UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

MARCOS VINÍCIUS RODRIGUES DE SOUZA

ABORDAGENS DE TESTES: UMA COMPARAÇÃO ENTRE AS METODOLOGIAS TRADICIONAL E ÁGIL

TRABALHO DE CONCLUSÃO DE CURSO

Recife

16 de Dezembro de 2016

MARCOS VINÍCIUS RODRIGUES DE SOUZA

ABORDAGENS DE TESTES: UMA COMPARAÇÃO ENTRE AS METODOLOGIAS TRADICIONAL E ÁGIL

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação, como parte dos requisitos necessários à obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Célio Andrade de Santana Júnior

Recife
16 de Dezembro de 2016

Marcos Vinícius Rodrigues de Souza
ABORDAGENS DE TESTES: UMA COMPARAÇÃO ENTRE AS METODOLOGIAS TRADICIONAL E ÁGIL/ Marcos Vinícius Rodrigues de Souza. – Recife, 16 de Dezembro de 2016-
58 p. : il. (algumas color.) ; 30 cm.

Orientador: Célio Andrade de Santana Júnior

Trabalho de Conclusão de Curso – UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
, 16 de Dezembro de 2016.

IMPORTANTE: ESSE É APENAS UM TEXTO DE EXEMPLO DE FICHA CATALOGRÁFICA. VOCÊ DEVERÁ SOLICITAR UMA FICHA CATALOGRÁFICA PARA SEU TRABALHO NA BILBIOTECA DA SUA INSTITUIÇÃO (OU DEPARTAMENTO).

Marcos Vinícius Rodrigues de Souza

ABORDAGENS DE TESTES: UMA COMPARA ÇÃO ENTRE AS METODOLOGIAS TRADICIONAIS E ÁGEIS

Trabalho entregue. Recife 13 de dezembro de 2016.

Trabalho aprovado. Recife, _____ de_____ de 2016.

_____

Célio Andrade de Santana Júnior
Orientador

_____

Sérgio Castelo Branco Soares
Avaliador

Dedico este trabalho e as vitórias que virão em sua consequencia aos esforços dos meus pais Ana Lúcia e Marcos Souza que sempre buscaram me garantir tranquilidade, apoio e uma educação de qualidade para seguir minhas escolhas.

Agradecimentos

Agradeço a Deus pela oportunidade de estar no convívio da minha família e amigos e de ter a oportunidade de aprender com eles.

Agradeço aos meus pais Ana Lúcia e Marcos Souza por não se furtarem do trabalho em educar seus filhos da melhor maneira possível, por sempre acreditarem e investirem no meu potencial e por serem minha base e meu exemplo de como seguir uma vida honrada e feliz. Agradeço às minhas irmãs Syntia Regina e Paloma Rodrigues por partilharem tantos momentos comigo, pelo incentivo a seguir sem pensar em abandonar o barco e pelo meu sobrinho Lucas que hoje é a alegria da casa. Também agradeço a Emanuelle Emy, mais que amiga e namorada, é a pessoa que me apoia sempre e me traz à realidade quando ando afastado dela e me mostra todos os dias que sempre há motivos para melhorar. E a Thiago Rodrigues, primo e irmão que a seu modo, sempre demonstrou que há recompensas no fim do longo caminho.

Não poderia deixar de agradecer a meus amigos Alysson Santos, Pedro Sousa e Rafael Brito, cumplices de aulas, provas, projetos e risadas ao longo do curso e amigos que certamente levarei para vida. Breno Augusto, Carlos Castor e Carlos Eduardo Brandão amigos da vida que sempre me deram força a persistir sem deixar me abater pelos percalços da caminhada.

Finalmente, gostaria de expressar minha eterna gratidão a todos os que fazem o Centro de Informática e a Universidade Federal de Pernambuco, mais do que ensinamentos técnicos, apresentaram a realidade do país e como a educação de qualidade pode modifica-la.

Muito Obrigado!

A primeira regra de qualquer tecnologia utilizada nos negócios é que a automação aplicada a uma operação eficiente aumentará a eficiência. A segunda é que a automação aplicada a uma operação ineficiente aumentará a ineficiência.
(Bill Gates)

Resumo

As mais variadas propostas para desenvolvimento de software tem etapas em comum, são elas: Especificação de Requisitos, Codificação e Testes. Assegurar a qualidade do que for produzido em cada uma destas etapas configura ao produto de software melhor índice de confiabilidade, eficácia e eficiência. A etapa de testes cuida da validação da eficácia do time, garantindo que o tudo que foi especificado, foi devidamente implementado e da eficiência do produto, comprovando que ele faz corretamente o que é esperado. Ao longo do tempo a academia e a indústria tem realizado esforços no sentido de propor novas metodologias de testes que melhorem a produtividade do time e a qualidade do produtos de *software*. Desta forma, várias metodologias e técnicas tomaram lugar como referência de processo de qualidade, testes tradicionais com a disciplina e rigidez que permitiam mensurar precisamente o esforço investido por cada colaborador e adapta-los ao cronograma de trabalho do projeto e as metodologias ágeis que se adequam facilmente às mudanças de requisitos propostos pelo cliente e/ou pelo mercado e procuram trazer o cliente para participar da construção do seu projeto. Este trabalho tem como objetivo apresentar as técnicas mais populares dessas metodologias e realizar um comparativo, entre as práticas adotadas, ciclos de vida e resposta do mercado. Assim, neste trabalho pudemos concluir que apesar de os testes na metodologia ágil terem mais benefícios do que críticas, ela por si só, não tem o poder de avalizar que o sistema em questão está livre de falhas, bem como os testes da metodologia tradicional também não os tem. É possível que um projeto tenha atividades de testes das duas metodologias sendo executados, cada um em seu escopo, e assim, tomar proveito das qualidades das duas metodologias e mitigando os pontos de melhoria que cada uma apresenta.

Palavras-chave: Testes de *Software*, metodologias ágeis, Modelo V, *Test Driven Development,* TDD*, Behaviour Driven Development*, BDD.

Abstract

The most varied proposals for software development have common steps, they are: Requirements Specification, Coding and Testing. Ensuring the quality of what is produced in each of these stages configures the best software program for reliability, effectiveness and efficiency. The testing stage takes care of the validation of the team's effectiveness, ensuring that everything that was specified was properly implemented and product efficiency, proving that it does what is expected to be done correctly. Over time the academia and industry has made efforts to propose new testing methodologies that improve the productivity of the team and the quality of software products. In this way, several methodologies and techniques have taken place as reference of quality process. Traditional tests with the discipline and rigidity that allowed to measure precisely the effort invested by each employee and adapt them to the work schedule of the project and the agile methodologies that fit easily to the changes of requirements proposed by the client and / or the market and seek to bring the customer to participate in the construction of their project. This work aims to present the most popular techniques of these methodologies and to carry out a comparison between the practices adopted, life cycles and market response to their time. In this way, it can be concluded that although the tests in the agile methodology have more benefits over critics, it alone does not have the power to guarantee that the system in question is fault-free, as well as the tests of the traditional methodology neither have them. It is possible that a project has test activities of the two methodologies being executed, each in its scope, and thus, take advantage of the qualities of the two methodologies and mitigating the improvement points that each presents.

Keywords: Software Testing, Agile Methodologies, Model V, Test Driven Development, TDD, Behavior Driven Development, BDD

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ATDD	Acceptance Test Driven Development
BDD	Behaviour Driven Development
DDD	Design Driven Design
DMP	Domain Model Pattern
DoD	Definition of Done
DSMP	Dynamic Systems Development Method
FDD	Feature Driven Development
ISTQB	International Software Testing Qualifications Board
TDD	Test Driven Development
UAT	User Acceptance Testing
XP	eXtreme Programming

Sumário

1 Introdução

1.1 Contextualização

A indústria do desenvolvimento de software está, desde sua concepção, em permanente estado de mutação e como consequência, diversas metodologias e técnicas de desenvolvimento de software surgiram com o objetivo comum de trazer eficácia e eficiência ao ritmo de produtividade esperado a seu tempo. Desta forma, surgiram diversos ciclos de vida de processos de software, tais como o modelo interativo e incremental, largamente adotado pelas metodologias ágeis como uma das propostas para constantemente entregar valor ao cliente e o introduzir a sua participação no processo, o modelo em espiral proposto em *"A Spiral Model of Software Development and Enhancement"* (BOEHM, 1988)"*Managing the Development of Large Software Systems"* (ROYCE, 1970).

Diversos questionamentos a respeito da efetividade "metodologias tradicionais" (cascata e espiral) foram lançados, devido aos frequentes estouros de cronograma e/ou orçamento e pela real dificuldade observada no gerenciamento de projetos de software que utilizam estas metodologias. Este fenômeno pode ser observado nas pesquisas primeiramente intituladas *Chaos Report*, que raz o resultado acumulado entre as taxas de sucesso, desafios e falhas de projetos em comparação às metodologias aplicadas como pode ser visto na Figura 1 a seguir. Recentemente foi lançada com o nome de *Chaos Manifesto* (The Standish Group International, 2015)

Figura 1 – Relação sucesso/falha nos projetos ágeis/ cascata

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

Podemos ver que cerca de 20% dos projetos realizados com o ciclo de vida cascata falham em comparação com os 9% dos projetos utilizando abordagens ágeis. A partir de números como estes é que as metodologias ágeis passaram a ter grande aceitação no mercado ao apresentar soluções para os problemas comuns ao processo de desenvolvimento de software, isso porque os valores adotados nas metodologias ágeis as caracterizam como metodologias adaptativas ao invés de preditivas e são orientadas às pessoas e suas interações e não ao processo.

Estes valores, assim como outros de igual importância, estão descritos no manifesto ágil (BECK et al., 2001) que é o documento que unifica as características comuns a todas as metodologias de desenvolvimento que são adeptas destes pensamentos. Aqui se destacam os princípios ágeis "*Software funcional mais que documentação abrangente*" e "*Colaboração do cliente acima das negociações contratuais*" que vislumbram técnicas de desenvolvimento onde o objetivo final é escrever um código enxuto, funcional e com qualidade. Nesse sentido, Kent Beck estendeu esse conceito de agilidade as práticas de teste ao propor em 2001 o *Test Driven Development (TDD)* (BECK, 2002) que, segundo o autor, é uma forma de guiar o desenvolvimento com testes automatizados. Com esse novo estilo de desenvolvimento, o TDD tornou-se rapidamente popular na academia e indústria.

Ao sugerir a prática dos testes como atividades iniciais e finais do processo de desenvolvimento, Beck tinha como objetivo mitigar alguns dos riscos de possíveis falhas de projeto atacando um dos ponto em que as organizações mais apresentavam novas alternativas e soluções para garantir um produto de qualidade. Essas alternativas contemplam novas ferramentas, novos processos de testes e novas técnicas associadas à fase de validação de software.

O conceito dos testes utilizado no TDD se diferencia em relação às metodologias tradicionais de teste ao colocar o teste como atividade núcleo do desenvolvimento mesmo quando se utiliza conceitos adotados na práticas tradicionais. Criam-se testes unitários relacionados à funcionalidade desejada que inicialmente deve falhar no primeiro momento por não ter a implementação do código da funcionalidade, e após essa primeira implementação, chamada de *mock*, implementa-se um código simples para fazer com que o teste passe e a partir daí refatora-se o código aplicando os padrões desejados de modo a deixá-lo limpo e funcional, sem "quebrar" os testes e funcionalidades já existentes. Assim, o TDD usa conceitos de teste unitário, de integração e regressão a cada etapa da implementação do sistema. Objetivando desta forma, tornar menos burocrático por manter o desenvolvedor encarregado de criar, passar e melhorar os teste.

Utilizar TDD em projetos reais, disseminar suas práticas torná-la mais acessível para programadores iniciantes fez com que Dan North publicasse o artigo "*Behavior Modification: The Evolution of Behavior-Driven Development*" mostrando inquietação com as dificuldades no uso do TDD e que por isso todos os projetos deveriam se iniciar a partir das seguintes questões:

- Por onde começar? O que priorizar?
- O que testar?
- Como definir os testes?
- Como compreender melhor a causa da falha de um teste?

Tendo como ponto de vista inicial que quem deveria escrever os testes são os desenvolvedores, mas que as auditorias devem ser realizadas pela equipe de qualidade com o foco não no código, ou implementação, e sim no comportamento esperado do sistema não vislumbrando apenas atividades ligadas aos testes unitários ou caixa branca. A busca por soluções destes problemas o levou a propor o *Behaviour Driven Development* (BDD) como uma resposta às limitações encontradas no TDD e aos métodos tradicionais de desenvolvimento de *software*.

1.2 Motivação

Este cenário de rápida evolução onde novas práticas de desenvolvimento eram propostas também se debruçou no próprio TDD uma vez que havia a preocupação cada vez maior com a parte relativas a negócios onde havia uma forte orientação ao domínio do escopo do negócio somada a um desenvolvimento visando testes (não apenas unitários, mas também de aceitação, e regressão) em que boas práticas do TDD são necessárias. Assim se fazia necessário ampliar os conhecimentos sobre as técnicas de desenvolvimento, em específico o BDD, que não se limita a uma nova interpretação do TDD e sim, se propor um modo novo de construir *software*, incorporando as práticas do TDD com conceitos trazidos pelo *Design Driven Design (*DDD*)* (EVANS, 2004)

BDD se propõe a mudar a forma como se aborda o desenvolvimento de *software*, dando mais ênfase aos comportamentos especificados, passando estes a guiar o desenvolvimento da aplicação e não puramente os testes, contudo, os testes continuam a ser importantes assegurando que as novas funcionalidades estão bem implementadas, que objetivem a solução para o comportamento esperado e garantindo que o código existente não foi “estragado”.

Assim, as práticas de BDD permitem/demandam que os clientes sejam protagonistas em todas as etapas do processo, sendo necessária a utilização de uma linguagem simplificada para especificação de requisitos, que facilite o entendimento de todos os participantes e permita concentrar nas razões pelas quais o código deve ser criado, e não se concentrar nos detalhes técnicos, além de minimizar traduções entre a linguagem técnica na qual o código é escrito e outras linguagens de domínio.

Desta forma, se torna interessante investigar essa nova forma de desenvolver *software* onde o papel central é deslocado para o cliente e como os anseios do cliente são garantidos. Ao mesmo tempo que esse formato diminui as barreiras entre cliente e equipe de desenvolvimento, apresenta novos desafios inerentes à Engenharia de Software, e ao desenvolvimento ágil em si, que precisam ser investigados.

1.3 Objetivos

Por se tratar de assunto relativamente novo e de certo interesse na comunidade ágil, este trabalho se propõe a investigar, a partir de uma pesquisa bibliográfica e documental, os principais conceitos do relacionamento entre as abordagens de testes tradicionais e as abordagens de testes ágil, e como o mesmo vem sendo retratado na bibliografia. Desta forma podemos inferir os seguintes objetivos específicos:

- Identificar na literatura acadêmica e técnica material referente a Testes nas metodologias Tradicionais e Ágeis;

- Iniciar uma coleta de dados baseado na técnica fichamento dos principais assuntos relativos a Abordagens de Testes;
- Fazer uma análise temática dos principais pontos relativos a Testes encontrados na literatura;
- Explanar as principais características dos testes tradicionais e testes ágeis;
- Realizar um breve comparativo entre as duas abordagens.

1.4 Estrutura do trabalho

O presente trabalho está estruturado em capítulos e, além desta introdução, será desenvolvido da seguinte forma:

- Capítulo II: Este capítulo contém a definição de Testes de Software, trazendo o contexto de sua aplicação, níveis, tipos e abordagens de teste que asseguram a validade dos requisitos.
- Capítulo III: Este capítulo detalha os procedimentos de Testes dentro das Metodologias Ágeis, citando os processos, técnicas, pontos positivos e negativos
- Capítulo IV: Este capítulo busca realizar uma comparação entre as abordagens de testes tradicionais e ágeis preparando para as considerações finais do trabalho
- Capítulo V: Conclui o trabalho apresentando um resumo do que foi aprendido durante a construção do mesmo e uma proposta de trabalhos futuros.

2 Testes de Software

Teste de *software* é o processo de execução de um produto para determinar se ele atingiu suas especificações e se funcionou corretamente no ambiente para o qual foi projetado. O seu objetivo é revelar falhas em um produto, para que as causas dessas falhas sejam identificadas e possam ser corrigidas pela equipe de desenvolvimento antes da entrega final (CLAUDIO, 2015).

Segundo (MYERS, 2004) em seu livro "*The Art of Software Testing*", diz que a tarefa de testar um *software* é extremamente criativa e intelectualmente desafiadora, portanto sugeriu princípios para que o teste seja considerado confiável, sustentável e agregue valor ao projeto de *software* e a execução destes, resumidamente, são eles:

- Um programador não deve testar seu próprio código, bem como uma equipe de desenvolvimento não deve testar seu próprio projeto. Testes de *software* são, por definição, uma atividade "destrutiva", ou seja, procura por falhas em um trecho do programa para que estes sejam corrigidos antes de o projeto chegue ao cliente. Por ser uma atividade com fim de criticar o trabalho, não deve ser feita por alguém que tenha envolvimento no desenvolvimento do mesmo, visto que este poderá viciar os testes, validando algo que não tem "valor de negócio" ou que não esforce o suficiente o teste para não gerar "provas contra si mesmo" diante os outros membros do time. Estendendo a mesma lógica, na visão tradicional de testes de *software*, projetos devem ser testados por equipes independentes para que haja isonomia e independência nos processos e resultados dos testes.
- Os casos de teste devem ser escritos tanto para condições de entrada inválidas e inesperadas, quanto para as válidas e esperadas sendo de vital necessidade deixar claro os resultados esperados de cada teste. Existe uma tendencia natural em profissionais em concentrar seu esforço para validar os casos esperados com entradas válidas, dessa fora, assegura-se que o programa é eficiente, fazendo aquilo que deve ser feito. Esta regra, busca valorizar os "testes negativos" em que se exercite tanto o que o programa não deve fazer mediante uma entrada válida, quanto como ele deve se comportar mediante uma entrada inválida, para uma melhor organização de qual teste está sendo executado, é necessário deixar claro quais testes validam positivamente e quais validam negativamente o software desenvolvido. Também se faz necessário revisar cada resultado dos testes executados em cada ciclo de testes, para que se tome conhecimento dos erros comuns, por exemplo, e que se tomem medidas para que os próximos ciclos de desenvolvimento sejam melhorados e não contenham os mesmos erros

e como fonte de inspiração para que os testadores melhorem a qualidade dos testes, validando aquilo que deu errado, com o mesmo esforço para validar coisas novas no projeto.

- "A probabilidade de encontrar falhas em um projeto de software é proporcional a quantidade de erros conhecidos vindos de projetos anteriores.". Este princípio visa condicionar as equipes de testes a serem criativas em suas atividades, que procurem melhorar seus casos de testes e não fiquem "presas" aos casos conhecidos. "Não se pode planejar o ciclo de testes, considerando a hipótese de que não haverão erros no produto." (MYERS, 2004)

2.1 O contexto do teste de software.

A impossibilidade de testar cada permutação possível no fluxo de um algoritmo complexo que compõe um produto de *software*. Além da possibilidade de os erros ocorrerem em uma etapa anterior à codificação, o que levaria, possivelmente, a um código funcional, mas que por não estar corretamente especificado, tem um comportamento divergente do esperado levou (MYERS, 2004)"*Não se pode garantir que todo software funcione corretamente, sem a presença de erros*"

Junto a esta definição, segundo o artigo "*The Economic Impacts of Inadequate Infrastructure for Software Testing*" há uma necessidade de mercado por métricas de performance, procedimentos e ferramentas que suportem atividades que assegurem a qualidade do *software* desenvolvido. Como consequência destas necessidades, as falhas em projetos de *software* podem ser separadas em categorias (TASSEY, 2002).

- Falhas por má qualidade. A mais comum no mercado, deixa claro a falta de padronização, organização e investimentos em tecnologias que assegurem a qualidade do *software*. Tem como consequência a perda de investimentos no projeto, visto que haverá reclamações sobre o produto e consequentemente a quantidade de retrabalho nas manutenções irá aumentar, pondo em risco a reputação da empresa dona do projeto perante seus clientes. São classificados por tipo, ponto onde foi introduzido (ou encontrado), nível de gravidade, frequência com que ocorre e custo associado ao prejuízo causado:
  - Falta de conformidade com as normas, quando ocorre um problema porque as funções do software e/ou o resultado de suas operações não estão em conformidade com o processo ou formato especificado por uma norma legal.

  - Falta de interoperabilidade com outros produtos, onde um problema é o resultado da incapacidade de um produto de software de trocar e compartilhar informações com outro produto, quando este é um requisito para tal.
  - Mal desempenho, onde a aplicação funciona, mas não como planejado/esperado.

- Aumento dos custos do desenvolvimento do projeto. Entende-se que quanto mais cedo as falhas forem conhecidas, menores serão os custos para sua correção. “O processo de identificar e corrigir defeitos durante o processo de desenvolvimento de software representa mais da metade dos custos de desenvolvimento. Dependendo dos métodos contábeis utilizados, as atividades de teste representam 30 a 90% do trabalho gasto para produzir um programa de trabalho” (BEIZER, 1990)

- Tempo para entrar no mercado. A consequência da demora de um produto ser disponibilizado no mercado, por conta das correções de falhas conhecidas é a perda de oportunidades de negócio. Estas oportunidades podem significar a sobrevivência da organização no mercado, pois não podem ser quantificadas monetariamente e dificilmente serão recuperadas quando o software entrar em operação.

Assim, economicamente os testes devem ter sua prioridade considerada, quando se fala de um produto que será consumido em qualquer escala que seja, e para isso devem se tomar medidas para que este tenha qualidade para se manter no mercado e torne sustentáveis seus custos de operação e manutenção. Para garantir esta sustentabilidade, é preciso ter os testes de *software* como *“(. . .) uma tarefa técnica, mas que também envolve algumas considerações importantes de economia e psicologia humana”* (MYERS, 2004)*,* dado que como citado anteriormente, é impossível testar todas as variações de código dentro de um produto de *software* e as consequências de um ciclo de testes mal feito, pode levar a perda de oportunidades e de espaço no mercado.

Como forma de se prevenir das perdas de um ciclo de testes ruins, é necessário planejar sua execução, adotando estratégias para encontrar a maior quantidade de falhas possíveis e agregar valor ao produto.

2.2 Níveis de teste de software.

Para melhor andamento do projeto de *software*, e elevar o grau de segurança e qualidade do produto gerado, os testes progridem em níveis diferentes, para cada estágio da metodologia de desenvolvimento adotado na organização. Para fins de comparação, iremos explorar, de acordo com o *“Software Engineering Body of Knowledge” (*SWEBOK*)* (BOURQUE; FAIRLEY, 2014) três níveis de testes reconhecidos por serem

distinguidos pelo alvo do teste, sem necessariamente impor um processo de teste específico ao projeto em desenvolvimento, são eles:

2.2.1 Testes Unitários

São as validações de cada item básico para criação do sistema, através dele cada classe de um componente podem ter sua funcionalidade validada individualmente (HUIZINGA; KOLAWA, 2007). São pequenos trechos de códigos, idealmente autônomos e isolados de outros testes unitários criados por desenvolvedores para assegurar que o esforço realizado por eles, atingiu as metas de qualidade conforme os padrões estabelecidos pela metodologia adotada.

Tem como benefícios a descoberta de erros antecipadamente com correções igualmente rápidas para que o código a integrado ao sistema seja confiável. Como veremos em capítulos posteriores, esta prática é fundamental em algumas técnicas de desenvolvimento de software, tais como o TDD e o BDD, sendo também priorizada no XP. É uma forma de documentação indireta do produto em desenvolvimento, visto que através de uma classe de teste unitário bem construída, pode-se inferir o comportamento esperado da funcionalidade em teste. E com um produto bem documentado, o impacto das mudanças que por ventura ocorrerem no projeto tem sua extensão facilmente medida pela quantidade de testes que falham após sua implementação. Porém, no contexto geral são limitados por sua definição, visto que não conseguem encontrar falhas em classes não cobertas, o que para um programador iniciante pode ser um desafio, já que este não terá a expertise para estabelecer um limite de cobertura dos testes produzidos.

2.2.2 Testes de Integração

São testes que procuram validar as integrações em qualquer componente do software, sejam elas as camadas de uma arquitetura, acesso aos dados, interface com o usuário, módulos que operam em sincronia, etc., combinando-os e testando como um grupo (BEIZER, 1990). Podem ser realizadas incrementalmente ou com todos os módulos de uma vez (veremos com mais detalhes na seção sobre “Abordagens de teste de software”).

Tem como proposito validar as funcionalidades especificadas para o produto (testes em “caixa-preta”), assim como sua performance (testes de carga) e confiabilidade (testes de recuperação). Como é um teste funcional, desenrola-se fornecendo dados como entrada para o sistema e analisando as saídas obtidas com a execução do software. Consequentemente, tem como limitações a documentação (ou falta dela) do projeto, ou seja, para se ter testes um de integração consistente e confiável, é necessário um trabalho anterior com a documentação correta e precisa do projeto.

Um ponto que merece destaque nos testes de integração, são os testes de interfaces do sistema que buscam validar as conexões entre módulos que trabalham em conjunto, podem ser APIs, WebServices ou strings de conexão que apesar de não terem interface com o usuário necessitam de um dado como entrada para gerar uma saída. Para realizar estes testes, não é necessário validar a funcionalidade do componente como um todo, apenas ter uma especificação precisa da entrara necessária para a saída esperada. Por exemplo, para validar um módulo que tenha um arquivo XML que serve como entrada e retorne um arquivo JSON, não precisamos validar o funcionamento deste módulo, apenas ter as especificações do XML e do JSON para assegurar que a integração será realizada com sucesso.

2.2.3 Testes de Sistema

Após o sistema estar completamente integrado e funcionalmente estável, é necessário realizar as validações de negócio em um ambiente próximo ao que será utilizado pelo cliente, este é o teste de sistema, ou fim a fim (IEEE, 1990). Porém, não somente realizar os testes das funções específicas no sistema como um todo, pois isso seria um retrabalho do esforço realizado em níveis anteriores, nesta fase se compara o resultado do projeto, com os objetivos funcionais e/ou funcionais estabelecidos em seu início (MYERS, 2004).

Tendo estas definições em mente, ficam claras as limitações deste. O teste não pode limitar-se ao sistema em si, deve incluir todo o processo para demonstrar como o produto alcança as expectativas especificadas, isso inclui os requisitos não funcionais, como usabilidade, confiabilidade, disponibilidade, etc. O teste, por definição, não pode ser executado caso não haja uma documentação dos objetivos esperados de forma mensurável.

Também pode ser utilizado como testes de aceitação, quando o cliente toma o controle das ações durante a execução do teste no sistema, e por isso é o último nível de testes antes do *release*.

2.3 Abordagens de teste de software.

As abordagens para testes de software podem ser subdivididas em duas categorias, os testes estáticos que focam em verificação dos resultados através de revisões e/ou inspeções de código. Este é implícito ao dia a dia do programador pois é executado por seus pares (no caso da revisão/inspeção de código) ou pela própria ferramenta de codificação com plug-ins que verificam a estrutura do código fonte, a sintaxe e fluxo de dados ou compiladores (pré-compiladores).

Já os testes dinâmicos envolvem a execução de casos de testes previamente

criados no programa (mesmo que parcialmente) em execução. É necessário que estes testes ocorram mesmo antes de o produto estar completamente acabado para testar determinadas seções de código e sejam aplicadas a funções ou módulos discretos utilizando stubs em ambiente controlado.

Estas duas categorias são tradicionalmente chamadas de testes “caixa-preta” e “caixa-branca” e são usadas para descrever o ponto de vista de quem está testando a aplicação no momento.

2.3.1 Caixa – Preta vs Caixa – Branca

Também chamada de teste estrutural ou orientado à lógica, a técnica de caixa-branca avalia o comportamento interno do componente desenvolvido. É chamado de caixa-branca para denotar a transparência de ter contato direto com o código fonte e dessa forma pode executar testes de condição, testes de fluxo de dados, testes de caminhos lógicos, cobertura de código, etc. “*O teste final de caixa branca é a execução de cada caminho no programa, mas o teste de caminho completo não é um objetivo realista para um programa com loops*” (MYERS, 2004).

Este tipo de teste é desenvolvido analisando o código fonte e elaborando casos de teste que cubram todas as possibilidades do componente de software. Dessa maneira, todas as mutações relevantes originadas por estruturas de condições, repetições, caminhos e códigos nunca executados são testadas. Como exemplo prático para esta abordagem de teste, é o uso de ferramentas xUnit (nome genérico para qualquer estrutura de testes automáticos unitários) nas aplicações desenvolvida (BRANDÃO et al., 2005)*plug-ins* nas ferramentas de desenvolvimento. O uso desta abordagem, normalmente, é de responsabilidade dos desenvolvedores que por conhecer o código fonte produzido deve ser responsável pela qualidade de sua porção de código no sistema e é recomendado durante as fases de testes unitários e de integração.

Já os testes em “caixa-preta” também são conhecidos como testes funcionais, comportamentares, orientados a dados ou a entradas e saídas, pois avaliam o comportamento da aplicação em execução, sem acesso ao código desenvolvido. Os métodos de teste de caixa preta incluem: particionamento de equivalência, análise de valor limite, tabelas de transição de estados, testes de tabelas de decisão, testes de casos de uso, testes exploratórios e testes baseados em especificações, etc. Nesta abordagem “(. . .) *Os testadores estão apenas conscientes do que o software é suposto fazer, não como ele faz isso*.” (PATTON, 2005)

Como os detalhes da implementação não são considerados, os testes em caixa preta são todos derivados da documentação do sistema, fornecendo entradas variadas e avaliando as saídas, assim, quanto mais entradas forem fornecidas, mais rico será o teste. Assim como na caixa branca, a situação ideal é utópica pois é impossível

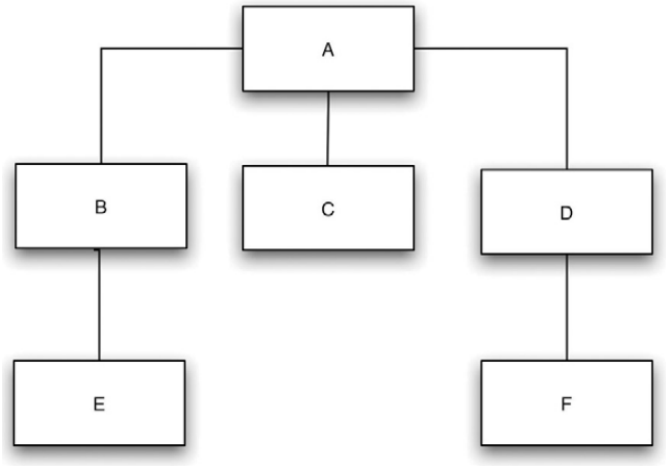
testar todas as entradas possíveis. Por isso, a abordagem realista é selecionar as entradas que potencialmente trarão mais risco as falhas do projeto, mostrando os erros, este é um ponto de dificuldade na implementação pois não é trivial encontrar as entradas perfeitas para as execuções. Outra dificuldade está na completude da documentação da aplicação, dado que estes testes derivam da especificação planejar os testes assumindo como hipótese uma documentação falha, trará testes pobres que não serão suficientes para mostrar os erros do sistema. Pode ser aplicado em qualquer fase de teste, mas isoladamente é insuficiente para identificar todos os riscos do projeto de software (BACH, 1999).

2.3.2 Incremental

De acordo com (MYERS, 2004) a grande questão para se compreender o conceito de teste incremental é: Devemos testar cada módulo do programa isoladamente e depois de finalizados os testes, adicionar novos módulos a serem testados ou devemos combinar o próximo módulo do planejamento de testes ao passo que executamos o teste de um módulo específico? A abordagem destacada na primeira pergunta define a abordagem “monoincremental” de testes e a segunda define a abordagem incremental. Para exemplificar este conceito, consideramos um programa composto por seis módulos as conexões da figura 3.1 representam a hierarquia entre eles, dessa forma, uma ação no módulo A, invoca o funcionamento dos módulos B, C e D, o módulo B “chama” o módulo E, e finalmente o módulo D “chama” o módulo F.

A abordagem “monoincremental” testa isolada e consecutivamente cada componente do sistema como um programa autônomo, para isso, faz uso de pequenos programas para criar a chamada do módulo em teste e um ou mais pequenos programas para validar a saída gerada pelo módulo estes pequenos programas são chamados de drivers e stubs consequentemente. Ao final do ciclo, quando todos os 6 módulos foram testados, há um novo ciclo para testar a integração entre eles. A abordagem incremental preza por combinar o módulo em teste aos módulos previamente testados realizando simultaneamente um teste de integração entre esses módulos.

Figura 2 – Exemplo de um sistema composto por seis módulos



Claramente há uma carga de trabalho maior na realização dos testes “monoincrementais” pois para cada módulo em teste, deve se criar um *driver* e pelo menos um *stub* para cada módulo e, somente ao final, rodar uma nova bateria de testes com todo o sistema devidamente integrado, em qualquer técnica da abordagem incremental este trabalho é simplificado, por conta do aproveitamento dos módulos anteriores, é necessária a confecção de apenas cinco *drivers* (sem nenhum *stub*) no caso da estratégia *botton-up* ou de cinco *stubs* (sem nenhum *driver*) no caso da abordagem *top-down*, além da facilidade de ter realizado testes de integração a cada novo módulo em teste.

2.3.3 Top-Down vs Botton-Up

Mais que uma abordagem, é considerado como um método de ordenação do conhecimento, usado em várias campos, incluindo software, humanística e teorias científicas, na prática, eles podem ser vistos como uma abordagem de pensamento e ensino. Segundo (MYERS, 2004)é necessário esclarecer que as abordagens de testes, assim como a de desenvolvimento top-down são similares, pois partem do mesmo princípio de ordenação dos códigos de teste e desenvolvimento (respectivamente) em módulos isolados, dessa forma, um programa que foi desenvolvido seguindo a abordagem top-down pode ser incrementalmente testado em qualquer abordagem.

A abordagem *top-down* como o próprio nome diz, sugere destrinchar o sistema a partir do topo até alcançar as camadas mais distantes, em outras palavras, começa pelo módulo de mais alto nível na aplicação e flui “chamando” seus módulos adjacentes que ainda não tenham sido testados anteriormente até alcançar os componentes mais internos à aplicação, sempre obedecendo a relação de que o método em teste, deve “chamar” o próximo a ser testado.

Assim, para execução dos testes *top-down*, deve-se realizar as tarefas do sistemas dando entradas conhecidas de modo que a funcionalidade principal invoque a execução das funcionalidades mais internas do software e assim sucessivamente, até que todo o sistema tenha sido alcançado, fazendo uso de *stubs* para simular o comportamento esperado de algum método que ainda não tenha sido completamente desenvolvido. É importante ter em vista, que no final do teste teremos a integração dos módulos testados e não o sistema como um tudo.

É vantajoso pois permite a verificação antecipada de comportamento de alto nível, não interrompe o andamento do processo de desenvolvimento pois métodos podem ser adicionados, um por vez, em cada passo e valida a cobertura do código através da busca em profundidade e/ou da busca em largura nos componentes do software. Porém, retarda verificação de comportamento de baixo nível, as entradas de casos de teste podem ser difíceis de formular, por geralmente, demandar de uma quantidade maior de informações e uso de stubs e as saídas de casos de teste podem ser difíceis de interpretar, cada passo executado, pode gerar um resultado diferente e podem haver inconsistências nessas saídas que devem ser validadas.

Já a estratégia *botton-up* diz que os testes devem começar a partir do módulo mais básico da aplicação, ou seja, aquele que não tem dependência com nenhum outro, assumindo que estes módulos já tenham sido previamente testados. Para isso, se faz uso de "*drivers*" que invocam a execução do próximo módulo a ser testado. Quando forem concluídos os testes, o driver é substituído pelo próximo modulo de teste e utiliza-se um segundo driver para simular a "chamada" do próximo, e assim sucessivamente até que todos os módulos tenham sido validados.

Como o processo de implementação é exatamente o contrário da abordagem top-down, seus benefícios e críticas também são opostos. Como comparativo, há de se considerar que a validação das principais decisões, normalmente, estão contidas nos módulos de mais alto nível que são testados primeiro na abordagem *top-down* que também favorece os testes de regressão visto que seus testes tem a interface do sistema como ponto de partida, no entanto, na abordagem botton-up a elaboração de casos de testes é simplificada por um driver que simula o comportamento real de um módulo.

2.4 Processos para Teste de Software.

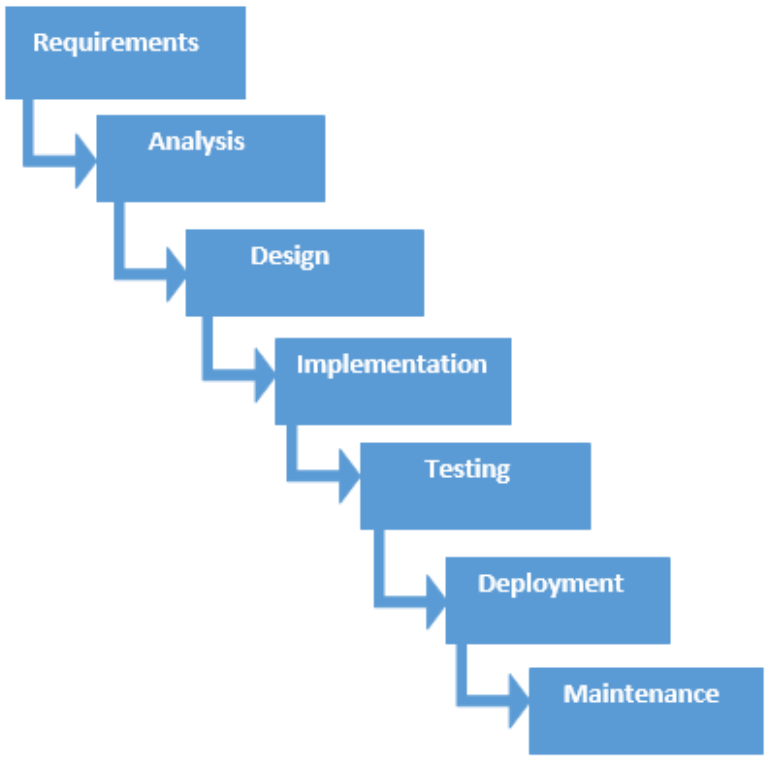
Segundo o Livro "Qualidade de Software – Teoria e Prática" (ROCHA; MALDONADO; WEBER, 2001) o planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software, tendo sua execução realizada no sentido inverso como bem resumiu (CRAIG; JASKIEL, 2002), porém como vamos explicar durante esta seção, estes pensamentos nem sempre foram o padrão adotado no mercado

de software.

2.4.1 O Modelo Cascata

Também conhecido como “linear” o modelo em cascata é um modelo de desenvolvimento sequencial, onde a próxima fase do processo não se inicia até que a fase anterior esteja completamente finalizada proposto por (ROYCE, 1970). Pode ser bem sucedido em projetos com poucos requisitos e que não sofram mudanças não planejadas depois do início da codificação.

Figura 3 – Ciclo de desenvolvimento do m*odelos Cascata*



Como opera de forma sequencial, todas as atividades de testes iniciam-se apenas quando o projeto já foi completamente codificado essa rigidez traz riscos ao projeto, pois, caso alguma não conformidade seja encontrada no código, todo o ciclo deverá ser refeito para sua correção, elevando os custos do projeto por conta do retrabalho a ser realizado. Em projeto com várias falhas, e consequentemente muito esforço de retrabalho, o orçamento/ cronograma negociados podem ser comprometidos, “encurtando” a fase de testes ou gerando custos para o cliente em uma possível negociação de “aditivos contratuais” para finalização do projeto.

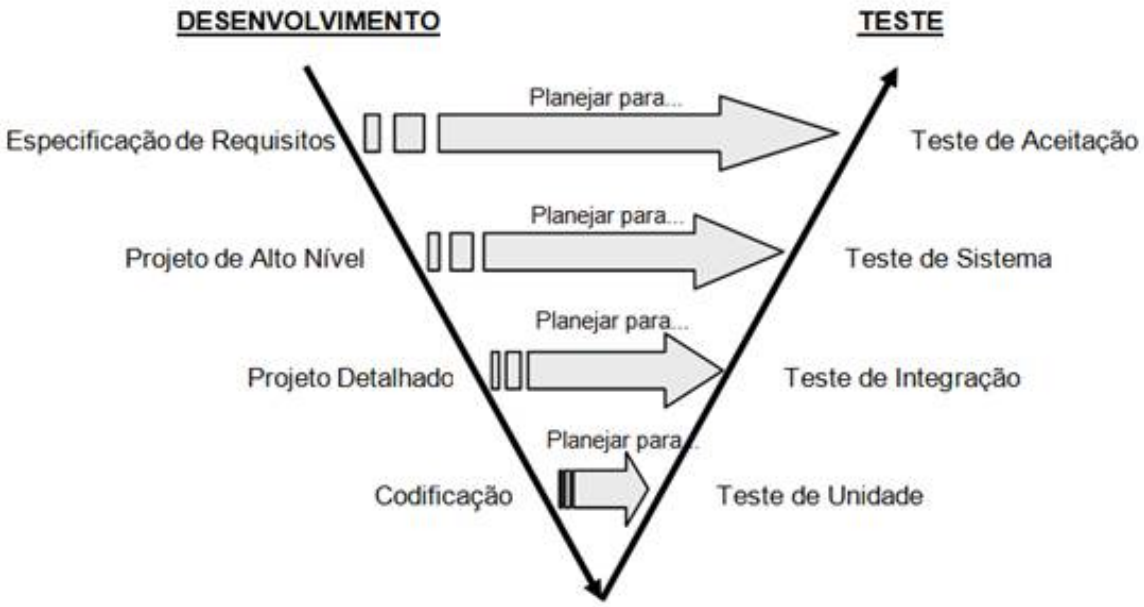
Apesar da facilidade em seguir este modelo e por ser de fácil negociação com clientes pois cada fase pode ser mensurada em tempo e custo com garantias de que sejam completadas para que o projeto seja entregue, os riscos descritos no

parágrafo anterior, mostram que esta metodologia não é efetiva em projetos grandes e de requisitos complexos, não ter flexibilidade para se adaptar as mudanças arrisca toda a negociação para o desenvolvimento do software.

2.4.2 O Modelo V

No modelo em V (ou Verificação e Validação) cada tarefa de teste segue uma sequência lógica em paralelo ao processo de desenvolvimento do software em questão, como bem resumiu (CRAIG; JASKIEL, 2002) no seu livro "*Systematic Software Testing*" e mostrado na Figura 4.

Figura 4 – Descrição do Modelo V



Neste modelo V os níveis de testes propostos da seguinte forma:

- Teste de Unidade: também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código.
- Teste de Integração: visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.
- Teste de Sistema: avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados

nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software. Verifica se o produto satisfaz seus requisitos.

- Teste de Aceitação: são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.

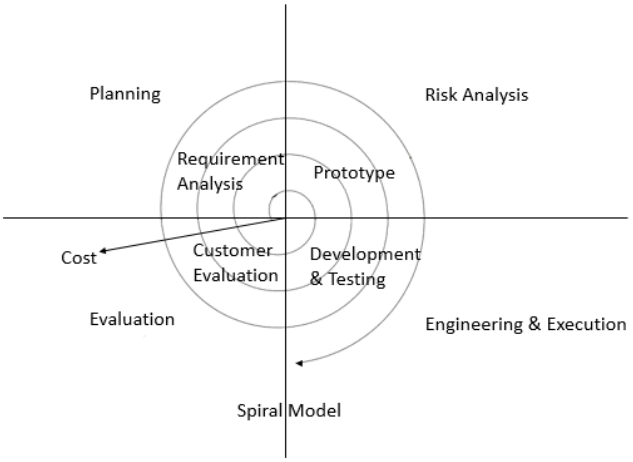
Este modelo V propõe um processo para testes e as suas atividades em paralelo ao desenvolvimento de software. Entretanto, outras etapas, e consequentemente tipos de testes, existem para outros fins como por exemplo, os testes de regressão que não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Consiste em se aplicar, a cada nova versão do software ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de testes anteriores do sistema garantindo que a nova versão do não tenha novos bugs em partes onde o mesmo já estava funcionando adequadamente.

Pode ser aplicado em projetos pequenos, com requisitos de fácil compreensão pelo time de desenvolvimento e que seja estável, ou seja, que não seja propenso a grandes mudanças durante o ciclo de desenvolvimento, pois este modelo não é flexível a mudanças e caso elas ocorram, haverá custos elevados de retrabalho.

2.4.3 O Modelo Espiral

É uma estratégia de testes que se beneficia do modelo incremental apresentado por (BOEHM, 1988) apesar de antigo, continua bem aceito no mercado, tanto que é a base para frameworks modernos com abordagens de desenvolvimento e teste ágil de *software*. Neste modelo todas as cinco fases (planejamento, análise de risco, execução, validação e avaliação) são executadas para cada tarefa do desenvolvimento do produto, desta forma, cada nova tarefa se vale dos conhecimentos adquiridos na fase anterior e há um espaço para replanejamento em caso de mudanças nos requisitos ou descoberta de uma falha crítica no sistema.

Figura 5 – Diagramação do processo no modelo Espiral



Seguindo a lógica do modelo, as tarefas de teste serão realizadas em todas as fases do desenvolvimento para cada item desenvolvido. Todos os requisitos são especificados na fase de planejamento, na fase de análise de riscos todas as possíveis dificuldades que serão encontradas no sistema são elencadas tendo como base as especificações levantadas na fase anterior, assume-se que todos os riscos podem acontecer durante as fases de codificação e testes do projeto. Na fase de engenharia e execução, todos testes são executados nos itens desenvolvidos e em desenvolvimento e o projeto é analisado para avaliar se há alguma mudança significativa no contexto do projeto ou alguma não conformidade com o que foi especificado, o resultado dessa análise serve como entrada para a fase de avaliação onde todo o progresso do projeto é revisado para propor melhorias ao próximo ciclo de desenvolvimento.

Faz sucesso por que fornece garantias para análise de riscos e gerenciamento das mudanças que tanto ocorrem nos projetos de desenvolvimento de software, a análise de requisitos é feita com esforços concentrados, mas que são melhorados a cada iteração e tem uma implantação de melhor qualidade, já que cada item é planejado, desenvolvido, testado e avaliado mais de uma vez durante o processo. Porém é a implementação deste modelo no time é de complexa adaptação para colaboradores pouco experientes, gera muito esforço que não resulta em código (documentação excessiva) e não é útil para projetos de risco baixo e poucos requisitos.

2.5 Abordagem Tradicional para Teste de Software.

Dadas todas as classificações anteriores, as metodologias tradicionais para testes de software se caracterizam por manter um processo rígido e de pouca tolerância as mudanças de que podem surgir durante o desenvolvimento do projeto gerando um potencial alto índice de retrabalho, além de exigir um esforço muito grande para

documentação do projeto, que leva a um tempo ocioso dos desenvolvedores e um menor tempo para testes da aplicação.

De um modo geral, nas abordagens tradicionais a atividade de teste é realizada ao fim da versão ou do desenvolvimento de uma porção significativa do código, onde o analista de teste estuda os requisitos a partir das especificações. Os artefatos gerados orientam o profissional de teste durante a execução de tais testes com o intuito de apurar se o produto está em conformidade com as especificações. Ao identificar um problema na fase de teste, o gerente de projeto tem de avaliar se há espaço no cronograma para correção e reteste (para avaliar o ponto de falha e assegurar sua correção) ou negocia-se um "aditivo contratual" para que as falhas encontradas, sejam solucionadas. De toda forma, a nova demanda retorna para o início do fluxo de desenvolvimento e impacta no cumprimento do prazo de entrega. O custo do projeto torna-se alto, pois os ciclos de desenvolvimento normalmente são longos, a quantidade de retrabalho é potencialmente alta e o risco de se desenvolver um produto que o cliente não está esperando também é alto.

3 Metodologias Ágeis

Segundo Drucker, em seu livro “Administrando em Tempos de Grandes Mudanças” (DRUCKER, 1999), existem duas categorias de trabalhadores o trabalhador manual e o trabalhador do conhecimento. Um trabalhador manual executa atividades que dependem basicamente de habilidades manuais e que não se baseiam no uso intensivo do conhecimento, em contrapartida, o outro é aquele que produz com base no uso intensivo da criatividade e do conhecimento. A principal característica que difere essas duas categorias é a forma com que cada um reage a mudanças e falhas durante o processo de trabalho. Para os trabalhadores manuais, falhas e mudanças devem ser constantemente evitadas, e processos rígidos e determinísticos (como os modelos de desenvolvimento em cascata e espiral) ajudam a alcançar este objetivo. Já para atividades que envolvem trabalhadores do conhecimento, erros devem ser encarados como coisas naturais, saudáveis e até inevitáveis. Neste caso, tentar sistematizar ou impor metodologias rígidas ao processo de desenvolvimento, visando o determinismo, no máximo torna as pessoas envolvidas defensivas e pouco criativas, para estes trabalhadores as metodologias ágeis se encaixam melhor.

No desenvolvimento de software encontramos essa dualidade de processos rígidos versus processos mais flexíveis ao confrontar as metodologias tradicionais (rígidas) e as metodologias ágeis. “*Metodologias ágeis são usadas para otimizar o processo de desenvolvimento. O objetivo delas é que software de qualidade seja produzido em curtos intervalos de tempo*” (LIVERMORE, 2007). Ou seja, se configuram como uma nova forma de gerenciamento e desenvolvimento de software, que foca sua abordagem e planejamento para os processos empíricos.

Diversas iniciativas independentes para flexibilizar o desenvolvimento de software se desenvolviam paralelamente e de forma isolada, entretanto foi necessário uma reunião em 2001, onde criadores destas iniciativas unificaram, através do manifesto ágil de software, um conjunto de princípios que deveriam reger esses novos métodos mais flexíveis. O objetivo era mitigar o risco dos projetos associados a “burocracia” do desenvolvimento da forma tradicional, onde as metodologias eram ditas “pesadas”, pois havia muito peso na forma de criar a documentação e no excesso de atividades que não resultavam em código e, por isso, boa parte do tempo era gasto na fase de planejamento de como o sistema deveria ser desenvolvido e a cada mudança, o retrabalho se tornava custoso.

Neste encontro, foram expostas as principais dificuldades enfrentadas pelas equipes de desenvolvimento e apresentadas potenciais soluções para uma abordagem bem sucedida. E em um estudo destas soluções, detectou-se pontos em comuns nas

abordagens utilizadas o que resultou em um acordo de implantação das ideias ágeis em quatro níveis (KOCH, 2005).

No primeiro nível, reconhece o problema e a necessidade de modificação do modelo de implementação para atender às novas expectativas de mercado, mudanças e adaptações exigidas nos projetos. O termo "Ágil" foi adotado por questões de mercado, pois denota velocidade aos projetos desenvolvidos utilizando a técnica proposta. O segundo nível torna explícito em forma de um manifesto (BECK et al., 2001) os valores centrais que devem sedimentar todas as técnicas que se denominam ágeis, são elas:

- "Indivíduos e interação entre eles mais que processos e ferramentas.
- Software em funcionamento mais que documentação abrangente.
- Colaboração com o cliente mais que negociação de contratos.
- Responder a mudanças mais que seguir um plano."

Onde o pensamento é de que mesmo havendo valor na entrega dos itens à direita, os itens mais valorizados do ponto de vista ágil estão à esquerda de cada princípio. O terceiro nível procura detalhar os valores descritos no manifesto em um documento (BECK et al., 2001) quebrando os valores citados anteriormente em doze princípios de modo que seja mais simples a implementação por quem deseje adotar a metodologia, são eles:

- "Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de software de valor.
- Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas.
- Entregar software funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.
- Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.
- Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.
- O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
- Software funcional é a medida primária de progresso.

- Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.
- Contínua atenção à excelência técnica e bom design, aumenta a agilidade.
- Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.
- As melhores arquiteturas, requisitos e designs emergem de times auto organizáveis.
- Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo."

Estes princípios deixam claro os conceitos e práticas que devem ser adotados para implementar a metodologia ágil em projetos de desenvolvimento de software. Em resumo, a ideia geral é fazer uma segmentação do problema em elementos menores tornando processo de desenvolvimento mais leve para seus participantes, prezando pela comunicação e colaboração entre a equipe de desenvolvimento e os especialistas de negócios, visando minimizar os riscos relacionados a incertezas do projeto, entregas constantes e com participação do cliente, otimizando a adaptação a imprevistos e mudanças no projeto. O quarto nível é livre para que cada abordagem ágil defina suas próprias características e procedimentos que melhor se adequam aos problemas enfrentados. Fazendo uso desta abertura abordagens como *eXtreme Programming* (Xp), Scrum, TDD, DDD, BDD e outras técnicas acrescentam suas especificidades.

Na aplicabilidade destas metodologias, alguns fatores devem ser levados em consideração, segundo (COHEN; LINDVALL; COSTA, 2004) em seu livro "*An introduction to agile methods. In Advances in Computers*" os principais fatores a serem considerados são: O produto, onde as metodologias ágeis são mais eficazes em projetos em que os requisitos ainda estão sendo especificados. A organização, onde devem ser medidas a cultura, as pessoas envolvidas e os métodos de comunicação entre elas, a cultura da empresa deve ser aberta o suficiente para criar um ambiente favorável à negociação, as pessoas confiantes e competentes na execução de suas atividades e a comunicação deve ser rápida e eficiente entre os envolvidos. Finalmente, o fator mais importante, é o tamanho do projeto. O autor cita que em projetos com mais de 40 pessoas, inviabiliza as práticas de reuniões rápidas diárias presentes na maioria dos frameworks ágeis em voga. No entanto o artigo "*New Directions on Agile Methods: A Comparative Analysis*" (ABRAHAMSSON et al., 2003) cita o *Dynamic Systems Development Method* (DSDM) e o *Feature Driven Development* (FDD) como exemplos de

metodologias ágeis que podem ser aplicadas a qualquer projeto independente de suas características.

Por ser mais recente e ter sido uma proposta de mercado, as metodologias ágeis desde sua concepção buscam trazer uma alternativa aos principais pontos de divergências nos projetos antes gerenciados com metodologias tradicionais. Por exemplo, nos métodos tradicionais de gestão, entende-se que o produto só faz sentido quando é entregue em sua totalidade, ou seja, apenas com 100% do projeto cumprido é que o cliente perceberá algum valor. Por outro lado, métodos ágeis permite que um conjunto de funcionalidades que solucione parte da necessidade do cliente ao ser entregue, mesmo que em parte, já representa uma diferença valorosa para ele. Outro ponto importante na comparação entre os dois métodos é que, no ágil, quando combinado que o projeto irá entregar as funcionalidades mínimas, o cliente nem sempre têm noção do custo total do produto. Já nos métodos tradicionais, o valor é fechado junto com o escopo, o que sugere que não estão previstas alterações significativas nesse quesito durante o andamento das ações, ou seja, as mudanças no projeto terão seu custo mensurados a parte.

Então podemos perceber que as metodologias ágeis vem propor uma nova visão para o desenvolvimento de sistemas mais voltada ao produto entregue e nas atividades que contribuem de fato para a construção do produto final, eliminando o excesso de esforço, que não é necessário ao longo da concepção de um produto. E por se refletir em todas as etapas do desenvolvimento de software (desde o planejamento até a implantação) esses princípios ágeis atingem também a parte de testes de software como veremos a seguir.

3.1 Teste Ágil

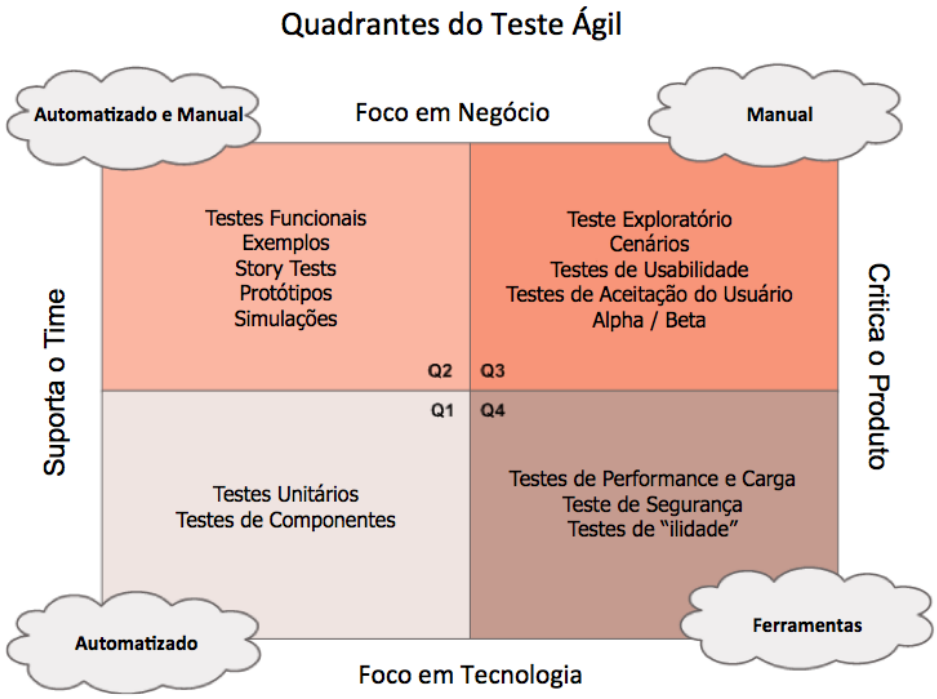
Com o surgimento Manifesto Ágil, que afetava todas as etapas do ciclo de vida do desenvolvimento, os testes também foram alvo de novas ideias que incorporam muitas das técnicas de teste utilizadas nas abordagens tradicionais, mas, passaram a considerar o teste antes (e não ao mesmo tempo) do desenvolvimento e em como eliminar a burocracia em realizar os testes (utilizando ferramentas de automação). Nos testes, a principal função é de avaliar o produto, ou seja, constantemente garantir que o que está sendo especificado e desenvolvido realmente atende as necessidades do cliente e irá entregar valor ao negócio. E tem como característica ser uma atividade desempenhada por todos os membros do time e que ocorre em todas etapas do ciclo de vida de desenvolvimento, através de mecanismos automatizados (quando possível), frequentemente e em ciclos contínuos (CAETANO, 2016).

Nas metodologias ágeis, todos são responsáveis pelos testes. Cada membro do time contribui para a qualidade do software realizando testes sob a sua perspectiva.

Dessa forma, os testes ocorrem de maneira colaborativa e complementar, onde os clientes testam sob a perspectiva da funcionalidade (estória por estória) e os desenvolvedores testam sob a perspectiva do código (método por método).

A atividade de teste no contexto ágil está focada na validação do negócio. Um de seus objetivos é realizar uma análise crítica do produto para tentar descobrir pontos de melhoria que agreguem valor ao produto final, o analista de teste possui um papel mais dinâmico e crítico dentro do time de desenvolvimento, uma vez que ele tem a visão do todo e poderá contribuir ativamente na qualidade do produto a ser entregue. Assim, fazem uso dos testes funcionais manuais para validar os requisitos e a partir destes conseguir criar testes automatizados para ser adicionados à suíte de testes de regressão. Nesta forma de trabalho, destaca-se o Quadrante do Teste Ágil (CRISPIN; GREGORY, 2009) onde os testes são subdivididos elevando a participação e posterior qualidade técnica do profissional que os executa, ideia que pode ser vista na Figura 6 a seguir.

Figura 6 – Quadrante do Teste Ágil



No Q1 temos duas práticas: teste de unidade, que valida uma pequena parte da aplicação como objetos e/ou métodos, e testes de componente que valida partes maiores da aplicação como um grupo de classes que provê o mesmo serviço. São usualmente desenvolvidos com ferramentas xUnit e medem a qualidade interna do produto. Não são destinados ao cliente, pois o cliente não irá entender os aspectos

internos do desenvolvimento e não devem ser negociados com o cliente, pois a qualidade do código deve sempre existir e não ser negligenciada. Os testes desenvolvidos usualmente executados dentro de uma abordagem de Integração Contínua para prover um rápido *feedback* da qualidade do código.

Já o Q2 os testes continuam guiando o desenvolvimento, mas de uma maneira mais alto-nível focando mais em testes que o cliente entenda, onde este define a qualidade externa de que ele precisa. Aqui o cliente define exemplos que serão usados para um entendimento maior do funcionamento da aplicação, e são escritos de forma com que o cliente ou papéis ligados ao negócio entendam. Todo o teste executado aqui tem um foco no funcionamento do produto e alguns deles podem até mesmo ter uma pequena duplicação com alguns testes criados no Quadrante 1. Também pode-se utilizar de recursos com conhecimentos em UX (*user experience*) para que, através de mockups e wireframes, o cliente possa validar a interface gráfica antes que o time comece a desenvolver esta camada.

No Q3 a intenção é criticar o produto e executá-lo como um usuário real usando nosso conhecimento e intuição na utilização da aplicação. O cliente pode executar este tipo de tarefa, usualmente chamada de *User Acceptance Testing* (UAT), dando um *feedback* mais preciso, aceitando a funcionalidade, analisando possíveis novas funcionalidades. Esta ação pode ser também um dos critérios de *Definition of Done* (DoD) de uma funcionalidade. O ponto central deste quadrante, além do UAT, são os testes exploratórios. Utilizando esta técnica qualquer membro do time é capaz de, simultaneamente, aprender sobre a aplicação e executar mais testes, usando o *feedback* do último teste para a execução dos próximos e também é capaz de extrair novos critérios, sempre observando o comportamento da aplicação.

No Q4 os testes são mais técnicos e criticam o produto em termos de performance, carga e segurança. Nos dias de hoje negligenciar aspectos como performance podem tirar a vantagem competitiva de um cliente. Geralmente aspectos já conhecidos, relacionados a performance e segurança. As técnicas aplicadas a performance, carga e segurança vão desde os níveis mais baixos (como um *profiling* de código) como a utilização de ferramentas que simulam diversos usuários simultaneamente.

Na visão ágil, testes de software não está ligado apenas a encontrar mais defeitos, é, principalmente, uma questão de não introduzir defeitos. Por isso é essencial que as equipes de desenvolvimento sejam capazes de reduzir a incidência e os custos associados a depuração e correção dos mesmos. Neste contexto, a mudança de perspectiva em relação ao teste de software constitui umas das grandes diferenças entre as metodologias ágeis em relação a metodologias tradicionais de desenvolvimento de *software*.

3.2 *Test-Driven Development* - TDD

Como dito em capítulo anterior, os métodos ágeis de desenvolvimento de *software* surgiram para resolver diversos problemas que surgem por conta do alto grau de incerteza dos processos tradicionais de desenvolvimento, como consequência das muitas mudanças de requisitos, escopo ou prazos.

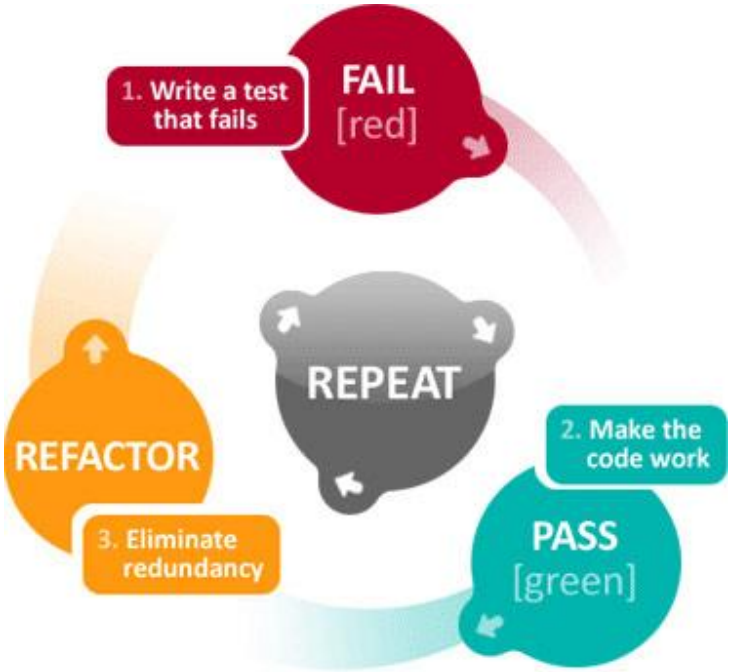
Essa técnica geralmente é associada com métodos ágeis, em particular com o *eXtreme Programming*. Além disso, o TDD apresenta ciclos de desenvolvimento pequenos e os casos de testes são escritos antes mesmo da codificação da funcionalidade. Esta característica é, segundo Kent Beck, o grande benefício do uso de TDD, pois os próprios desenvolvedores são forçados a escrever seus próprios testes unitários - auxiliados por *moks* ou *stubs* - o que ajuda a evitar vários vícios de programação. Assim, devido ao fato de que nenhum código é escrito a não ser para passar em um teste que esteja falhando, testes automatizados tendem a cobrir cada caminho de código desta forma, os testes automatizados TDD tendem a ser mais eficientes: eles irão mostrar qualquer mudança inesperada no objetivo esperado do código. Isto ajuda a identificar problemas cedo, que poderiam aparecer ao consertar uma funcionalidade que ao modificada, inesperadamente altera outra funcionalidade. O *test-driven development* é uma técnica de programação popularizada por Kent Beck (BECK, 2002)(ASTELS, 2003) define TDD como uma técnica em que:

- Código somente entra em produção se houver um teste associado a ele;
- Mantém-se um conjunto exaustivo de testes do programador;
- Testes são escritos antes da implementação;
- Testes determinam o código que será escrito.

3.2.1 Ciclo de Vida

O ciclo proposto no TDD pode ser simplificado em criar um teste para a funcionalidade desejada (que inevitavelmente vai falhar), desenvolver um código que execute corretamente o objetivo esperado (de modo que o teste passe) e em seguida, refatorar o código buscando deixá-lo o mais simples e funcional possível. Apesar de parecer, não é um processo simples, pois exige que os desenvolvedores tenham pleno conhecimento do negócio, de ferramentas de testes automáticos além de seus conhecimentos em técnicas de programação.

Figura 7 – Ciclo de desenvolvimento TDD.



Com o uso continuado desta técnica, o tempo de depuração do software tende a diminuir significativamente, diminuindo os gastos da empresa consequentemente, pois o programador saberá se a funcionalidade que está sendo desenvolvida é pertinente ou se esse código vai apresentar erros futuramente.

As experiências de sucesso com a técnica apresentada levou ao surgimento de vários estilos de programação como o *Keep It Simple Stupid* - KISS, o *You Ain't Gonna Need It* - YAGNI, focados em escrever código somente para os testes passarem e consequentemente o design do sistema ser mais limpo e o *Fake Till You Make It* sugerido pelo próprio Kent Beck , que propõe uma nova rodada de refatoração para eliminar possíveis duplicações de código.

3.2.2 Benefícios vs Críticas

Na opinião de vários autores, além do próprio Kent Beck, a simples mudança de ordem no ciclo de desenvolvimento traz como benefícios ao projeto: maior simplicidade, menor acoplamento, maior coesão das classes criadas e maior cobertura dos testes criados com criação de base para testes de regressão. Entre eles Dave Astels (ASTELS, 2003) e Robert Martim

Porém existem limitações, entre elas a falsa sensação de segurança gerada pelo alto número de testes unitários, que podem resultar em menos atividades de garantia de qualidade, como testes de integração e aceitação (WASMUS; GROSS, 2007)(MASSOL, 2003), "*Test-Driven Development é uma prática de programação que instrui desenvolvedores a escrever código novo apenas se um teste automatizado*

estiver falhando, e a eliminar duplicação de código. O objetivo do TDD é um código claro que funcione".

Possuem em si um alto custo de manutenção, o que pode onerar o desenvolvimento. Por exemplo, pequenas alterações de requisitos no sistema poderem fazer vários testes falharem, tornando o trabalho a princípio simples, em algo mais complexo. Devido ao aspecto ágil de TDD, a documentação gerada não é abrangente e/ou suficientemente detalhada como nas metodologias tradicionais (DEURSEN, 2001)

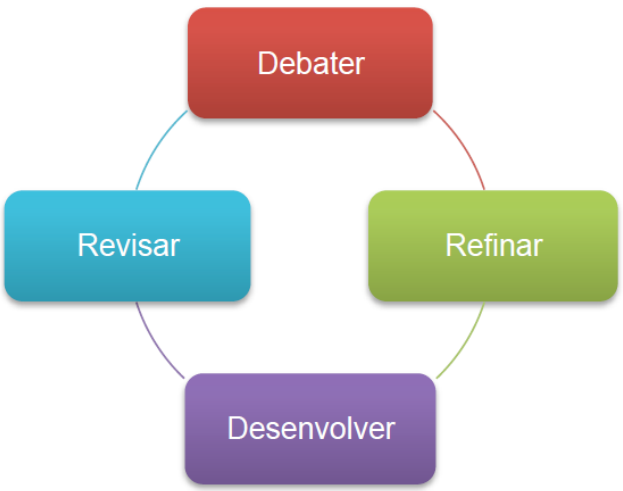
O que pode ocasionar problema para novos membros na equipe. Os testes não proveram a documentação necessária e as informações arquiteturais foram complementadas por conversas curtas com desenvolvedores (WASMUS; GROSS, 2007).

Também existem dificuldades em fazer uso do TDD em situações onde testes totalmente funcionais são obrigatórios para determinar o sucesso ou falha do projeto. Exemplos disso são interfaces de usuários, tarefas relacionadas a base de dados, e muitos outros que dependem de configurações específicas de rede.

3.2.3 *Acceptance Test Driven Development* - ATDD

Acceptance Test Driven Development (ATDD), ou desenvolvimento orientado a testes de aceitação foi proposto por(HENDRICKSON, 2008), utilizando conceitos do TDD. É uma prática onde os requisitos são obtidos de forma colaborativa onde exemplos concretos e testes automatizados são utilizados para especificar os requisitos, tornando-os mais claros, com o objetivo de criar especificações executáveis. Antes de codificar cada funcionalidade, membros do time colaboram criando exemplos. A partir daí, os times traduzem tudo em testes de critérios de aceitação. A situação ideal é que time, cliente ou pessoas experts no domínio possam escrever testes de aceitação sem consultar os desenvolvedores.

Figura 8 – Ciclo de desenvolvimento do AT*DD*



A diferença em relação ao TDD se dá pelo fato de haver uma colaboração maior entre o desenvolvedor, analista de qualidade e negócio (cliente/partes interessadas). Enquanto o teste unitário do TDD está intrinsicamente relacionado com o código, de um ângulo do desenvolvedor (visão interna), o teste de aceitação prezado pelo ATDD está voltado ao ponto de vista do usuário, em uma visão externa ao sistema. O ciclo de vida da abordagem ATDD descrito na Figura 4.2 segue um processo bem simples para guiar sua utilização desde a fase inicial do projeto até a automação dos testes e demo para o cliente.

A fase de debate, compreende um período de interação com o cliente para a obtenção dos requisitos (*user stories*), que geralmente é na reunião de preparação do backlog do produto (*backlog grooming*) onde a equipe faz uma série de perguntas para obter exemplos práticos de utilização daquele requisito para que todos possam entender melhor o que está sendo discutido e escrever este entendimento como testes. Na fase seguinte (Refinar), é o momento para formatar os requisitos especificados anteriormente em um documento que possa ser integrado a frameworks de automação de testes, Por exemplo, Fit [http://fit.c2.com/], Cocumber [http://cocumber.io/], RSpec [http://rspec.info/] e Robot Framework [http://robotframework.org/]. Desta forma a documentação poderá ser entendia por todos no time e principalmente pelo cliente.

Após a especificação e formatação dos requisitos onde todos os envolvidos no projeto já tem claro o que esperam dele e como ele vai ser implementado, a fase “desenvolver” é o momento para codificar as especificações, preferencialmente utilizando o ciclo de vida do TDD, mostrado na seção 4.1 deste trabalho e em seguida apresentar o uma “Demo” dos resultados produzidos ao cliente, que pode aprova-lo ou não.

Dentro da comunidade ágil, o ciclo de vida ATDD se confunde bastante dom o ciclo de desenvolvimento do BDD por ambas as abordagens focarem em uma relação mais próxima com o cliente em seu artigo “*ATDD vs. BDD vs. Specification by Example vs . . .*.” (TASSEY, 2002)“*Ao chamarmos de “BDD”, ou “ATDD”, ou “*Especificação por Exemplo*”, queremos o mesmo resultado - um entendimento comum compartilhado do que será construído para tentar entregar o que é certo, da primeira vez. Sabemos que nunca será, mas com menos retrabalho, ficará melhor. (. . .) Continuarei a usar o termo ‘*Acceptance Test Driven Development (ATDD)’*, a menos que a indústria decida sobre um vocabulário comum, porque acho que as áreas de negócio da empresa não só entenderão como dar exemplos, mas também compreenderão quando eu falar sobre os testes de aceitação que comprovam a intenção da história ou funcionalidade. A equipe entenderá suficientemente o escopo para então iniciar a codificação e os testes.*”. De um modo geral, as premissas e propostas são as mesmas, porém, o modo como alcançam o objetivo é diferente.

3.3 Domain Driven Design – DDD

> *"Softwares são feitos para melhorar ou ajudar algum processo que exista no mundo real. Se seguirmos essa linha de raciocínio, podemos pensar que os softwares partem de algo que já existe. Alguns softwares serão uma representação e outros vão auxiliar algo já existente. O conhecimento sobre o mundo real deve estar embutido no software. No processo de criação de um sistema temos dois papéis fundamentais: aquele que conhece o domínio muito bem e aquele que conhece a parte técnica muito bem. Esses dois papéis não podem de maneira nenhuma trabalhar separadamente. Quem entende do domínio precisa da equipe técnica para conseguir expressar seu conhecimento através de software e a equipe técnica precisa da equipe que entende do negócio para poder escrever um software útil e de acordo com a necessidade."*
> *Domain-Driven Design: Além dos Patterns,* (MOREIRA; PRADO, 2010).

Domain Driven Design significa Projeto Orientado a Domínio. Foi proposto por Eric Evans no livro "Domain - Driven Design: Tackling Complexity in the Heart of Software" (EVANS, 2004) e pode ser visto por alguns como a volta da orientação a objetos, mas, em sua essência tem coisas como:

- **Alinhamento do código com o negócio:** o contato dos desenvolvedores com os especialistas do domínio é algo essencial quando se faz DDD;
- **Favorecer reutilização:** a construção do software em módulos, facilitam o reaproveitamento de um mesmo conceito de domínio ou um mesmo código em vários lugares;
- **Mínimo de acoplamento:** Com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos;
- **Independência da Tecnologia:** DDD não foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no modelo de domínio e consequentemente no código. Não que a tecnologia usada não seja importante, mas essa não é uma preocupação de DDD.

É importante ressaltar que DDD não se trata de um framework nem uma metodologia, mas sim de uma disciplina ou abordagem para a modelagem de software, ou seja, uma série de ideias e princípios que ao serem aplicados no processo de desenvolvimento de software podem simplificar o mesmo e agregar mais valor ao produto final. A principal ideia do DDD é a de que o mais importante em um software não é o seu código fonte, arquitetura utilizada, nem a tecnologia sobre a qual foi desenvolvido, mas sim o problema que o mesmo se propõe a resolver, ou em outras palavras, a regra de negócio.

3.3.1 Ciclo de Vida

De acordo com os preceitos do DDD para se ter um software que atenda perfeitamente a um determinado domínio, é necessário que se estabeleça, em primeiro lugar, uma Linguagem Ubíqua, de modo que o que quer que seja descrito nela, seja compreendido por todos não havendo ambiguidades.

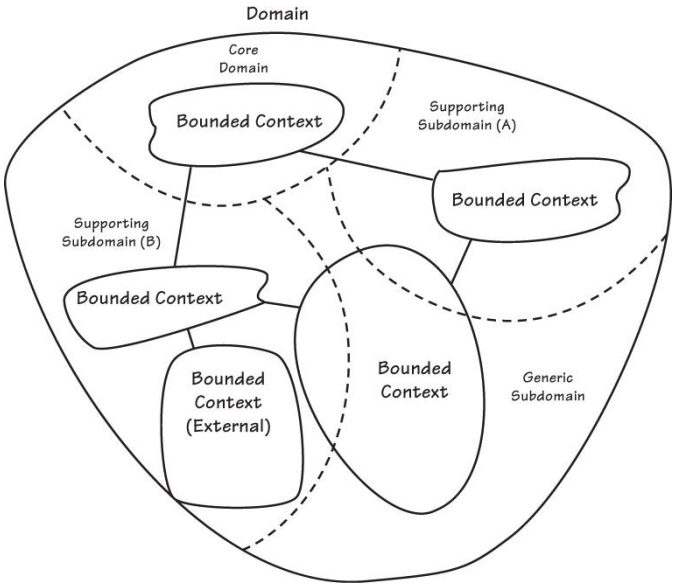
Segundo (EVANS, 2004), uma linguagem ubíqua é uma linguagem baseada no domínio e que é utilizada para representar modelos e documentações, de forma que estes sejam compreendidos pelo cliente, analista, projetista, desenhista, testador, gerente, etc. Visto que os especialistas de um determinado domínio possuem um entendimento limitado do vocabulário técnico, e um vasto vocabulário da sua própria área. Por outro lado, desenvolvedores também possuem uma linguagem própria baseada em termos descritivos e funcionais. Em meio a essa divisão linguística, especialistas descrevem vagamente o que querem, esta linguagem deve ser utilizada na comunicação entre a equipe de desenvolvimento e os especialistas no domínio, ou seja, é a linguagem utilizada no dia-a-dia por todos os envolvidos no projeto.

De acordo compara que desenvolvedores e especialistas no domínio falem a mesma língua, é preciso que ambos utilizem os mesmos termos em sua rotina diária. Para os especialistas no domínio isso é natural e já acontece, já para os desenvolvedores isso pode ser um desafio. Para que a linguagem se torne uma linguagem ubíqua é preciso utilizá-la em tudo o que é produzido: documentação, diagramas e principalmente no código.

E por ser uma linguagem do cotidiano, a maior dificuldade para seu uso é o entendimento para os termos utilizados. Equipes menos experientes tendem a impor sua realidade ao invés da realidade do domínio ou fazer traduções de termos usados em outros projetos o que pode ocasionar erros de interpretação por a linguagem utilizada não refletir a realidade no negócio.

A partir da linguagem ubíqua definida para o projeto, cria-se a modelagem estratégica de modo que se deixe claro como deve de ser a separação dos domínios do sistema em módulos menores e responsáveis por sua porção de negócio. Para documentar esta modelagem, pode fazer uso do mapa de contextos.

Figura 9 – Mapa de contextos conceitual no DDD.



A ideia é que cada contexto, ou modelo abstrato, delimitado possua sua própria Linguagem Ubíqua. Seu modelo abstrato deve ser uma representação perfeita do seu domínio, ou seja, tudo que existe no seu negócio deve aparecer no modelo e só deve aparecer no modelo aquilo que está no negócio. Este modelo abstrato deve ser feito em grupo, com todo o time presente, para que todos tenham entendimento do que é de responsabilidade de cada um no projeto. O modelo definido deve ter evolução contínua, onde o modelo servirá de guia para a criação do código e, ao mesmo tempo, o código ajuda a aperfeiçoar o modelo.

Criados a linguagem ubíqua e o modelo abstrato, o próximo passo é definir a arquitetura do sistema, cada contexto pode possuir uma arquitetura independente dos demais. O DDD não prega a necessidade de uma arquitetura específica, fica a cargo do time, definir qual solução se aplica melhor a necessidade do sistema a ser desenvolvido.

E a partir da arquitetura escolhida, estrutura-se a modelagem tática, que é uma abordagem de como escrever as classes que vão mapear os modelos do mundo real e implementar os comportamentos do negócio. Para modelar essa parte, o DDD propõe o padrão Domain Model Pattern - DMP, que objetiva isolar dos detalhes da arquitetura. Seguindo o DMP, para descrever a modelagem tática, podem ser utilizados os seguintes componentes:

- **Entidades** – São classes de objetos definidos por uma identidade. Normalmente são elementos do domínio que possuem ciclo de vida dentro da aplicação. A definição de um objeto deve ser simples e concentrada na continuidade do ciclo de vida e identidade, é necessário encontrar um meio de distinguir cada objeto,

independente da sua forma ou histórico e atentar aos requisitos que exigem combinações de objetos por atributos.

- **Objetos de Valores** – São objetos que só carregam valores, mas que não possuem distinção de identidade ao contrário das entidades. Na maioria das vezes, são imutáveis e possuem um ciclo de vida curto. Quando a preocupação de um objeto é somente com os seus atributos, esse deve ser classificado como um "objeto de valor".

- **Agregações** – São conjuntos de Entidades ou Objetos de Valores que são encapsulados numa única classe. Evans (2004) sugere que Entidades e Objetos de Valor sejam agrupados em Agregações e limites devem ser determinados para cada um. Deve ser escolhida uma entidade para ser a raiz de cada agregado e que controle o acesso aos objetos dentro do limite através da raiz.

- **Fábricas** – São classes responsáveis pelo processo de criação dos Agregados ou dos Objetos de Valores. É importante citar que o processo deve ser atômico. A responsabilidade pela criação desses complexos objetos não faz parte do modelo do domínio, mas é uma importante parte do projeto.

- **Serviços** – São classes que contém lógica de negócio, mas que não pertence a nenhuma Entidade ou Objetos de Valores. A interface deve ser definida em termos de outros elementos do modelo do domínio e A operação não deve possuir estado.

- **Repositórios** – São classes responsáveis por administrar o ciclo de vida dos outros objetos, normalmente Entidades, Objetos de Valor e Agregados. O domínio não tem que lidar com a camada de infraestrutura para referenciar objetos. Quando um objeto é criado, ele pode ser armazenado em um Repositório e utilizado depois. A diferença entre um Repositório e um acesso tradicional à Camada de dados é que nos Repositórios, os objetos são acessados em uma Coleção.

- **Módulos** – São abstrações que têm por objetivos agrupar classes por um determinado conceito do domínio, o princípio geral deste padrão está na alta coesão e no baixo acoplamento. Um baixo acoplamento entre módulos minimiza o custo da relação entre elementos em Módulos diferentes e possibilita uma análise mais clara do conteúdo de um módulo.

Tendo finalizados estes passos, o time é livre para escolher a melhor técnica de programação para o desenvolvimento do sistema, forma de apresentação e distribuição, dado que este não é mais escopo do DDD.

3.3.2 Benefícios vs Críticas

O DDD é indicado para projetos que possuem um conjunto de regras de negócio muito complexas, para um sistema simples o uso de DDD não traz benefícios que cubram seu custo de implementação. Mas pode fazer uso das técnicas para extrair um modelo factível e de valor para o cliente.

De um modo geral, iniciar um projeto usando a abordagem de DDD previne que o sistema cresça cada vez mais de uma forma não orientada ao domínio. As vantagens advêm do alcance do objetivo do DDD. O ápice de um software desenvolvido usando DDD é dar a possibilidade de delegar a correção de um bug ou mesmo o desenvolvimento de um recurso novo a uma pessoa nova no time depois de ter conversado com ela usando apenas lápis e papel, sem nem ligar o computador - ou seja, não se conversa sobre o código, conversa-se sim a respeito do negócio.

3.4 *Behaviour Driven Development* – BDD

O *behavior-driven development* ou desenvolvimento guiado por comportamento, é uma técnica de desenvolvimento ágil focada na colaboração entre desenvolvedores, testadores, analistas de negócios ou mesmo pessoas não técnicas. Faz uso de práticas do DDD como a linguagem ubíqua para especificação dos casos de uso e orientação ao domínio do negócio e da técnica do TDD com testes sendo criados antes da implementação de fato. O BDD surgiu da necessidade de colmatar algumas das falhas detectadas na utilização do TDD, algumas delas citadas anteriormente. Estas falhas são comuns a programadores pouco experientes no uso do TDD que por não compreenderem o propósito da metodologia ou por não seguirem o ciclo como proposto, não vislumbram todo o seu potencial. E foi na busca por soluções que diminuíssem a barreira de entrada de novos programadores à técnicas avançadas de desenvolvimento de software que Dan North propôs as práticas para o desenvolvimento de software orientado aos comportamentos esperados do sistema.

Para tal, ele notou que ao trocar a palavra “*test*” na nomenclatura dos seus testes criados em TDD pela palavra “*should*” seguido de uma sentença literal com o comportamento esperado do teste, traria uma modificação na interpretação do contexto ao qual o teste se aplica e também torna explícito o que o teste “deve” fazer para atender ao comportamento, tornando claro o motivo da falha.

Com esta nova forma de visualizar os testes, os desenvolvedores notam que além de ter clareza do comportamento dos testes e o motivo de sua falha, a nova nomenclatura fazia sentido como uma forma de documentação do código e escrevendo nomes de métodos na linguagem do cliente, os documentos gerados faziam mais sentido para o cliente, analistas e testadores, ampliando o pensamento para que todo

o ciclo de desenvolvimento tenha a mesma base. Por isso a importância de fazer uso da linguagem ubíqua praticada no DDD, com algumas modificações.

O Tabela 1 representa um exemplo de padrão de escrita de histórias onde claramente se pode ler a funcionalidade (*add items back to stock when they're returned*) a pessoa ou papel de quem vai fazer uso da funcionalidade *(store owner*) e o valor de negócio associado à funcionalidade (*keep track of stock*).

Tabela 1 – Descrição de um cenário em linguagem ubíqua DDD

Story: Returns go to stock	
As a	Store owner
In order to	Keep track of stock
I want to	Add items back to stock when they are returned.

Na visão de NORTH, esse tpo de descrição é simples e funcional, pois expõe o valor de entrega da história quando você a define inicialmente, porém não é suficiente para garantir que o código gerado a partir deste cenário seja correto pois não reflete o comportamento esperado dela. E como comportamento esperado, o autor cita simplesmente que devem atender aos critérios de aceitação. Então a proposta de linguagem ubíqua no BDD é a de que os cenários sejam escritos junto com seus critérios de aceitação, utilizando o modelo entradas-eventos-saídas ("*given-when-then*") com adições de operadores lógicos e-ou (*"AND-OR"*) o objetivo seria alcançado como mostram as tabelas 2 e 3.

Tabela 2 – Descrição de um cenário em linguagem ubíqua BDD.

Scenario 1: Refunded items should be returned to stock	
Given	that a customer previously bought a black sweater from me
And	I have three black sweaters in stock
When	he returns the black sweater for a refund
Then	I should have four black sweaters in stock

Tabela 3 – Descrição de um cenário em linguagem ubíqua BDD.

Scenario 2: Replaced items should be returned to stock

Scenario 2: Replaced items should be returned to stock	
Given	that a customer previously bought a blue garment from me
And	I have two blue garments in stock
And	Three black garments in stock.
When	*he returns the blue garment for a replacement in black*
Then	*I should have three blue garments in stock*
And	*Two black garments in stock.*

Tanto TDD quando BDD são formas de se pensar em testes partindo de premissas ligeiramente diferentes mas chegando ao mesmo resultado prático. Por isso, BDD não pode ser visto como uma oposição direta a TDD em termos de práticas, mas como uma corrente filosófica que expande os conceitos do TDD incluindo práticas de outras metodologias, com o objetivo de envolver pessoas além do corpo técnico do projeto. Para tornar o ambiente mais produtivo, o uso de ferramentas de ferramentas e frameworks para escrita de histórias e definição de critérios de aceitação são incentivados, tendo o próprio Dan North participado da concepção de alguns JBehave [http://jbehave.org/], RSpec [http://rspec.info/], Cucumber [https://cucumber.io/], SpecFlow [http://www.specflow.org/] entre outros.

3.4.1 Ciclo de Vida

Diz North, que BDD é uma metodologia de “fora para dentro”. Começa no exterior, onde se identifica os resultados de negócio, e posteriormente é feito uma série de exercícios para levantamento de requisitos e estudos dentro de um conjunto de recursos, o qual permite alcançar os resultados esperados.

(CHELIMSKY et al., 2010), diz que para criar um cenário utilizando os conceitos do BDD, deve-se seguir o seguinte modelo de aplicação:

- Criação de cenários: as funcionalidades são descritas em um cenário, uma por vez, mantendo o desenvolvedor focado na sua tarefa;
- Escrever a especificação para o cenário: antes do desenvolvimento da funcionalidade deve ser escrito a especificação em uma linguagem comum como o que foi apontado pelo cliente;
- Executa o script e assegurar a falha (vermelho);
- Escrever uma especificação de unidade: essa parte é a implementação do código, para executar com sucesso a especificação do usuário. Mas a especificação do usuário está relacionada apenas a parte externa do software não dizendo nada

com os artefatos do código- fonte. Assim o desenvolvedor deve reescrever essa especificação fazendo pequenas alterações para que ela passe;

- Executar novamente o script de teste para garantir que passe (verde);
- Refatorar o código, pois foi implementado apenas o mínimo necessário para fazer a especificação passar. A refatoração reescreverá o código com a intenção de melhorar a sua estrutura, mas sem afetar seu comportamento (azul).

Nota-se claramente a influência do TDD no ciclo proposto, pois deve-se escrever um teste (comportamento) e notar sua falha, criar o código que satisfaça o comportamento esperado e vê-lo obter sucesso e então refatorar o código para que este seja claro e legível, melhorando sua estrutura.

3.4.2 Benefícios vs Críticas

O grande diferencial no BDD para quem o usa é a integração de regras de negócio com tecnologia, onde se torna claro que é papel do negócio representar, em alto nível, as principais características do sistema que corresponde a uma descrição resumida dos valores envolvidos, descrever os casos de uso, com pré-requisitos, ações e resultados esperados, e também, uma interação entre agente externo (usuários do sistema) e resultado esperado para um dado cenário.

Assim o BDD se concentra naquilo que é mais importante e de grande valor para o cliente, assegurando a qualidade do projeto por meio de códigos limpos, funcionais e sem ambiguidades já que o especialista de negócio é uma ponta ativa da equipe de desenvolvimento. A comunicação com o cliente também tem uma melhora por trazê-lo para formalização do projeto e dando ciência a ele de todo progresso, por meio da linguagem comum adotada.

Como limitações, pode-se citar a falta de profissionais experientes para fazer uso pleno da técnica visto que ela é relativamente nova no mercado. Dessa forma, pode ser necessário um investimento em capacitação da equipe e em ferramentas. O grau de envolvimento do cliente no projeto também pode ser um fato complicador, pois este pode não estar disponível em datas-chave para o andamento do projeto.

4 Testes Ágeis vs Testes Tradicionais

O planejamento e execução das tarefas de testes, sejam eles ágeis ou tradicionais, provém da metodologia escolhida para o gerenciamento do projeto em questão, desta forma, as características de gerenciamento do projeto, são refletidas em todas as atividades até sua finalização. Assim, podemos sumarizar todas as características apontadas nos capitulos anteriores, em tres itens que estão descritos nas seções a seguir.

4.1 Práticas de planejamento de projetos

As práticas adotadas no planejamento de projetos que usam metodologias tradicionais, diferem dos projetos ágeis por desprenderem tempo demasiado em atividades que não são convertidas em código (que devem ter mais valor no ponto de vista do cliente), ou seja, as atividades de documentação e análise, não sustentam o projeto como as fases de codificação e testes.

Na Tabela 4 a seguir pode-se perceber a rigidez imposta pelas metodologias tradicionais em face a flexibilidade proporcionada pelos projetos gerenciados na metodologia ágil.

Tabela 4 – Comparativo entre planejamentos de projetos gerenciados com as metodologias Tradicional e Ágil

	Modelo Tradicional	Modelo Ágil
Planejamento	Pensado e detalhado e acordado no início do projeto, com cronograma e recursos especificados para todo o ciclo.	Planejamento macro realizado no início do projeto, com definições específicas realizadas a cada iteração.
Escopo	Amplamente discutido e documentado no planejamento do projeto. Qualquer alteração está sujeito a um novo planejamento.	O escopo geral é resumido e aprovado no início do projeto, todos os detalhes das funcionalidades são solicitados ao cliente conforme o avanço e entendimento do projeto.
Priorização	São definidas, detalhadas e acordadas ainda na fase de planejamento do projeto, qualquer mudança após essa fase, pode levar a alteração de escopo e um novo planejamento para o restante do projeto.	Priorização inicial é realizada junto com o planejamento do projeto, mas com liberdade para repriorizar itens durante o andamento do projeto, caso julgue necessário.
Documentação	Formal, detalhadas e extensivas, para cada etapa do projeto. São documentadas e assinadas pelas partes antes do início do desenvolvimento.	Documentação mínima exigida a cada iteração do projeto, conforme o andamento. Escritas em linguagem natural para facilitar o entendimento por todas as partes do time e facilitar o controle por frameworks de automação.

4.2 Execução do projeto

Para as práticas de execução de projetos, os modelos de desenvolvimento e níveis de testes utilizados pelas metodologias tradicionais (Cascata e o Modelo V citados na seção 2.4 deste trabalho) são rígidas, não receptivas à mudanças nos requisitos especificados no início do projeto e necessitam de uma grande carga de documentação para serem efetivos no que se propõem, o tempo destinado a tarefas de teste pode ser influenciado por demandas de codificação prejudicando assim a qualidade e/ou a quantidade de itens testados.

Já nas metodologias tradicionais, o esforço de trabalho é dividido entre os

integrantes do time, onde todos são responsáveis por uma porção da qualidade do software final. Como os itens de trabalho só são considerados finalizados após cada um ser testado, todos os itens obrigatoriamente tem o mesmo tempo para carga de teste ser executada as mudanças são bem vidas e a necessidade documentação abrangente é mitigada por testes assertivos e de qualidade (SCRUM.AS, 2013).

A Tabela 5 busca resumir as características da execução e desenvolvimento de projetos nas duas metodologias onde as metodologias ágeis buscam focar na melhoria de componentes e iteração com o cliente, para que este tenha o perfeito entendimento do que foi pedido.

Tabela 5 – Comparativo entre a execução de projetos usanto metodologias Tradicional e Ágil

	Metodologia Tradicional	Metodologia Ágil
Arquitetura	Definida para todo o projeto com foco em reuso de componentes. Utilizando modelos de desenvolvimento cascata, espiral ou V.	Simples, interativa e evolutiva, normalmente baseada em refatoração de componentes com modelos TDD, ATDD ou BDD.
Execução	Deve seguir conforme o planejamento e qualquer mudança deve ser avaliada, acordada e possivelmente replanejada.	Feita em iterações (quinzenais ou mensais) com *checklists* das metas nos intervalos, onde as mudanças são avaliadas e planejadas para as próximas iterações.
Correções/Lições	Correções são realizadas no final de cada fase, conforme liberdade de cronograma. Lições aprendidas são documentadas e detalhadas para auditorias posteriores.	Correções e desvios de produtividade são realizados na iteração seguinte a sua descoberta, lições aprendidas, são documentadas e consultadas a cada planejamento.

4.3 Relações entre os envolvidos no projeto

Um outro ponto de disrupção entre as metodologias são os papéis dos personagens para que um projeto funcione, são eles: os especialistas de testes, desenvolvedores e pessoal envolvido com o negócio. A diferença está na carga de trabalho exercida por cada um destes ao longo do ciclo de desenvolvimento do *software*.

Em projetos da metodologia tradicional, os especialistas de negócio tem uma grande carga de trabalho no início do projeto quando são levantados e especificados

os requisitos, especificações funcionais e não funcionais, casos de uso e demais documentações, os desenvolvedores tem demandas a tratar apenas na fase de codificação e podem ou não testar unitariamente seu código, analistas de testes só entram em ação no fim do projeto. Ou seja, há muito tempo ocioso para manter profissionais em um projeto, por vezes com custos não convertidos em esforço de trabalho muito por conta dos papéis bem definidos e a pouca autonomia para inovar no processo de trabalho.

Já nos projetos executados utilizando metodologias ágeis, todos os personagens trabalham colaborativamente durante todo o ciclo de desenvolvimento isso por conta do princípio ágil que estabelece que as equipes devem ser autonomas e trabalhar em interações curtas com entregas pequenas porém de grande valor. Para definir o grande valor a ser entregue os especialistas de negócio devem levantar requisitos no início do projeto, porém a cada iteração devem se reunir com o representante do cliente e o time para priorizar os itens a serem desenvolvidos naquela iteração, levantar casos de testes para aquela iteração e cobrar do cliente os insumos necessários bem como tratar para que a equipe não sofra interferências que atrapalhem o andamento do projeto. Como todos tem participação durante todo o projeto, os custos são diluídos de maneira igual assim como os esforços de trabalho e as contribuições para o valor do projeto.

A Tabela 6 mostra as características da relação entre as pessoas e recursos interessados no projeto nas duas metodologias e a comunicação entre estes.

Tabela 6 – Comparativo entre as relações dos envolvidos no projeto

	Metodologia Tradicional	Metodologia Ágil
Equipe	Papéis rigidos e bem definidos, com pouca autonomia para o time inovar em práticas e processos.	Multifuncional, autonoma, independente e autoregulada possui poder para tomar decisões no projeto e interagir com o cliente, caso julgue necessário.
Cliente	Envolvido nas etapas de planejamento e validação dos resultados.	Sempre envolvido em atividades ao longo do projeto, com a possibilidade de ter um representante dentro da equipe do projeto.
Comunicação	Formal e documentada com relatórios frequentes e detalhados, via email, atas de reunião ou contratos.	Informal (preferencialmente verbal) aberta e direta com informações visíveis para todos os envolvidos no projeto.

5 Considerações Finais

Este trabalho buscou apresentar uma comparação entre as diversas formas de assegurar a qualidade do *software* durante seu ciclo de desenvolvimento, apresentando diversas técnicas e abordagens dentro das metodologias de teste tradicional e ágil.

Ficou claro que no diagrama da abordagem tradicional os esforços de teste acontecem no final da codificação e antes da publicação. O diagrama é idealista, porque dá a impressão de que há tanto tempo para o teste como existe para a codificação, o que em muitos projetos, não é a realidade. O teste pode ser prejudicado porque a codificação leva mais tempo do que o esperado, e porque as equipes entregam um projeto pronto somente no final. É comum no desenvolvimento tradicional, os testadores receberem uma versão com todas as alterações do projeto e a partir de aí começarem suas verificações.

Por outro lado, o Ágil é iterativo e incremental, portanto os testadores podem testar cada incremento de codificação, assim que a funcionalidade é publicada. O time analisa, desenvolve e testa uma funcionalidade ou parte dela apenas, e segue fazendo isso até que a solicitação do cliente seja atendida. O time deve terminar a codificação e testes no mesmo momento, porque uma funcionalidade não é “pronta” até que tenha sido testada. Assim, principal diferença entre as duas abordagem está na diferença na definição de “item pronto”, na abordagem tradicional o produto fica “pronto” para então ser testado, já nas abordagens ágeis, a funcionalidade só está “pronta” após os testes.

Cada uma delas tem seus benefícios e pontos de melhoria, porém isoladamente, não conseguem assegurar a ausência de falhas em um projeto, dado que é impossível validar todos os comportamentos possíveis no fluxo de um algoritmo complexo, como os que compõem um sistema, de qualquer porte que este seja.

Na visão do autor ter as duas metodologias sendo executadas no mesmo projeto, obviamente, cada uma trabalhando em seu escopo, trará maior segurança no produto em que se pretende lançar no mercado, pois haverá a garantia da correta especificação por parte do cliente e o entendimento destas especificações por todos os componentes do time (utilizando ATDD ou BDD), também será garantido a correta implementação do que foi pedido pelo cliente (novamente utilizando ATDD, BDD ou mesmo TDD) e fazendo uso das verificações funcionais das metodologias tradicionais, serão assegurados que o sistema funciona a contento (testes de carga/performance, usabilidade, etc.).

Como trabalho futuro, pretende aprofundar-se no estudo de aplicações reais no mercado com uso de metodologias de teste ágil para conhecer o comportamento destes durante e após seu lançamento, validar se o retorno do investimento foi como

esperado e se a expectativa dos *stakeholders* foram alcançadas.

Referências

ABRAHAMSSON, P. et al. New Directions on Agile Methods: A Comparative Analysis. *Proceedings of ICSE'03*, p. 244 – 254, 2003.

ASTELS, D. *Test-Driven Development: A Practical Guide*. 2. ed. [S.l.]: Prentice Hall, 2003.

BACH, J. Risk and Requirements-Based Testing. *IEEE Computer Society*, 1999. Disponível em: <http://www.satisfice.com/articles/requirements_based_testing.pdf>.

BECK, K. *Test-Driven Development: By Example*. [S.l.]: Addison-wesley Professional, 2002.

BECK, K. et al. *Agile Manifesto*. 2001. Disponível em: <http://agilemanifesto.org/iso/ptbr/manifesto.html>. Acesso em: 01/12/2016.

BEIZER, B. *Software Testing Techniques*. [S.l.]: International Thomson Computer Press, 1990.

BOEHM, B. W. *A Spiral Model of Software Development and Enhancement*. [S.l.]: Northrop Grumman, 1988.

BOURQUE, P.; FAIRLEY, R. *Software Engineering Body of Knowledge*. [S.l.]: IEEE Computer Society, 2014.

BRANDÃO, H. A. et al. *xUnit – Testes Unitários Automatizados*. [S.l.], 2005. Disponível em: <http://paginas.fe.up.pt/~aaguiar/es/artigosfinais/es_final_6.pdf>.

CAETANO, C. *Testes Ágeis*. 2016. Disponível em: <http://www.qualister.com.br/blog/testes-ageis>. Acesso em: 09/12/2016.

CHELIMSKY, D. et al. *The RSpec Book (Pragmatic Bookshelf)*. [S.l.]: O'Really, 2010.

CLAUDIO, A. *Introdução a Teste de Software*. 2015. Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: 01/12/2016.

COHEN, D.; LINDVALL, M.; COSTA, P. *An introduction to agile methods. In Advances in Computers*. [S.l.]: Elsevier Science, 2004.

CRAIG, R.; JASKIEL, S. P. *Systematic Software Testing*. [S.l.]: Artech House Publishers, 2002.

CRISPIN, L.; GREGORY, J. *Agile Testing – A practical guid for Testers and Agile Teams*. [S.l.]: Addison-Wesley Professional, 2009.

DEURSEN, A. V. Program Comprehension Risks and Opportunities in eXtreme Programming. *Working Conference on Reverse Engineering*, 2001.

DRUCKER, P. *Administrando em Tempos de Grandes Mudanças*. [S.l.]: Pioneira, 1999.

EVANS, E. *Domain Driven Design – Tackling Complexity in the Heart of Software*. 2. ed. [S.l.]: ALTA BOOKS, 2004.

HENDRICKSON, E. *Agile Testing: Nine Principles and Six Concrete Practices for Testing on Agile Teams*. Quality Tree Software, Inc., 2008. Disponível em: <http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf>.

HUIZINGA, D.; KOLAWA, A. *Automated Defect Prevention: Best Practices in Software Management*. [S.l.]: IEEE Computer Society Press, 2007.

IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. [S.l.]: IEEE, 1990.

KOCH, A. S. *Agile Software Development - Evaluating the Methods for Your Organization*. [S.l.]: Artech House, 2005.

LIVERMORE, J. A. Factors that impact implementing an agile software development methodology. *SOUTHEASTCON. Proceedings*, p. 82 – 86, 2007.

MASSOL, V. *JUnit in Action*. 2. ed. [S.l.]: Manning Publications, 2003.

MOREIRA, G.; PRADO, G. Domain-Driven Design: Além dos Patterns. *MundoJ*, 2010.

MYERS, J. G. *The Art of Software Testing*. 2. ed. [S.l.]: John Wiley & Sons, Inc., 2004.

PATTON, R. *Software Testing*. 2. ed. [S.l.]: Sams Publishing, 2005.

ROCHA, A.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de Software: Teoria e Prática*. [S.l.]: Prentice Hall, 2001.

ROYCE, W. W. Managing the Development of Large Software Systems. *Proceedings of IEEE*, p. 1 – 9, Agosto 1970.

SCRUM.AS. Fundamental Agile Testing Principles, Practices, and Processes. In: *Academy - International Agile Tester Foundation*. [s.n.], 2013. Disponível em: <https://www.scrum.as/academy.php?show=2&chapter=4#sthash.0wRnPlUU.dpuf>.

TASSEY, G. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. [S.l.]: National Institute of Standards and Technology, 2002.

The Standish Group International. CHAOS Report. In: . [s.n.], 2015. Disponível em: <https://www.projectsmart.co.uk/white-papers/chaos-report.pdf>.

WASMUS, H.; GROSS, H. *Evaluation of Test-Driven Development*. [S.l.]: Delft University of Technology., 2007.