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



**Vinícius Temóteo Ferrari**

***Test-Driven Development* como alternativa de garantia de qualidade no desenvolvimento de software**

São Caetano do Sul/SP  
2013

**Vinícius Temóteo Ferrari**

*Test-Driven Development* como alternativa de garantia de qualidade no desenvolvimento de software orientado a objeto

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  
2013

**Lista de figuras**

**Lista de abreviaturas e siglas**

TDD - Test-driven development

TF – Test First

UML - Unified Modeling Language

XP – eXtreme Programing

**Sumario**

**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 destas, o reaproveitamento de linhas escritas, transformou a linguagem estruturada, que era difícil de compreender, além de dar suporte a sistemas antigos e sistemas de outros programadores, e ainda trouxe ao mundo real o modelo de objetos.

Os sistemas orientados a objetos deveriam ser mais fáceis de entender, ler, dar manutenção, porém, problemas parecidos surgiram e obrigaram especialistas a criar maneiras de tornar os sistemas padrões mais fáceis de modificar e compreender, criando linguagens de modelagem como a UML – *Unified Modeling Language* – e até ferramentas para melhorar a qualidade no decorrer do desenvolvimento do software. Mesmo assim o tempo, recursos e custos despendidos em testes ainda é grande, e há o risco de no final o projeto já se tornar obsoleto.

Os estudiosos de engenharia de softwares buscam diminuir o risco tanto da obsolescência quanto dos altos custos e tempo. Pensando nisso Kent Beck defende o Desenvolvimento Guiado por Testes (TDD).

Resumidamente, o TDD guia através de pequenos passos para que se alcance o Todo, o objetivo, que é a conclusão do projeto, porém baseando-se em especificações que foram previamente definidas, desta forma caso precise fazer qualquer mudança basta rodas os testes e assim garantir que o Todo continue a funcionar.

Mas, TDD não é somente isso, tem muito mais a agregar e a colaborar com a engenharia de software, entretanto, o TDD é recente e mesmo profissionais com mais de dez anos de experiência ainda não o conhece, além de haver alguma resistência justamente pela falta de conhecimento e de toda sua potencialidade.

Kent Beck, considerado o criador do TDD, lançou a obra TDD Desenvolvimento Guiado por Testes, o qual apresenta um conjunto de técnicas de 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 próximo do real potencial, ou seja, otimizando seus potenciais.

No caso desta obra, o mesmo aplicou prioritariamente na área financeira, e apresenta exemplo com foco neste ramo.

Com base na importância e colaboração que o TDD pode acrescentar e agregar à engenharia de software, através desta pesquisa o pesquisador não busca esgotar o assunto, mas trazer à luz da pesquisa científica através desta pesquisa a exposição de uma aplicabilidade de TDD por meio de um exemplo prático.

Para tanto, o pesquisador adota a metodologia da pesquisa de abordagem qualitativa guiando-se por autores, através de livros, artigos idôneos, e utiliza imagens e vídeos para compor a apresentação teórica e prática de sua pesquisa acadêmica.

**Objetivos**

O projeto tem por objetivo pesquisar sobre test-driven development (TDD) como alternativa de garantia de qualidade no desenvolvimento de software de forma a estudar sobre a qualidade na produção de software; a garantia de legibilidade do código produzido, e também sobre a escalabilidade e a tolerância à mudança de requisitos do sistema.

**Justificativa**

Trata-se de algo complexo e delicado o desenvolvimento de um software, mesmo compondo uma significante parcela do nosso dia-a-dia. Profissionais de todos os ramos munem-se de softwares para facilitar tarefas difíceis, ou repetitivas atividades profissionais. À medida que a complexidade do problema aumenta a qualidade dele tende a diminuir, isso porque é difícil documentar todas as necessidades, com isso houve uma revolução nos métodos de desenvolvimento surgindo o manifesto ágil, que aconteceu em 2001. Foram definidos vários padrões tanto de modelagem quanto de desenvolvimento ou até métodos de melhores práticas para tentar solucionar os principais problemas de qualidade de software e entre produtos de softwares que satisfazem os *stakeholders*. O trecho do artigo abaixo mostra que durante um processo de desenvolvimento de software a qualidade e a satisfação dos *stakeholders*, foram garantidas ao utilizar o desenvolvimento ágil somado a 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, *grifo do autor*)

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 mesmo prazer, que nós entregamos exatamente o que eles pediram para cada iteração. Nós implementamos características que as empresas concorrentes pensavam 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,* eles tentam estabelecer parâmetros de qualidade e permitir que o desenvolvimento seja versátil e fácil de ser modificado, e que são passiveis a 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.

Escrever código que possa ser compreendido por outro desenvolvedor ou por uma equipe de desenvolvimento é um desafio para engenharia de software e o TDD tenta solucionar o problema obrigando cada unidade de código, cada método ou função, seja testada antes de ser alterada, impedindo falhas que podem ser ocasionadas durante a 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, por quanto os passos de implementação sejam 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, *grifo do autor*)

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 produzido código que passou entre 18 e 50 por cento de casos de teste mais externos do que código produzido por grupos de controle não usando TDD. Além disso, os programadores TDD gastou 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 garantir que caso alguma alteração quebre as associações entre os métodos ou classes, os testes, informarão rapidamente o desenvolvedor ou o time que esta 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, mantendo a concentração na necessidade real daquele problema, em um software, ajuda o programador a escrever só o necessário para prosseguir para próxima necessidade 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 seja do produto final, módulo ou de um protótipo do sistema.

**Materiais e procedimentos**

Para a pesquisa serão estudadas algumas literaturas sobre o assunto, serão utilizados recursos, tanto impressos (livros), como digitais (e-livros) encontrados em bibliotecas ou na internet. Imagens e vídeos também serão analisados para compor o estudo.

O projeto é o desenvolvimento de um ou vários códigos-fonte, módulos ou protótipos de sistemas de software, utilizando tal técnica de programação. O software utilizará ferramentas livres e/ou de código aberto, será utilizado um framework para testes (JUnit), próprio para linguagem Java. O sistema será desenvolvido na mesma linguagem de programação e terá exemplos com o código escrito, guiado pelo TDD.

**Resultados esperados**

É esperado que o referido exemplo exposto nesta pesquisa apresente um desempenho superior por conta da aplicabilidade do TDD em comparação às técnicas comuns, ou seja, antecessores ao TDD. O estudo também espera analisar a forma de construção de código de um software, e observar as classes e métodos criados e o design legível superior ou equivalente a técnica padrão de programação.

A facilidade de implantação da técnica em um produto de software iniciado é provavelmente mais dificultosa que no modelo padrão de desenvolvimento e a qualidade do produto de software é superior ou igual à de um programa desenvolvido por modelos.

**1 FUNDAMENTAÇÃO TEÓRICA**

1.1 Qualidade

Weinberg (1993) aponta a questão da relatividade quanto à qualidade, 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, o autor 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.

Porque para 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 isto se soma a questão “valor” o quanto alguém esta 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 o autor 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”.

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

Existe atualmente uma área totalmente voltada para garantir qualidade em produtos de software, órgãos como ISO, ABNT, entre outros, criam norma 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, porem quando pensamos exatamente o que é necessário e o que é preciso, conseguimos estabelecer um parâmetro e então avaliar em qual nível o produto esta.

Quando Weinberg (1993, p.10) trata a definição de qualidade, ele exemplifica com um jogo de *Cribbage*, que mesmo com defeitos, da contagem do placar, mostra-se satisfeito com a qualidade do produto, já que adquiriu gratuitamente.

Desenvolver um teste antes de programar pode ser uma coisa difícil de se pensar para programadores que não estão acostumados com a prática, porém fazemos praticamente isso sempre que programamos, quando é criada uma classe que deve retornar um cálculo difícil e criamos um formulário rápido para preencher os parâmetros necessários do método, roda-se o programa, passam os valores dos parâmetros e é retornado um resultado que já era esperado, entretanto o método devolveu um valor diferente do esperado, é uma situação que a técnica pretende te levar, porém antes mesmo de ter a assinatura da função antes de criar a primeira classe, já existe um teste especifico para esse problema, e o que deve ser feito é, o teste passar da maneira mais simples possível, nem que isso implique em um método retornando o valor esperado. Pensar em passos pequenos é um dos primeiros passos que é preciso ter em mente, e a premissa básica é: “*Escrever código novo apenas se um teste automatizado falhou*” (Beck, 2010), ou seja, antes de qualquer coisa é necessário ter um teste falhando antes de modificar alguma coisa, em um programa recém iniciado, não existe nada além de um único teste que faça o programa falhar, esse teste de início não precisa nem compilar ainda. Como mostra a figura 7;

Figura 7: Primeiro teste

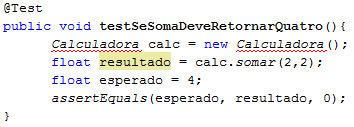


Figura nossa

O Teste não compila, pois não existe classe calculadora e nem método somar.

Depois de ter o primeiro teste falhando, deve-se criar o mínimo necessário para que o teste passe a funcionar e retornar o valor esperado (programar o mínimo necessário é conhecido como *baby-steps* ou passos de bebe entre os programadores em TDD). Para que o teste passe, será feito um *stub* – um *stub* ou esboço, é um método ou trecho de código que simula uma operação de computação, são comumente adotados para testar código em desenvolvimento. A figura 8 mostra um exemplo de *stub*.

Figura 8: *Stub* de classe Calculadora.

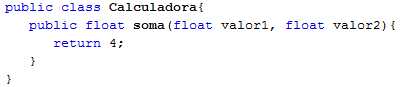


Figura nossa

A partir deste momento o programador pode agora ou ficar satisfeitos com o programa ou criar mais um novo teste para melhora-lo e podemos criar um novo teste esperando um outro valor de retorno, tornar o programa funcional, aplicando uma técnica praticamente obrigatória no TDD a refatoração. Antes de refatorar, é executado o teste verificamos todos que falharam e depois refatorar e executar novamente os testes.

Outra regra que Beck (2010) colocou no desenvolvimento da técnica, foi “*Eliminamos Duplicação*” para garantir que a complexidade diminua sempre que refatorar, é preciso colocar a repetições e métodos separados, assim reusamos sempre que necessário.

O ciclo de vida do TDD fica restrito a três passos simples, explicado no capítulo anterior.

As etapas citadas supõem garantir uma densidade reduzida, aumentando a qualidade e tornando a qualidade relativa para proativa, diminui as surpresas que o desenvolvimento de software, tornando a causa do problema mais fácil de ser encontrada, o retorno das informações tornam-se mais rápidas o que torna também o desenvolvimento mais rápido al decorrer do processo.

O TDD como técnica, visa o *feedbeck* do teste como alternativa do desenvolvimento do software, tornando a migração de um método menos crítica para um sistema, a colaboração fica mais fácil pois quando um membro da equipe quebra um método todos os programadores podem imediatamente solucionar o problema, a evolução tende a ser acelerada, e o suporte para testadores já estará coberto.

O que é qualidade

Porrque é importante

Efeito do feedback .. vantagem do retorno rápido do tdd

Quando usar tdd

Quando descartar código ..refatoraçao

Não existe bala de prata

**1.3 Test-Driven Development - TDD**

Framework, segundo a definição de alguns autores, refere-se a um conjunto de classes que constitui um projeto abstrato voltado à solução de uma família de problemas; com uma arquitetura definida e desenvolvida para uma família de subsistemas, o qual oferta os construtores básicos para criá-los, e para atingir a máxima reutilização com grande potencial de especialização; e exposto *hot-spots* (pontos de extensão) onde devem ser feitas as adaptações do código para um funcionamento específico de certos módulos (MATTSSON, 1996; MATTSSON, 2000; JOHNSON, 1991; GAMMA et al, 1995; BUSCHMANN et al, 1996; PREE, 1995; PINTO, 2000).

Software segundo a Lei 9609/98 (Lei do Software) em seu artigo 1º define ‘programa de computador’, software é

“[...] a expressão de um conjunto organizado de instruções em linguagem natural ou codificada, contida em suporte físico de qualquer natureza, de emprego necessário em máquinas automáticas de tratamento da informação, dispositivos, instrumentos ou equipamentos periféricos, baseados em técnica digital ou análoga, para fazê-los funcionar de modo e para fins determinados”

Segundo Cabral (2007)[[1]](#footnote-1) a lei prevê proteção aos programas de computador similar as obras literárias, artísticas e científicas, apesar do sistema legal brasileiro vedar a proteção de software através de concessão de patente mas, o permite como bens protegidos pelo direito de autor. Por causa do ‘código-fonte’.

O código-fonte’ refere-se a linguagem

que permite a um determinado programador desenhar instruções lógicas para um computador sobre aquilo que ele deverá executar. O computador opera, entretanto, com o que se chama “código objeto”, conjunto de 0 (zeros) e 1 (uns) na maioria das vezes impenetrável para o entendimento humano ordinariamente. Dessa forma, as instruções dadas pelo programador através do “código fonte” são posteriormente “compiladas” pelo computador, isto é, traduzidas da linguagem intermediária do código fonte para a linguagem de máquina, composta de 0 (zeros) e 1 (uns) (CABRAL, 2007).

O termo copyleft, segundo Cabral xxx, traz à tona peculiaridades quanto as questões envolvendo o software, no caso, os livre, que se centram muito mais em peculiaridades jurídicas do que em técnicas, assim ele se distingue dos demais em virtude de mecanismos técnicos, e jamais deve-se confundir software livre do gratuito, são questões totalmente diferentes.

Retomando as questões técnicas, há 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.

Mas 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 softwares 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) sobre qualidade, “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.

E como o mesmo aponta, essa arte esta coberta de alegrias, como a satisfação de construir algo, a de construir algo útil para as pessoas, o fascínio de desenvolver algo complexo, a aprendizagem que esse desenvolvimento oferece e sem repetições, e a maleabilidade de trabalhar com pensamento puro (2009, p. 7).

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 (BROOK JR, 2009, p. 8).

Pensando muito mais nos contra que nos prós, muita pesquisa e estudos tem 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.

“*Clean Code* de Roberto C. Martin e 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 (BOOCH)

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 (THOMAS)

Por que um código limpo?

Segundo Beck (2010, 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; melhora-o 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 xxx, p. prefácio alerta que através desse o TDD, escreve-se o código novo apenas se um teste automatizado falhar; e eliminando duplicação.

Seguindo essas duas regras, Beck diz que isso ira 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/refator’ 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 duplicadas 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.

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.

O autor explica que através do TDD quando um teste esta em funcionamento, ele esta 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”.

**1.3.1 TDD – Alternativa de garantia de qualidade no desenvolvimento de software**

TDD ou Teste First (TF) 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 1 mostra a relação das palavras citadas por Beck com o objetivo de cada passo.

Figura 1: Mantra do TDD

****

Fonte Diogoosorio, 2012

1.3.2 *JMock*

O *JMock* é uma biblioteca que auxilia o *Test Driven Development* através dos *mock objects*. É uma biblioteca que vai criar implementações de mentira específicas para o seu teste, de uma maneira rápida e simples, sem ter que se preocupar com os métodos que não vamos usar no teste, sem ao menos ficar criando classes “a toa”. Com o *JMock* podemos definir o comportamento necessário do objeto de mentira, para criarmos a situação pedida pelo teste (CAVALCANTI, 2008).

**1.3.3 JUnit: Framework para testes automáticos**

A principal ferramenta para se desenvolver em TDD é o JUnit uma ferramenta para automatizar testes de unidade em softwares desenvolvidos em Java, esta ferramenta permite a execução de grandes quantidades de testes através de modo textual ou modo gráfico.

Massol (2005) tenta 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.

Produzir enquanto roda uma bateria de teste é mais uma facilidade que o framework 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.

**1.3.4 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.

As figuras 2 mostra o trecho de código extraído do livro 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 2: Teste não compila



Fonte: Adaptado de Beck, 2010.

A Figura 3 mostra o framework JUnit exibindo os resultados do teste da Figura 1 de forma gráfica, mais a frente será abordado a ferramenta de teste JUnit.

Figura 3: 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 esta 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*.

Neste 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 4 mostra as linhas adicionadas e o resultado do teste depois da execução.

Figura 4: 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 5.

Figura 5: 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 Martin Fowler, com ajuda de Kent Beck (2004), descrevem as melhores práticas de refatoração, entretanto logo no prefacio Fowler 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 modulo 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, Martin Fowler 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.”.

Esta 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 6: 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 para ou novas funcionalidades ao sistema.

Mais adiante a pesquisa irá abordar técnicas de refatoração, pois é um dos principais itens que garante a qualidade do software e permite a legibilidade e a compreensão para equipes, além de influenciar na qualidade final do software.

* + 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. **Teste caixa-preta ou teste Funcional**

Os testes de caixa-preta são testes funcionais e tenta enquadrar todas as possibilidades de teste possíveis, não se preocupam com a estrutura do programa em si e sim na funcionalidade do método. Os métodos são testados sem saber como foram implementados e esses teste tentam colocar a regra do negocio no teste importando-se apena com o resultado final.

Esta categoria de teste procura testar o comportamento de cada pedaço do sistema, tendo como base os requisitos e as especificações do produto, pretendendo validar a ação que cada parte do sistema irá operar. Quando é feito este tipo de teste é esperado verificar todas as possibilidades de entrada e comparar suas saídas, entretanto a pratica é inviável na maioria das vezes, já que os testes ocupam tempo e recursos computacionais elevados.

O sistema é tratado como uma caixa-preta, cujo comportamento pode ser somente determinado por meio do estudo de suas entradas e as saídas relacionadas. Outro nome para isso é teste funcional, pois o testados concentra-se na funcionalidade, e não na implementação do software. (SOMERVILLE, 2007, p. 359).

Existe uma extensa gama de testes funcionais que tem por objetivo garantir a qualidade em todos os componentes e módulos do sistema.

* + 1. **Teste caixa-branca ou teste Estrutural**

Testes de caixa-branca, são testes estruturais que em nível de código, testa unidades do sistema, são criados para complementar o código produzido. É comumente criado conforme é codificado o componente, o programador constrói o componente, classe ou funcionalidade e cria testes para garantir que o mesmo retorne os valores esperados, nesta etapa o programador que define os casos de teste. Outra abordagem que é comum em XP é aplicar TDD para construção de código.

Sommerville (2007, p. 367) especifica esse tipo de teste dizendo, “O teste estrutural é uma abordagem para projetar casos de teste na qual o teste são derivados do conhecimento da estrutura e da implementação do software” e completa observando que também é conhecido como teste caixa-branca.

* 1. Diagramas de Influência

Segundo Weinberg (1993) é muito importante observar como os ‘elementos de um sistema afetam um ao outro’, e para ele há três elementos:

Atividades, anotadas como uma palavra ou frase curta

Conexões positivas,anotadas como uma flecha direcionada entre duas atividades, significando que mais da atividade origem tende a criar mais da atividade destino, ou menos da atividade de origem tende a criar menos da atividade destino.

Conexões negativas, anotadas com flechas direcionadas entre duas atividades com um círculo sobre ela, significando que mais da atividade de origem tende a criar menos da atividade destino, ou menos da atividade de origem tende a criar mais da atividade destino. (WEINBERG, 1993, p. 227).

Daí o Feedback, pois uma “[...] influência não funciona em apenas uma direção. Frequentemente o efeito de uma atividade volta para mudar a própria atividade, positivamente ou negativamente [...]”, Para Weinberg (1993, p. 228) “Sempre que você tem um ciclo em um diagrama de influência, você tem *feedback*”.

E há dois tipos de feedback: positivo ou negativo.

O positivo causa sistemas que encorajam mais e mais de uma atividade. Já o negativo desencoraja ou reduz uma atividade (WEINBERG, 1993, p. 229).



Fonte Test Driven Development: Teste e design no mundo real com Java

Mauricio Aniche

Figura 7: Efeito *feedback*

FALTA:

* curva de detecção de falhas (E gráfico)
* quando usar
* descartar codigo
* não existe bala de prata(artigo sobre qualidade)

**Cronograma do projeto**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Maio | Junho | Julho | Agosto | Setembro | Outubro | Novembro |
| **Elaboração do projeto** | X | X |  |  |  |  |  |
| **Entrega do projeto** | X |  |  |  |  |  |  |
| **Pesquisa bibliográfica** | X | X | X | X |  |  |  |
| **Coleta de dados** |  |  | X | X |  |  |  |
| **Apresentação de discussão dos dados** |  |  |  | X | X |  |  |
| **Conclusão** |  |  |  |  |  | X | X |
| **Entrega do TCC** |  |  |  |  |  |  | X |
| **Defesa da banca** |  |  |  |  |  |  | X |

**REFERÊNCIAS**

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

CABRAL, Antonio. Software e Software Livre. Disponível em:< http://academico.direito-rio.fgv.br/ccmw/index.php/Software\_e\_Software\_Livre#Software:\_Defini.C3.A7.C3.A3o> acesso em set 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.

DIOSORIO. **Test driven development tdd using phpunit**. Disponivel em http://diogoosorio.com/blog/entry/test-driven-development-tdd-using-phpunit acesso em 12/05/2013 .

FOWLER, Martin et al. **Refatoração**: aperfeiçoando o projeto de código existente. Porto Alegre: Bookman, 2004. Disponível em: <http://books.google.com.br/books?id=zPdb4QJKBtkC&pg=PA19&hl=pt-BR&source=gbs\_toc\_r&cad=4#v=onepage&q&f=false>. Acesso em: 13 maio 2013.

FOWLER, Martin. **Refactoring Home Page**. Disponível em: <http://www.refactoring.com/>. Acesso em: 14 maio 2013.

HAUGSET, Børge; STÅLHANE, Tor. Automated Acceptance Testing as an Agile Requirements Engineering Practice. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 45, 2012, Hawaii. Computer society. Hawaii: Ieee, 2012. p. 5289 - 5298.

HERNANDEZ, Alexander A et al. JLearn: An Instructional Environment for Java Program Composition integrating Test-Driven Development and Life-Cycle Management for Software Quality Assurance. INTERNATIONAL CONFERENCE ON NETWORKING AND INFORMATION TECHNOLOGY. Manila, p.166-170.2010.

JANZEN, David S.; SAIEDIAN, Hossein. Does test-driven development really improve software design quality?. **Feature: Software Metrics**. California, p.77-84. abr. 2008.

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

MÜLLER, Matthias M.; HÖFER, Andreas. The effect of experience on the test-driven development process. In: Empirical Software Engineering, 12, 2007. Ieee, 2006. v. 12, p. 593 - 615.

PANCUR, Matjaz; CIGLARIC, Mojca. Impact of test-driven development on productivity, code and tests: A controlled experiment. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 45., 2012, Ljubljana. Information and Software Technology. Ljubljana: Ieee, 2011. v. 56, p. 557 - 573.

SOMMERVILLE, Ian. **Engenharia de Software**. 8. ed. São Paulo: Pearson, 2007. 552 p.

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

WIKIPEDIA. **Refatoração**. Disponível em: <http://pt.wikipedia.org/wiki/Refatoração>. Acesso em: 14 maio 2013.

FALTA INCLUIR AS NOVAS BIBLIOGRAFIAS!!

1. http://academico.direito-rio.fgv.br/ccmw/index.php/Software\_e\_Software\_Livre#Software:\_Defini.C3.A7.C3.A3o [↑](#footnote-ref-1)