante é alta qualidade

a) Para o pessoal de desenvolvimento que dá um grande valor às opiniões de seus colegas.

b) Para os professores de ciência da computação que apreciam elegância.

Alto desempenho é alta qualidade

a) Para os usuários cujo trabalho sobrecarrega a capacidade de suas máquinas.

b) Para o pessoal de venda que tem de submeter seus produtos à benchmarks.

Baixo custo de desenvolvimento é alta qualidade

a) Para os usuários cujo trabalho sobrecarrega a capacidade de suas máquinas.

b) Para os gerentes do projeto que estão com orçamentos apertados.

Desenvolvimento rápido é alta qualidade

a) Para os usuários cujo trabalho está esperando pelo software.

b) Para os distribuidores que desejam colonizar um mercado antes de seus concorrentes.

Facilidade para o usuário (userfriendliness) é alta qualidade

a) Para os usuários que gastam oito horas por ida na frente de uma tela utilizando o software.

b) Para os usuários que não conseguem se lembrar de detalhes de interface (WEINBERG, 1993, p. 6-7).

Weinberg (1993) ressalta aspectos importantes sobre qualidade por meio da conhecida frase “Mais qualidade para uma pessoa pode significar menos qualidade para outra”, e explica que se o objetivo de um projeto for Qualidade Total a opção a partir da definição citada, seria a de desenvolver um processo que abrangesse todos as pessoas relevantes, atribuir uma medida de qualidade para cada um, e a soma dessas medidas resultaria na qualidade total para cada abordagem diferente.

No entanto, Weinberg (1993) diz que “na prática, é claro, nenhum projeto de desenvolvimento de software utiliza sempre tal processo elaborado”, para ele isso não ocorre porque as maiorias das pessoas são previamente eliminadas por quem toma a decisão, o que o autor faz uma reflexão “Qual opinião sobre qualidade deve ser levada em conta quando da tomada de decisão?”

Para Weinberg (1993, p. 8), a tomada de decisão nem é um negócio racional, pois geralmente a decisão tem uma base emocional, daí a “definição de qualidade é sempre política e emocional”, e se justifica exemplificando que a decisão de um gerente de projeto sobre a qualidade de um processador de texto pode ser diferente se alguém de seu círculo emocional (familiar etc.) estiver envolvido.

Weinberg (1993, p. 8) “ela sempre envolve uma série de decisões sobre quais opiniões contam e quanto elas pesam umas em relação às outras”, porém estas decisões usualmente ficam “escondidas da visão do público”; e o fazem “[...] escondidas mesmo das mentes conscientes das pessoas que as tomam”.

A isso se soma a questão valor o quanto alguém está disposto a fazer (ou pagar) para que suas necessidades, requisitos sejam alcançados. Weinberg (1993) relata um episódio pessoal, que consistia na lembrança de seu pai jogando com ele Cribbage (do poeta sir John Suckling), um jogo pouco popular em algumas regiões o que dificultava encontrar parceiros para jogar. Entretanto, ficou muito feliz ao descobrir o programa de Cribbage chamado Precision Cribbage (Macintosh), o qual mesmo descobrindo dois *escores* errados no jogo ele ficou suficientemente satisfeito a ponto de jogar por horas e ainda pagar a taxa voluntária de distribuição.

Com este exemplo, Weinberg (1993) elucida a questão valor, o qual detém sua importância individual.

Além disso, há outros aspectos sobre qualidade, no caso, sobre padrões de opinião, com o tempo não se busca melhorar se determinado software já é superior que a concorrência, mesmo reconhecendo algum ou alguns erros no programa. Assim, se há pessoas que utilizam o produto e estão satisfeitas é decidido que tem boa qualidade; quanto aos erros, “todos os softwares têm erros (no mínimo, nós não podemos provar o contrário)”; e se há pessoas que o escolhe ao invés de comprar o da concorrência é porque detém qualidade superior (WEINBERG, 1993, p. 11).

Por isso que há pouca motivação para melhorar a qualidade, exceto quando fatores como esses se reverterem, daí poderão servir de motivação para a busca por melhoria na qualidade dos mesmos; ou seja, sairão da estagnação, até que sintam satisfeitos com o que foi produzido, e aí novamente a estagnação.

Entretanto, quando se trata de programa há outro fator que também contribui para a estagnação, ou simplesmente, porque há tanta desistência em melhorar algum programa: “Como reduzir defeitos de software para menos que um em um milhão de linhas de código?” (WEINBERG, 1993, p. 12).

Crosby (apud WEINBERG, 1993, p. 12) sobre melhorias em um programa, usualmente iniciam com um estudo do custo da qualidade (valor da qualidade), e geralmente os gerentes mantém o pensamento em cortar custos ou acelerar um cronograma, porém quanto ao “[...] valor da qualidade melhorada parece ser algo que eles nunca pensaram em medir”.

Weinberg (1993, p. 13) explica que “na medida em que uma organização começa a entender o real valor da qualidade, sua motivação para melhorar aumenta” e isso ocorre porque a partir daí isso o guiará a ter um melhor entendimento de como melhorar e quase consequentemente, o “[...] levará a um melhor entendimento do valor da qualidade”.

Existe a necessidade do software passar pelo teste do software, que consiste em uma investigação por meio de um processo da engenharia de software que envolve ações a fim de testar o bom funcionamento e a qualidade do mesmo.

Esse processo é realizado pelo testador de software que aplica ações que vão desde o levantamento de requisitos até a execução do teste propriamente dito, visto que o projeto a ser desenvolvido muitas vezes demanda muitas pessoas envolvidas, e muitas vezes possuem um volume significativo de estados, com fórmulas, atividades e algoritmos complexos, e todo esse universo o torna mais complexo, e mais difícil de garantir que o software funcione corretamente sem erros, daí a necessidade do testador de software.

Independentemente da metodologia de trabalho adotada, já visando obter o melhor produto final, tornou-se imprescindível a busca constante pela melhoria dos processos de engenharia de software. Assim, com o tempo tomou-se por hábito utilizar como base modelos sugeridos por entidades internacionais balizadas, como os conhecidos os SW-CMM, SE-CMM, ISO/IEC 15504 e o mais conhecido CMMI.

Mesmo assim, os testes de software, mesmo com os riscos e falhas de teste, ainda são vitais e indispensáveis, contudo recai em outra problemática, o testador de software, se for ineficiente ou agir de maneira subestimada, até por falta de tempo, recursos humanos ou financeiros, a etapa imprescindível pode não ser a resposta à garantia de qualidade do software.

Para Weinberg (1993, p. 1), “produzir e manter software não constitui uma série aleatória de eventos. Existem padrões, e esses padrões oferecem uma oportunidade de controlar nossos produtos, nossas organizações e nossas vidas”.

Brooks Jr. (2009) já disse que gerenciar um projeto de programação de computadores é como gerenciar qualquer outro empreendimento, porém o saber é cumulativo, e lida-se com uma arte que demanda muita criatividade, especialmente se baseada em boas técnicas.

Mas nem tudo são alegrias, há as tristezas, como ter de produzir perfeitamente, são os outros que definem os objetivos, fornecem os recursos e trazem a informação, a grande dependência de programas que foram desenvolvidos por outros profissionais que tendem a ser mal projetados, ou não tem código-fonte ou com documentação incompleta, a partir daí o programador tem que investir muitas horas corrigindo o que na teoria deveria estar pronto, disponível e utilizável; os longos momentos de horas melancólicas de trabalho monótono e cansativo (BROOKS JR, 2009, p. 8).

A pior das tristezas é a que as vezes, o produto que se investiu tanto tempo, esforços, recursos, parece obsoleto quando fica pronto, e seus concorrentes já estão buscando novas e melhores ideias (BROOKS JR, 2009, p. 8).

Pensando muito mais nos contra que nos prós, muitas pesquisas e estudos têm sido feitas em busca de encontrar meios para transformar parte desse processo em algo mais agradável e seguro, e especialmente diminuir as chances de se investir tempo e recursos em algo que pode ficar obsoleto, e se possível reduzir consideravelmente o tempo de produção e erros.

Martin (2011), descreve os princípios necessários para um código legível, elegante e eficiente.

Um código limpo é simples e direto. [...] tão bem legível quanto uma prosa bem escrita. [...] jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas (MARTIN apud BOOCH, 2011, p. 3)

Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes de unidade e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem uma API mínima e clara. O código deve ser inteligível já que dependendo da linguagem, nem toda informação necessária pode expressar no código em si (MARTIN apud THOMAS, 2011,p. 4)

Por que um código limpo?

Segundo Beck (2010, p. prefácio), há muitas razões para se investir em um código limpo, por ser uma forma previsível de desenvolver; sabe-se exatamente quando acabou; a certeza de que não ficou para trás uma trilha de erros; maior oportunidade de aprender com cada código fonte; melhorá-lo para o usuário e consequentemente suas vidas; melhor ambiente de trabalho; mais satisfação em escrevê-lo. Ao se reduzir a densidade de defeitos então a garantia da qualidade pode mudar de trabalho reativo para pró-ativo, e muitas outras boas razões.

Mas como obter um código limpo que funcione?

“Conduzindo o desenvolvimento com testes automatizados, um estilo de desenvolvimento chamado Desenvolvimento Guiado por Testes” (TDD) (2010, prefácio).

Beck (2010, p. prefácio) alerta que através do TDD, escreve-se o código novo apenas se um teste automatizado falhar; eliminando a duplicação.

Seguindo essas duas regras, Beck (2010) diz que isso irá gerar “um complexo comportamento individual e de grupo com implicações técnicas”, deve-se:

“[...] projetar organicamente com código, executando e fornecendo feedback entre as decisões; escrever seus próprios testes; o ambiente de desenvolvimento deve fornecer resposta rápida a pequenas mudanças; o projeto deve consistir em muitos componentes altamente coesos e fracamente acoplados para tornar os testes fáceis” (BECK, 2010, p. prefácio).

Assim se organizou com o esquema ‘vermelho/verde/refatorar’ que consiste em vermelho ‘escrever um pequeno teste que não funcione e que talvez nem compile inicialmente; verde ‘fazer rapidamente o teste funcionar, mesmo cometendo algum pecado necessário no processo; e refatorar significa eliminar todas as duplicidades criadas apenas para que o teste funcione”.

Mas apesar das inúmeras vantagens apontadas, pode ocorrer alguma resistência, engenheiros de software pode se perguntar por que escrever testes automatizados e assim dar passos tão pequenos, para Beck (2010, p. xi) o que torna possível, primeiramente, é a coragem.

Feathers (2013, p. 86) diz que “TDD é o recurso mais valioso e é a mais poderosa técnica de inclusão de recursos, […]”. “Ela permite concentrarmos em uma característica de cada vez”. (FEATHERS, p. 91)

Beck (2010, p. prefácio) explica que o medo convive com o programador e o faz hesitar, ele se comunica menos, o afasta do *feedback*, torna-o mal humorado, e o TDD possibilita transformar positivamente, ou seja, facilita inverter cada uma dessas situações.

Beck (2010) explica que através do TDD quando um teste está em funcionamento, ele está em funcionamento naquele momento e no futuro, e assim mais próximos de ter tudo funcionando, mas para tanto, ele sugere que “quanto mais árduo for o problema de programação menor deve ser o terreno que cada teste deve cobrir”.

* 1. **Qualidade de produto de software**

Na área de Garantia da Qualidade, ou no inglês *Quality Assurance (*QA), que é totalmente voltada para garantir qualidade em produtos de software, órgãos como ISO, ABNT, entre outros, criam normas e especificam padrões de qualidade, atingir tais padrões é o desafio da engenharia de software desde o início das linguagens de programação.

Definir qualidade é um assunto tão difícil quanto tentar chegar nela. Especialistas do mundo todo debatem formas de conseguir atingir todos os pontos de qualidade, desde o código, até o suporte do mesmo. Modelos de negócio, gestão de áreas, tudo isso entra em conta, quando se quer medir qualidade, porém quando se pensa exatamente o que é necessário e o que é preciso, consegue-se estabelecer um parâmetro e então avaliar em qual nível o produto está.

* 1. **Qualidade de código**

Fowler (2007, p. 15) disse: “Qualquer tolo consegue escrever código que um computador entenda. Bons programadores conseguem escrever código que humanos conseguem entender”; essa frase é muito citada no mundo ágil e muitos desenvolvedores tentam a adotar, porém mesmo em seu livro, Fowler (2007) diz que se você precisa refatorar é essencial ter uma bateria sólida de testes, já que isso evita problemas de mudança de comportamento entre as refatorações.

Outro problema comum de código mal escrito é que as pessoas mais antigas da equipe que conhecem o código, são muitas vezes impedidos de se afastarem desse código, já que são os principais conhecedores dos mesmos, e quando uma nova funcionalidade difícil é pedida essa pessoa que é encarregada a introduzir o código no sistema. Esse cenário é comum em empresas de todo o mundo. (CAELUM, 2014, p. 51).

E para tentar assegurar qualidade foram criadas métricas de qualidade de código.

* + 1. **Complexidade Ciclomática (CC)**

Essa métrica de qualidade de software foi formulada por Thomas J. McCabe em 1976, e mede a quantidade de caminhos possíveis que um método, uma classe ou arquivo, contém e é calculado através de um grafo direcionado.

Em um de seus artigos McCabe (1976), questiona a necessidade de uma forma de medir a facilidade de manter e testar um método; ele questiona o fato de os softwares nessa época gastar valores de dinheiro consideráveis em manutenção e teste.

Para isso McCabe (1976), desenvolveu um teorema matemático guiado nos caminhos possíveis de um código; e quanto maior a quantidade de caminho, moires as condições a serem testadas e cobertas, além de uma necessidade maior de entendimento.

O teorema de forma resumida é a quantidade de setas do grafo representado pela letra e, menos a quantidade de nós que o algoritmo tem representado por n, mais 2 que resulta na complexidade ciclomática v(G).

v(G) = e – n + 2

Ou seja, o método de uma função de Fibonacci utilizando a implementação interativa tradicional visto na Figura 1

Figura 1: Implementação interativa tradicional do algoritmo de Fibonacci



Fonte: Adaptado de Gonçalves, 2012

Quando transformado em grafo de caminhos (Figura 2), que o código pode seguir.

Figura 2: Grafo de caminhos do algoritmo interativo de Fibonacci



Fonte: Adaptado de Gonçalves, 2012

Tem uma Complexidade Ciclomática de 4, dada a formula v(G) = 9 – 7 + 2.

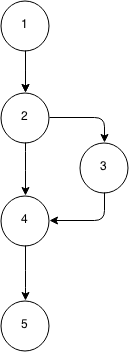
Figura 3: Implementação recursiva tradicional do algoritmo de Fibonacci



Fonte: Adaptado de Gonçalves, 2012

Já o mesmo programa o da figura 2 tem a Complexidade Ciclomática de 2 dada a formula v(G) = e – n + p como v(G) = .5 – 5 + 2. Que é representada em forma de grafo pela figura 4

Figura 4: Grafo de caminhos do algoritmo recursivo de Fibonacci



Fonte: Adaptado de Gonçalves, 2012

* + 1. **Número de Linhas De Código (LOC – *Lines of Code*)**

São as Linhas de Código efetivas em um sistema.

Considerada a medida mais simples e também a menos precisa, já que não existem distinção de linhas em branco, comentários, e código, a métrica de Linhas De Código (LOC) serve como referência para identificar o tamanho do projeto, pois, um programa de 4 milhões de linhas tende a ser mais complexo que um programa de 4 mil linhas. (KOSCIANSKI, 2007, p. 229).

Em seu exemplo Koscianski (2007), mostra que a Figura 5 tem três linhas, sendo três lógicas e duas físicas

Figura 5: Linhas de códigos de em Java



Fonte: Adaptado de Koscianski, 2007

* + 1. **Dívida Técnica (*Technical Debt*)**

O conceito de dívida técnica surgiu através de uma metáfora feita por Cunningham (2014).

*In this metaphor, doing things the quick and dirty way sets us up with a technical debt, which is similar to a financial debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice (*FOWLER, 2003)

Nesta metáfora, fazendo as coisas da maneira rápida e suja nos leva à uma dívida técnica, que é semelhante a uma dívida financeira. Como uma dívida financeira, a dívida técnica incorre em pagamentos de juros, que vêm sob a forma de o esforço extra que temos de fazer no futuro do desenvolvimento devido à escolha rápida e suja de design. (Tradução nossa)

Pouco depois que Cunningham fez essa metáfora, a comunidade de desenvolvimento de software passou a evoluir o conceito, considerando a dívida técnica um dos grandes aspectos de qualidade, já que um software com baixa dívida técnica tende a ser menos difícil de entender, mais seguro de modificar, exige menor tempo de manutenção, que consequentemente diminui o custo do projeto. (GOMES ,2014, p. 88)

* + 1. **Cobertura de código.**

Essa métrica é usada para medir a quantidade de código que foi testada ou deixou de ser normalmente expressa em porcentagem, ou em linhas de código lógico.

Uma cobertura de código de 100% indica que todo código de produção tem ao menos um teste passando por ele. (ANICHE, 2014, p.139)

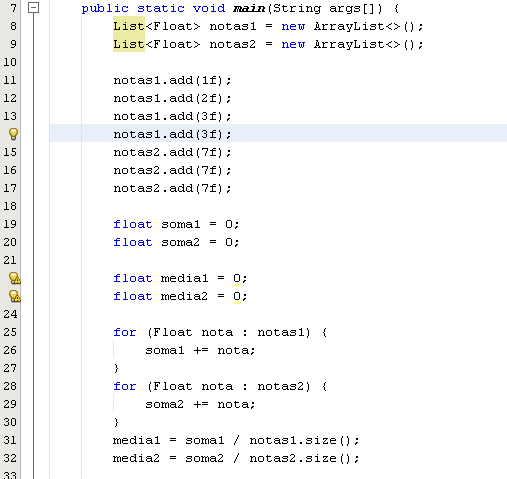
* + 1. **Duplicação de código.**

É o termo usado para o código que foi repetido em mais de um lugar, ele pode ser o mesmo código de outro trecho, pode ser um código parecido mais com uma variável diferente, no entanto o código duplicado é considerado por Fowler (2007), o principal problema de um software, e se isso ocorre é essencial encontrar uma maneira de unificá-los, pois, diminui assim a complexidade do sistema, e sua dívida técnica.

Uma maneira de remover código duplicado é introduzir um método ou uma classe, e passar suas dependências por parâmetro.

Por exemplo o código expresso na Figura 6 pode ser representado por um único método e chamado duas vezes

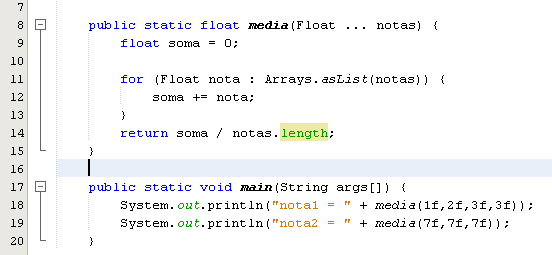
Figura 6: Código com duplicação



Fonte: Adaptado de Fowler, 2007

Quando o método é refatorado o novo código fica como na figura 7:

Figura 7: Código refatorados da Figura 6



Fonte: Adaptado de Fowler, 2007

1. **TDD – TEST-DRIVEN DEVELOPMENT**

Test-Driven Development (TDD) é uma técnica de programação criada por um membro do manifesto ágil Kent Beck, que extraiu do modelo de desenvolvimento ágil XP, também criado por ele. A técnica é simples e muitas vezes repetitiva já que são necessários somente três passos para começar a aplicar a prática. Beck (2010, p. prefácio) define os passos como sendo:

1 - Vermelho – Escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente.

2 - Verde – Fazer rapidamente o teste funcionar, mesmo contendo algum pecado necessário inicialmente.

3 - Refatorar – Eliminar todas as duplicatas criadas apenas para que o teste funcione. (BECK, 2010).

No meio informal da informática é considerado um mantra tais passos para a programação, a Figura 8 mostra a relação das palavras citadas por Beck(2010) com o objetivo de cada passo.

Figura 8: Mantra do TDD



Fonte Diogoosorio, 2012

* 1. **JUnit: *Framework* para testes automáticos**

Uma ferramenta Java para se desenvolver em TDD é o JUnit um *framework* para automatizar testes de unidade em softwares desenvolvidos em Java, esta ferramenta permite a execução de grandes quantidades de testes, de modo textual ou modo gráfico.

Massol (2005) descrever, de maneira simples o que é um *framework* e como o JUnit se enquadra em um *framework*.

Um *framework* é um aplicativo semicompleto. Um *framework* fornece uma estrutura reutilizável e comum que pode ser compartilhada entre aplicativos. Os desenvolvedores incorporam o *framework* em seu próprio aplicativo e o estendem até que eles cumpram suas necessidades especificas. *Frameworks* diferem de kits de ferramentas porque eles fornecem uma estrutura coerente, ao invés de um conjunto de classes utilitárias. (MASSOL, 2005, p.5).

Para Guerra (2012), um *framework*, não deve somente proporcionar o reuso do código, mas também o reuso de sua estrutura e de seu *design*, provendo o reaproveitamento de funcionalidades, assim acelerando o ritmo do desenvolvimento e direcionando o desenvolvedor a utilizar as boas práticas de programação.

Produzir enquanto roda uma bateria de teste é mais uma facilidade que o *framework* JUnit pode proporcionar, ele foi escrito em código aberto e pode ser visualizado todo o comportamento e modificado quando necessário.

Não existem dificuldades de utilização já que pode ser criado pacotes de teste em separado, a implementação de uma interface garante todas as funcionalidades de teste, coso o usuário necessite de mais funcionalidades, existem API’s que podem criar objetos fictícios de conexão com banco ou de acesso a outros serviços, existem também ferramenta para verificar comportamento de requisições HTTPs.

* 1. **Passos do TDD**

O processo de desenvolvimento do TDD sugerido por Beck (2010) envolve três passos, sendo eles:

1. Ver o teste falhar (Vermelho);
2. Ver o teste passar (Verde);
3. Ver refatorar tanto os testes quanto o código de produção (Refatorar).
   * 1. **Vermelho: Criando um novo teste**

O primeiro passo, o software ainda não existe e deve-se criar o primeiro teste unitário para resolver o problema proposto, aqui, o programador coloca exatamente o que a unidade do programa deve realizar, e depois que especificado, o teste deve-se rodar o *framework* de teste e esperar o retorno de falha.

A figura 9 mostra o trecho de código extraído de Beck (2010, p. 17) demonstrando a técnica na pratica, o exemplo que ele propõe é um sistema de gerenciamento de investimento de títulos, que uma companhia vende. Neste exemplo o sistema já existe somente as novas modificações que estão sendo feitas estão utilizando TDD.

Figura 9: Teste não compila



Fonte: Adaptado de Beck, 2010.

A Figura 10 mostra o *framework* JUnit exibindo os resultados do teste da Figura 1 de forma gráfica.

Figura 10: JUnit em modo gráfico junto da IDE NetBeans.



Fonte: Adaptado de Beck, 2010.

* + 1. **Verde: Fazendo o teste passar**

O próximo passo é resolver o problema dos erros, o *framework* JUnit vai mostrar que está faltando a classe Dollar, porém ainda existem mais três erros para que o teste passe a funcionar, o correto seria corrigir um erro, rodar os testes, capturar mais um erro e corrigir um de cada vez, entretanto como no exemplo de Beck o programador pode, caso esteja seguro dos próximos passos, ir um pouco além e corrigir alguns erros a mais, contudo nada além da necessidade de passar nos teste, mesmo que implique em métodos retornando valores falsos ou *stubs*.

Nesse passo foi implementado o mínimo necessário para que o compilador passe a responder de forma aceitável, resolvendo os erros de compilação e devolvendo pelo menos algum valor de retorno. A figura 11 mostra as linhas adicionadas e o resultado do teste depois da execução.

Figura 11: Resultado dos testes após a implementação de *stubs*.



Fonte: Adaptado de Beck, 2010.

Agora existe um programa compilando, contudo ele não devolve o retorno esperado, quando Beck roda este teste em seu livro, ele afirma que temos progresso no desenvolvimento do software, e sugere a menor implementação possível para que possa fazer o teste passar. Como alteração modifica-se a linha 3 para “public int amount = 10;” obrigando o teste a funcionar, como mostra a figura 12.

Figura 12: forçando o resultado de um teste



Fonte: Adaptado de Beck, 2010.

Com isso pode-se prosseguir para o passo final do ciclo do TDD a refatoração.

* + 1. **Refatorar: Generalizar o método**

O último passo do “mantra do TDD” é a refatoração, Refatorar o código produzido é um passo delicado, pois é aqui que vamos garantir a qualidade do código ou o fracasso de entendimento do mesmo, em um livro específico de refatoração de Martin Fowler, com ajuda de Kent Beck (2004), descrevem as melhores práticas de refatoração, entretanto logo no prefacio Fowler (2007) descreve a dificuldade de expor em uma introdução o tema e resolve pular direto para exemplos práticos de refatoração.

O problema do software WyCash consiste em que ainda existe poucas coisas a serem mudadas no novo módulo a ser implementado por isso a refatoração necessária para chegar a um design razoável requer somente mais uma execução do teste. As principais alterações são: remoção da constante 10; implementação do construtor; implementação do método times. A maneira correta de resolver os problemas de refatoração é refatorar gradualmente, porém este exemplo e simples o suficiente para dar passos maiores. A figura 6 mostra o código após a refatoração e o resultado do teste.

Refatoração é uma palavra muito utilizada no campo da informática, Fowler (2013) em seu site publicou a definição de refatoração como sendo, “*Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*“, ou na tradução acessada pelo wikipedia “Refatoração é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.”.

Essa técnica tem por objetivo melhorar, além da legibilidade e qualidade no design, como na escalabilidade, desempenho etc. Refatorar código legado é um desafio, muito comum que programadores muitas vezes passam, e com a ausência de código de teste, a chance de fracasso é muito maior. Fowler (2004, p. 82) em seu livro observa dizendo, “Se você quiser refatorar, a pré-condição essencial é ter testes sólidos.”.

Figura 13: Refatoração da classe Dollar e execução do teste.



Fonte: Adaptado de Beck, 2010.

Com esse passo completo, é possível agora voltar ao passo um e prosseguir, adicionando novos testes para estes métodos ou criar um novo teste ou novas funcionalidades para o sistema.

* + 1. **Teste**

Existem várias formas de teste de sistemas variando de testes funcionais a testes estruturais, ou também conhecidos por testes de caixa preta a testes de caixa branca. Cada ambiente de teste visa garantir a captura de falha em cada etapa de desenvolvimento, os testes podem variar de testes automáticos a testes de usuários.

* 1. ***Baby-Steps***

*Baby-Steps* é o termo usado quando o programador é guiado pelo TDD durante a codificação, esse termo significa Passos de Bebês, pois, bebês dão passo pequenos ao caminhar. Isso exemplifica o tanto de modificações que o desenvolvedor deve fazer em cada interação, e assim avançar gradativamente.

Dessa forma o *Baby-Steps* guia através de pequenas modificações de código levando a uma solução funcional e objetiva, não fugindo da implementação mais genérica e abstrata atendendo de forma mais abrangente os testes já feitos

Aniche (2012, p. 38) relata que “O desenvolvedor deve buscar pela solução mais simples, e não pela modificação mais simples. Veja que a modificação mais simples não é necessariamente a solução mais simples”.

* 1. **Feedbacks dos testes**

Aniche (2012, p. 38), diz que o *feedback* que os testes podem trazer é igual se testados antes ou depois de implementado o código, porém se usado o TDD, o desenvolvedor irá receber *feedbacks* antes da implementação total do código, o que é um benefíco ao desenvolvedor, já que ele pode tomar medidas antes de terminar toda a codificação e reduzir o custo do tempo de espera para tomada de decisão, isso porque o tempo de espera é menor em relação a abordagem tradicional

A figura 14 expressa graficamente o efeito que o *feedback* traz em relação TDD versus a abordagem tradicional

Figura 14: Feedback dos testes em relação ao código



Fonte: Aniche, 2012

1. **FERRAMENTA DE PUBLICAÇÃO DE LIVRO DIGITAL NO FORMATO EPUB: IMPLEMENTAÇÃO UTILIZANDO OS CONCEITOS DE TDD**

Este projeto tem por objetivo exemplificar a prática do TDD, durante o processo de desenvolvimento do software, que será aplicado em uma ferramenta de publicação de livros digitais no formato ePub, porém o processo de transformação em arquivo físico, é parte de uma API de terceiros com código fonte aberto disponível em http://www.siegmann.nl/epublib.

A principal funcionalidade da ferramenta é mostrar a criação das páginas em formato texto e transformar em um formato que a API consiga interpretar e transformar no arquivo adequado para a publicação.

* 1. **Configuração do projeto**

É um projeto na linguagem de programação Java com a utilização de *frameworks* para desenvolvimento Web e uma API de terceiros para publicação no arquivo físico ePub

* 1. **Ferramentas utilizadas**
* Netbeans – Ambiente de desenvolvimento Java distribuído pela Oracle;
* JSF – *Framework* Web;
* Primefaces – *Framework* Web;
* Epublib – *Framework* ePub;
* JUnit - – *Framework* de testes;
* Maven – Ferramenta de *build*;
* Git – Controle de versão;
* Sonar – Ferramenta de análise de código.
  1. **Demonstração do TDD no desenvolvimento do módulo**

Esta seção demonstra como criar uma camada entre a API epub-lib e a camada de usuário, facilitando o processo entre entrada de texto pelo usuário e a transformação em HTML, formatado e especificado sob as regras ePub.

Também apresenta o processo de TDD na construção do módulo de transformação de texto em código e publicação do mesmo

* + 1. **Criar listas de apoio**

Como sugestão para controlar o processo de TDD, é possível fazer a criação de listas, figura 15, para facilitar a evolução dos testes e também o alcance dos objetivos do sistema

Figura 15: Lista de apoio



Fonte: Adaptado de Beck, 2010

As listas podem envolver maiores complexidades e podem ser quebradas em sub listas para facilitar o processo, como na figura 16.

Figura 16: Lista subdividida



Fonte: Adaptado de Beck, 2010

Esta lista possibilita evitar o problema de falta de planejamento e ajuda o início dos testes, de forma que o sistema não necessita de todo o processo de planejamento inicial, e não deixa o desenvolvedor sem os primeiros passos para o sistema

* + 1. **Passo 1: Teste falha (vermelho)**

Com a lista já minimamente planejada é fácil descobrir qual tarefa é a mais simples, assim é possível começar com o menor passo e menor teste. Os itens: transformar \*\* em <b> ; transformar \_\_ em <u>; transformar // em <i>, aparentam pertencer ao mesmo grupo de código, portanto é possível criar os testes para cada tipo e depois generalizar e remover as duplicações.

O primeiro teste mais simples e funcional que pode ser pensado é passar uma String por parâmetro contendo um dos símbolos da lista e esse símbolo devolver o retorno esperado, como na figura 17.

Figura 17: Primeiro teste do Ciclo de TDD.



Fonte: Figura Nossa

A figura 18 é executado o teste e o *framework* acusa um erro de compilação do sistema avisando que não existe a classe TransformaTag,

Figura 18: Erro de compilação



Fonte: Figura Nossa

Resolvido o problema da ausência da Classe; o teste é executado novamente e mais um problema de compilação ocorre. O método ainda não existe, e mais uma vez foi criado um método que sirva somente para consertar este erro, e por fim simplesmente retornamos o *null*

A figura 19 representa a classe com um método *stub* de retorno nulo para compilar o teste e assim prosseguir com o TDD.

Figura 19: Classe com método stub com retorno nullo



Fonte: Figura nossa

Com a classe Transforma Tag o teste passa a compilar normalmente. A figura 20 mostra a IDE sem erros de compilação.

Figura 20: IDE compilando a classe corretamente



Fonte: Figura nossa.

Quando é executado os testes por meio do *framework*, o teste acusa um problema de asserção. A figura 21, mostra que era esperado “<b>vinicius</b>”, porém, encontrou *null*.

Figura 21: Erro de asserção



Fonte: Figura nossa

* + 1. **Passo 2: Teste passa (verde)**

O processo de implementação deve ser o mais simples possível para atender o menor caso possível, visando atender de forma simples o primeiro teste criado.

Na figura 22 é retornado uma String com o resultado esperado pelo teste.

Figura 22: Método stub com retorno da String esperada



Fonte: Figura nossa

O teste é executado no *framework* com sucesso como na figura 23.

Figura 23: Teste passa



Fonte: Figura nossa

O próximo passo no ciclo seria o de refatoração, entretanto a implementação é pequena e não gerou nenhum tipo de dívida técnica no código.

No exemplo foi possível fazer o primeiro teste passar, porém ainda não é um teste com um código funcional que exerça algum valor ao projeto, sendo necessário criar um novo teste que falhe e que possa dar um novo caminho para o código, guiando a algo mais usual e mais reaproveitável. Para isso é feito primeiro um novo teste que falhe guiando a implementação mais adequada.

Figura 24: Novo teste



Fonte: Figura nossa

Assim que os testes são executados novamente, um dos métodos de teste passa o outro método falha, pois a implementação que foi utilizada não atende os dois casos de teste. A figura 25 mostra o erro de asserção

Figura 25: Erro de asserção



Fonte: Figura nossa

E assim pode-se criar um código simples e funcional que atenda os dois casos de teste. O código na figura 26 mostra a menor implementação funcional dos dois casos.

Figura 26: Código funcional



Fonte: Figura nossa

Assim que executado os testes, os dois casos são atendidos, como mostra a figura 27.

Figura 27: Passam todos os testes



Fonte: Figura nossa

Para concluir a primeira etapa pode-se nos satisfazer com o código criado e partir para uma nova bateria de testes que garanta algumas funcionalidades antes de refatorar e remover as duplicações, porém é necessário sempre iniciar com o teste falhando no processo de codificação.

* + 1. **Passo 3: Refatoração**

Durante a refatoração é importante rodar todos os testes do sistema independente da funcionalidade ou Classe que está sendo desenvolvida, para garantir que tudo está funcionando como antes; caso algum teste falhe é prioridade consertá-lo.

A refatoração mais adequada que existe no código a ser feita é a remoção da grande quantidade de *ifs* que foram criados para atender o método.

O ideal é utilizar as técnicas de Orientação a Objetos para remover tal duplicação, entretanto para o código exemplificado é fácil resolver utilizando a própria API String contida no núcleo da linguagem Java.

Para uma refatoração mais adequada, é necessário a introdução de dois novos parâmetros no método para HTML, assim é evitado a necessidade do condicional *if*, a figura 28 mostra a nova implementação.

Figura 28: Refatoração do método para HTML



Fonte: Figura nossa

A refatoração do método influenciou nas Classes de teste, por este motivo é necessário a refatoração dos testes para que eles atendam o novo *design* do método como na figura 29.

Figura 29: Refatoração dos testes



Fonte: Figura nossa

Após o processo de refatoração, é necessário executar todos os testes para garantir que nada tenha sido afetado durante o processo de refatoração.

A figura 30 mostra todos os testes do sistema, após a execução de uma quantidade considerável de ciclos de TDD. Depois que o projeto atingiu uma quantidade razoável de classes e de teste foi necessário focar no passo de refatoração para que o *design* fosse melhorado e a manutenibilidade garantida.

Figura 30: Execução de todos os testes do sistema



Fonte: Figura nossa

O Teste testControledDeveProcessarOconteudoSalvaEPUB\_EmC\_temp falha, pois como sugestão de Beck (2010) no desenvolvimento usando TDD em um projeto solo é importante saber o último passo dado.

Beck (apud Gabriel, Richard, 2010, p. 168) adotou essa técnica, e tornou-a um *pattern* de TDD, nomeado Broken Test, ou teste quebrado, que ajuda o desenvolvedor a localizar o último ciclo não implementado em seu projeto solo.

1. **RESULTADOS OBTIDOS**

Este capítulo mostra os resultados que o uso da técnica do TDD forneceu no decorrer do projeto, implementando a aplicação Web capaz de transformar texto em um arquivo de livro digital ePub.

* 1. **Diagrama de classes**

O TDD deixa o modelo de dados desacoplado e testável, assim é possível modificar o *design* do sistema de forma segura e com a garantia de que o sistema mantenha seu comportamento quando o processo de refatoração estiver concluído

A cada item que foi concluído da lista (Figura 16), foi aplicado um processo de refatoração focado no *design* e na remoção da duplicação de código. A figura 31 mostra o diagrama de classes do projeto.

Figura 31: Diagrama de classes.



Fonte: Imagem Nossa

Com o auxílio do TDD e do processo de refatoração, o *design* do projeto pode ser melhorado a ponto da inclusão de um *Design Pattern*, o Chain of Responsibility, responsável por desacoplar o processamento de funcionalidades do processador de texto para HTML

Com o *design pattern* Chain of Responsibility, é possível incluir novos tipos de processadores de texto, sem a necessidade de grandes alterações na estrutura do código, tornando-o fácil de utilizar, A interface Mark fornece a assinatura para a correta utilização.

A figura 32 mostra o diagrama do *Design Patter* Chain of Responsibility

Figura 32: Implementação do Chain of Responsibility



Fonte: Figura nossa

* 1. **Análise de qualidade com a ferramenta SonarQube**

Com auxílio da ferramenta SonarQube, foi feita uma análise de qualidade do código do projeto. As métricas analisadas são as mesmas abordadas no capítulo 1 sobre qualidade.

A figura 33 exibe a Dashboard do SonarQube

Figura 33: Dashboard da ferramenta de qualidade SonarQube



Fonte: Figura nossa

A Dashboard do SonarQube é simples e exibe de forma clara e sucinta as principais métricas de qualidade usadas no dia-a-dia do desenvolvimento de software. Para o trabalho foi mantida a configuração de métricas de qualidade padrão do SonarQube

A ferramenta possibilita a edição das métricas conforme necessário.

* + 1. **Linhas de Código (LOC)**

O primeiro quadro exposto na Dashboard é o de LOC que mesmo não sendo o maior fator de qualidade é um importante sinalizador do tamanho do projeto.

A figura 34 destaca o LOC do projeto.

Figura 34: Linhas de código do projeto



Fonte: Figura nossa

Para o SonarQube existem dois tipos de LOC, um é analisado somente o código Java excluindo comentários e linhas em branco, o outro que no quadro é sinalizado como Linhas, considera tanto comentários como também os espaços em branco, além do próprio código Java. O projeto quando analisado pela ferramenta informa a quantidade de LOC efetivas de código, sendo, 296 em código Java, e o projeto contém 14 arquivos, 14 classes e 10 diretórios, sendo o total de Linhas 394. A quantidade de métodos, instruções e Acessores não estão sendo levados em consideração neste trabalho. As linhas de código de testes não entram na análise.

* + 1. **Dívida Técnica (*Technical Debt*)**

A ferramenta de qualidade SonarQube faz uma verificação do código e identifica os possíveis pontos de dívida técnica e informa em duas partes separadas.

A primeiro classifica a pontuação total do projeto SQALE Rating de A à F e a porcentagem de dívida técnica total do projeto. Na segunda parte o SonarQube mostra com mais detalhes a Dívida Técnica e o tempo necessário para pagá-la, que normalmente exposta em minutos “min”, horas “h”, ou dias “d”. O SonarQube sugere também os pontos de melhoria, *issues*, que estão divididas em Muito alta, Alta, Média, Baixa e Muito baixa.

Na figura 35 estão expostos os dois quadros explicados.

Figura 35: Divida Técnica do projeto



Fonte: Figura nossa

Software Quality Assessment based on Lifecycle Expectations (SQALE) Rating é uma métrica desenvolvida por terceiros que ajuda na avaliação da qualidade no decorrer da evolução do projeto. Esta métrica junta vários fatores da dívida técnica para avaliar uma nota de A à E.

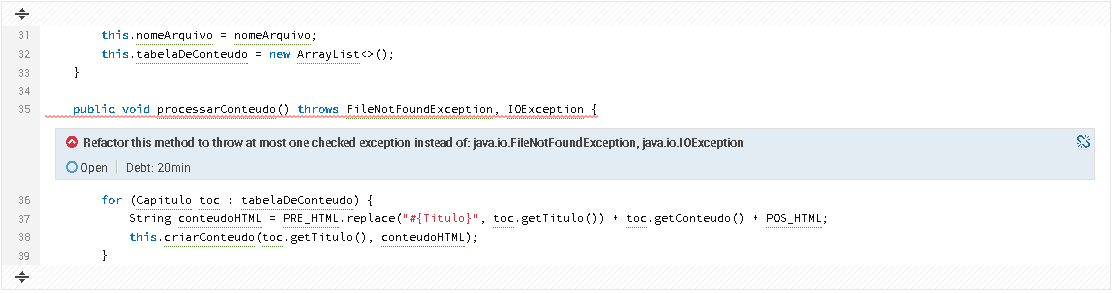
Technical Debt Ratio é a porcentagem geral de dívida técnica que o projeto analisado tem. Quanto mais alta a porcentagem, maiores são os problemas de *design* de código, Cobertura de código, entre outros.

Technical Debt é a quantidade de tempo aproximado para solucionar os problemas de *design* do projeto, Cobertura de código, entre outros.

*Issues* são os problemas encontrados pela ferramenta de qualidade e indicam principalmente os problemas relacionados a padronização do código. As *issues* podem ser configurados para aceitar outras normas de codificação. Elas são classificadas entre mais críticas e menos críticas, sendo as mais críticas aquelas que violam um *design* de código a ponto de causar um comportamento inesperado, e as menos críticas as que somente dificultam a legibilidade do código.

O projeto do trabalho quando analisado pelo SonarQube, tem uma SQALE Rating A, com Technical Debt Ratio de 0,3%, totalizando o tempo de 30 minutos para pagar a dívida técnica. Existe apenas uma *issue* em aberto de nível médio, que a ferramenta sugere a correção de uma exceção redundante no código, onde é ideal ter uma só exceção passada em um método público. A figura 36 mostra a sugestão da ferramenta.

Figura 36:Sugestão de issue.



Fonte: Figura nossa

* + 1. **Duplicação de Código**

A ferramenta SonarQube faz uma validação do código utilizando algumas normas para identificar código duplicado. Quando analisado o código a ferramenta procura por linhas de códigos repetidas ou muito parecidas, blocos completos com comportamentos próximos ou arquivos que aparentam ser iguais. Figura 37 expõe as informações sobre duplicação.

Figura 37: Duplicidade de código no projeto



Fonte: Figura nossa

No trabalho, a ferramenta não encontrou nenhum código duplicado. O mesmo ocorre por conta do processo de refatoração e também por não ser um projeto grande.

* + 1. **Cobertura**

A análise de cobertura de código é feita através do SonarQube com ajuda de um *plug-in* do Maven, chamado Cobertura esse é um projeto *open source*. O *plug-in* Cobertura fornece um relatório em XML para o SonarQube que expõem os dados de forma sucinta tornando as informações fáceis de serem analisadas.

O quadro de cobertura de código é dividido em duas Partes:

A primeira mede a porcentagem de cobertura de código atingida pelos testes unitários que é a relação Cobertura De Linha com *Condition Coverage* que são as condições totais de cobertura.

Cobertura De Linha são todos os códigos executáveis em métodos do projeto que os testes passaram pelo menos uma vez.

*Condition Coverage* são as condições, *if*, *while*, *for* etc que têm mais de um caminho a ser coberto.

A segunda exibe a porcentagem de sucesso ao executar todos os testes do sistema, a quantidade de teste que falha, que gera erro, e que atendem as condições de sucesso, além do tempo que demora para executar toda a bateria de testes.

A figura 38 mostra o quadro de análise de testes do SonarQube.

Figura 38: Quadro de cobertura de testes



Fonte: Figura nossa.

O projeto do trabalho conta com 49 testes, todos executados com sucesso, e levaram uma média de 111 milissegundos para serem executados. A cobertura que os testes unitários atingiram foi de 99,5% do projeto, e tem o alcance de todo o código fonte (100%), e atende 96,9% das condições possíveis.

A figura 39 mostra a forma gráfica da cobertura de código do projeto. O SonarQube colore em escala de vermelho a amarelo os pacotes ou classes que têm cobertura inferior a 50%, de amarelo a verde-claro os pacotes ou classes com cobertura entre 50 e 80%, e em escala de verde-claro a verde-escuro aqueles que forem superiores a 80%

Figura 39: Gráfico de cobertura.



Fonte: Figura nossa

* + 1. **Complexidade Ciclomática (CC)**

O SonarQube faz uma análise da complexidade total do projeto, e a média por método, por Classe e por arquivo, além de expor um gráfico de colunas com a complexidade ou por método ou por arquivo, se selecionado. A figura 40 mostra a análise de complexidade.

Figura 40: Quadro de complexidade



Fonte: Figura nossa.

O trabalho leva em consideração somente a média de complexidade por método, pois ela se enquadra na classificação de CC. Este projeto tem uma CC média por método de 1,4. O gráfico de colunas mostra que a maior parte das classes tem CC 1 (por método) e o restante em 2 e 4, sendo o último a maior CC, porém com menor incidência.

**CONSIDERAÇÕES FINAIS**

Este trabalho oferece uma visão objetiva de como o TDD pode ser uma técnica alternativa no desenvolvimento de um software, alcançando uma qualidade adequada na implementação de um código funcional.

O conceito técnico do TDD é abordado em um ciclo do processo do trabalho, mostrando como a técnica pode ser aplicada na produção de uma aplicação Web, capaz de transformar um texto em código HTML para a geração de um livro digital ePub.

Com o TDD foi possível alcançar o objetivo esperado, que era garantir a qualidade do código produzido e um *design* de código legível, com baixo acoplamento, alta coesão, testável e passível de mudanças. O processo do desenvolvimento aplicou o TDD tanto na produção do código e quanto no design do modelo de classes.

Foi possível notar o benefício da técnica utilizada, após alguns ciclos com a técnica do TDD, já que a aplicação foi ganhando forma e tornando-se funcional. O objetivo não foi a produção completa da aplicação, mas mostrar que o TDD desacopla a produção do projeto e torna-o menos dependente e mais reutilizável.

Para auxílio da produção do código foi usado o conceito de lista de tarefas, para que o desenvolvedor solo não fique à deriva no início das interações do TDD na produção da aplicação.

Assim que concluído alguns ciclos, foi possível ver o surgimento de um bom *design* de código e a aparição, mesmo que não planejada, de *design patterns* provando que o TDD guia não somente a implementação de código, mais também o *design*, mostrando ser uma técnica completa para o desenvolvimento de um software.

O trabalho principalmente utilizou o *framework* JUnit para facilitar a utilização da técnica de TDD, omitindo a complexidade da utilização da ferramenta de testes, focando o trabalho na técnica do TDD e seus benefícios.

É possível continuar em trabalhos futuros a produção da aplicação de ePub, por meio do TDD, tornando-a uma ferramenta completamente funcional de publicação de livros digitais. Também é possível abordar o *framework* JUnit de forma a explanar melhor as técnicas e alternativas de testes para demonstrar as maneiras que o *framework* de testes auxilia a técnica de TDD. É interessante utilizar a técnica TDD e efetuar a comparação utilizando o SonarQube para avaliar a eficiência da obtenção de qualidade.

**REFERÊNCIAS**

ANICHE, Mauricio. Test Driven Development: Teste e Design no Mundo Real. Internet: Casa do Codigo, 2014. 181 p.

BECK, Kent. **TDD: Desenvolvimento Guiado Por Testes**. Porto Alegre: Bookman, 2010. 240 p.

BECK, Kent et al. **Manifesto para o desenvolvimento ágil de software**. 2001. Disponível em: <http://www.manifestoagil.com.br/>. Acesso em: 01 jan. 2001.

BROOKS JUNIOR, Frederick Phillips. O Mitico Homem-Mes: Ensaios sobre engenharia de software. Rio de Janeiro: Campus, 2009. 300 p.

CAELUM (São Paulo). Caelum Ensino e Inovação. Práticas Ágeis de Desenvolvimento de Software em Java: Curso PM-87. São Paulo, 2014.

CRISPIN, Lisa. **Driving Software Quality**: How Test-Driven Development Impacts Software Quality. In: QUALITY TIME. Quality time. Ieee, 2006. v. 23, p. 70 - 71.

CUNNIGHAM, Ward. Technical Debt. 2014. Disponível em: <http://c2.com/cgi/wiki?TechnicalDebt>. Acesso em: 22 mar. 2015.

DEVMEDIA. Sequência de Fibonacci em Java. Disponível em: <http://www.devmedia.com.br/sequencia-de-fibonacci-em-java/23620>. Acesso em: 22 mar. 2015.

FEATHERS, Michael C.. Trabalho eficaz com codigo legado. Porto Alegre: Bookman, 2013. 406 p.

FOWLER, Martin. TechnicalDebt. 2003. Disponível em: <http://martinfowler.com/bliki/TechnicalDebt.html>. Acesso em: 22 mar. 2015.

FOWLER, Martin. Refactoring. 2013. Disponível em: < http://refactoring.com/>. Acesso em: 30 abr. 2015.

FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code. Mishawaka: Better World Books, 2007. 431 p.

GOMES, André Faria. Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio. Internet: Casa do Codigo, 2014. 176 p.

GONÇALVES, Eduardo Corrêa. Sequência de Fibonacci em Java. 2012. Disponível em: <http://www.devmedia.com.br/sequencia-de-fibonacci-em-java/23620>. Acesso em: 18 mar. 2015.

GUERRA, Eduardo. Design Patterns com Java: Projeto orientado a objetos guiado por padrões. Internet: Casa do Codigo, 2012. 305 p.

KOSCIANSKI, André. Qualidade de Software: Aprenda as metoodologias e técnicas mais modernas para o desenvolvimento de doftware. 2. ed. São Paulo: Novatec, 2007. 395 p.

MARTIN, Robert C. et al. Código Limpo: habilidade prática do Agile Software. Rio de Janeiro: Alta Books, 2011. 456 p.

MASSOL, Vincent; HUSTED, Ted. **Junit em ação**. Rio de Janeiro: Ciência Moderna, 2005. 404 p.

MCCABE, Thomas J.. A Complexity Measure. Ieee: Transactions on software engineeting, Fort George G. Meade, v. 4, n. 2, p.308-319, dez. 1976.

SEVERINO, A. J. **Metodologia do Trabalho Científico**. 23ª Ed. São Paulo: Cortez, 2007. p. 122-123.

SIEGMANN, Paul. Download. Disponível em: <http://www.siegmann.nl/epublib/download>. Acesso em: 22 mar. 2015.

WEIMBERG, Gerald M.. **Software com qualidade**: Pensando e idealizando sistemas. São Paulo: Makron Books, 1993. 387 p.