UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE EDUCAÇÃO CONTINUADA
ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Alisson Marcelo

# ATDD E DOCUMENTAÇÃO DE USUÁRIO COMO FERRAMENTAS ÁGEIS DE DEFINIÇÃO E VALIDAÇÃO DE REQUISITOS

São Leopoldo
2016

UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE EDUCAÇÃO CONTINUADA
ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Alisson Marcelo

# ATDD E DOCUMENTAÇÃO DE USUÁRIO COMO FERRAMENTAS ÁGEIS DE DEFINIÇÃO E VALIDAÇÃO DE REQUISITOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Qualidade de Software, pelo curso de Pós-Graduação Lato Sensu em Qualidade de Software da Universidade do Vale do Rio dos Sinos - UNISINOS.
Orientador: Prof. MSc. Guilherme Silva de Lacerda

São Leopoldo
2016

# ATDD e documentação de usuário como ferramentas ágeis de definição e validação de requisitos

**Alisson Marcelo**

[1]Universidade do Vale do Rio dos Sinos - Unisinos
Av. Unisinos, 950 - Cristo Rei, São Leopoldo - RS, 93022-000

alisson.marcelo@gmail.com, guilhermeslacerda@gmail.com

***Abstract.*** *Agile development environments frequently use direct communication with customers in order to refine requirements. This approach works fine for software products developed for a single customer. Some problems emerge when software is part of a product, as in telecommunication devices and smartphones. The different device models can have functionalities performed by hardware or software part. Depending on the target market of these products and requiring the same tests to be performed on many devices. In this scenario with software as part of a product, it grows the need for agile requirements definition in the first product containing a functionality. The next products should have the same requirements validated. The use of Acceptance Test Driven Development (ATDD) creates an executable version of the requirements document validation and an agile definition of the tests that should be executed to validate them. This work presents the increase of understanding of requirements when we use the customer documentation in conjunction with ATDD to improve the interaction between user, customers, testers and developers.*

***Resumo.*** *Ambientes de desenvolvimento ágil tendem a definir os requisitos através de comunicação direta com os clientes. Isto funciona bem para um produto de software único desenvolvido para um cliente específico. Problemas surgem em cenários onde o software é parte de diferentes produtos de hardware(HW) da empresa, como equipamentos de telecomunicações e smartphones. Cada modelo pode ter diferentes funcionalidades nativas em HW, dependendo do foco do produto, devendo o mesmo requisito ser atendido e testado em diferentes modelos de equipamentos. Neste cenário de software como parte de diferentes produtos, cresce a necessidade de uma definição ágil dos requisitos no primeiro produto e da sua validação nos demais. O Acceptance Test Driven Development (ATDD) usa os testes como ferramenta de análise de requisitos. Neste trabalho é apresentado o uso da documentação de usuário como artefato de entrada para o desenvolvimento do software, ampliando a capacidade de comunicação do ATDD entre os envolvidos no processo.*

## 1. Introdução

O desenvolvimento ágil tem sido adotado em diversas áreas do desenvolvimento de software como solução para alguns problemas do desenvolvimento tradicional, principalmente a sua baixa capacidade de adaptação do escopo a mudanças de requisitos durante o projeto [Lucia e Qusef 2010]. Apesar de ter diversos casos de sucesso, os métodos ágeis ainda possuem alguns desafios que devem ser tratados para que eles possam ser melhor utilizados em projetos maiores e de longa duração, principalmente no

rastreamento de requisitos e na pouca quantidade de documentação gerada pelo projeto [Cao e Ramesh 2008]. Como os métodos ágeis privilegiam comunicação direta ao invés de documentação extensa [Beck et al. 2001], este trabalho propõe o uso da documentação de usuário e os testes sejam utilizados não apenas como ferramentas de validação e registro do que foi implementado, mas como documentos de especificação de requisitos e comunicação com os usuários.

O objetivo deste trabalho é demonstrar a validade do uso da documentação de usuário descrevendo os cenários de aplicação dos produtos como entrada para o desenvolvimento orientado a testes de aceitação (ATDD) ser usado como ferramenta ágil de definição e validação de requisitos. Este trabalho avalia os benefícios de antecipar no processo de desenvolvimento não apenas os testes mas também a documentação de usuário, visando aumentar o entendimento dos desenvolvedores a respeito do cenpario de uso dos clientes. Estes cenários de aplicação são especialmente úteis no software de produtos que dependam da interconexão com outros ou nos casos de produtos para os quais apenas um bom projeto de interface não seja suficiente para garantir que o usuário consiga intuir todas as formas de utilizar o produto.

Para validar os conceitos e propostas feitas neste documento, foi conduzida uma pesquisa-ação [Baskerville 1999] focada no desenvolvimento de equipamentos de acesso a redes de computadores, como *switches* e roteadores. Este estudo é particularmente interessante por se tratar de um projeto grande e de longa duração, com múltiplas versões e plataformas a serem suportadas ao longo do tempo, o que evidencia a eficiência da abordagem proposta em casos nos quais o desenvolvimento ágil ainda tem críticos mais fortes devido a problemas de documentação e escalabilidade.

A ideia central da proposta é adiantar o momento no qual documentos de usuário que seriam construídos de forma incremental ao final de cada entrega sejam feitos antes delas, servindo como descrição do cenário de aplicação do software a ser desenvolvido. Estes cenários de aplicação são definidos com os setores comerciais da empresa, e registrados em documentos que futuramente serão entregues para os clientes como guias rápidos ou *application notes*, dependendo do caso.

Esta proposta além da questão de requisitos também resolve o problema do descompasso entre as entregas realizadas no software e a documentação disponibilizada aos usuários que normalmente ocorre e que acaba por retardar o seu lançamento. Como a elaboração desta documentação já é prevista em muitos projetos, não há um aumento do esforço em documentação, apenas a antecipação do momento em que ele é realizado.

Utilizando o ATDD como metodologia de desenvolvimento, esta documentação dos cenários a atender é utilizada pelos desenvolvedores e testadores para definir o que pode ser testado automaticamente e o que deve ser feito de forma manual ou exploratória. Aquilo que pode ser automatizado se torna uma especificação dinâmica de requisitos que ao final do desenvolvimento dos testes e do código serve para validar e rastrear os requisitos atendidos de forma automática.

Esse desenvolvimento da documentação é realizado em um momento no início da iteração de modo a não gerar documentação de algo que não será imediatamente implementado, preservando o atendimento aos princípios ágeis. Esse objetivo aumenta o desafio de sempre ter o software em condições de fechamento de versão a cada nova

entrega.

Ao final deste trabalho espera-se responder: qual o impacto do uso da documentação de usuário como entrada de requisitos para o ATDD no desenvolvimento de software embarcado? A análise foi realizada em termos de quantidade de defeitos, consenso sobre o que deve ser implementado e cobertura dos requisitos de usuário.

O restante deste documento é organizado como segue: na próxima seção será fornecida referência teórica para o assunto. Na seção 3 são apresentados trabalhos relacionados e discutidas as diferenças entre eles e este trabalho. Na seção 4 é discutida a proposta da metodologia, o estudo de caso, analisados os resultados e discutidos as conclusões a respeito deles. A última seção encerra o trabalho resumindo as conclusões obtidas ao longo dele, resumindo a sua contribuição científica e apontando caminhos a seguir na condução de trabalhos futuros.

## 2. Fundamentação Teórica

### 2.1. Impacto das Falhas nos Requisitos

A interação entre a área de análise de requisitos e o desenvolvimento ágil tem sido o foco de diversos trabalhos [Lucia e Qusef 2010] [Paetsch et al. 2003] [Ricca et al. 2009], pois falhas nos requisitos ocasionam diversos problemas no ciclo de vida do software, dentre eles o não atendimento das necessidades dos usuários e um alto custo de manutenção como pode ser visto na figura 1 (Fonte: IBM Systems Sciences Institute, 2004).

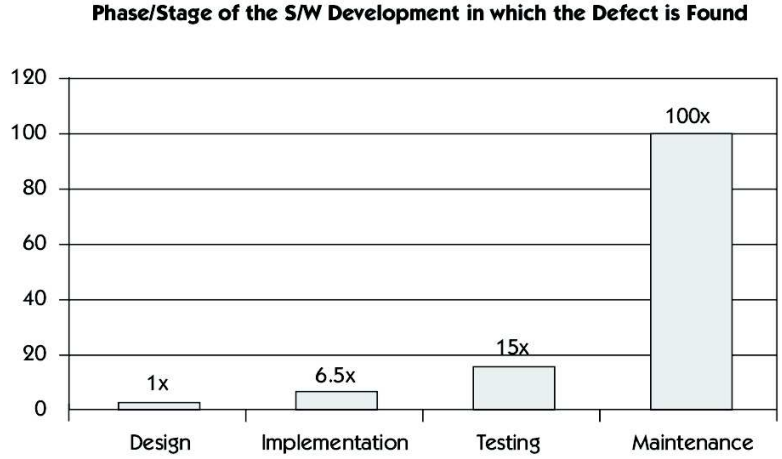


**Figura 1. Custo de correção de defeitos em SW.**

Como falhas no entendimento das necessidades dos usuários são defeitos gerados na fase de análise ou projeto, eles geram custo de correção/alteração como as falhas no desenvolvimento. Com os métodos ágeis o custo de alterações no software durante o desenvolvimento do projeto é atenuado, mas na fase de manutenção tendem a ser muito semelhantes, pois envolvem os mesmos custos de deslocamento, parada de serviços, pessoal de suporte técnico e de imagem da empresa.

Estes custos se referem a uma alteração ou correção isolada e não ao volume total de recursos aplicados em cada fase. Os métodos ágeis [Beck et al. 2001] tem a intenção de reduzir o custo de correção na fase de manutenção através do aumento da comunicação

entre desenvolvedores e clientes e no constante *feedback* destes. Reduzindo as falhas de entendimento, a implementação entregue ao cliente tende a atender melhor as suas necessidades, inclusive aquelas que forem identificadas no decorrer do desenvolvimento do projeto e não apenas no seu início.

## 2.2. Metodologias Ágeis de Desenvovimento

Sempre que se busca adicionar novas práticas ao desenvolvimento ágil é importante revisar o manifesto ágil [Beck et al. 2001] para garantir que os valores contidos nele sejam preservados. O manifesto ágil valoriza:

| |
|---|
| Indivíduos e interações 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 |

A partir destes valores foram definidos os 12 princípios por trás do manifesto ágil [Beck et al. 2001]:

1. Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado.
2. Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
3. Entregar frequentemente software funcionando, em intervalos de poucas semanas a poucos meses, com preferência à menor escala de tempo.
4. Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto durante todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho.
6. Informações para e entre uma equipe de desenvolvimento é através de conversa face a face.
7. Software funcionando é a medida primária de progresso.
8. Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
9. Contínua atenção à excelência técnica e bom design aumenta a agilidade.
10. Simplicidade, a arte de maximizar a quantidade de trabalho não realizado, é essencial.
11. As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

## 2.3. ATDD

O ATDD - *Acceptance Test Driven Development* ou Desenvolvimento Orientado a Testes de Aceitação é uma prática na qual todo o time de desenvolvimento discute com os usuários, *stakeholders* e testadores, os critérios de aceitação, com exemplos, traduzindo-os em

um conjunto concreto de testes de aceitação acordado por todos antes do início do desenvolvimento. Estes testes funcionam como uma especificação executável. Eles são gerados em sessões de criação do *backlog* do produto, com a participação da equipe, *Product Owner*, além dos demais interessados. Esta sincronização de expectativas garante que o resultado final esperado está claro e equivalente para todos, pois a definição de pronto é (*Definition of Done* [1] ou DoD) compartilhada por todos [Hendrickson 2008a]. O artigo de Elisabeth Hendrickson [Hendrickson 2008a] é comumente citado como a referência mais conhecida sobre ATDD [Nogueira 2016] [Rebelo 2014] [Gregory 2010], embora alguns trabalhos anteriores [Andersson et al. 2003] citem [Beck 2002] como a primeira menção ao termo.

O ATDD visa criar testes automatizados antes mesmo de criar o código que representam as expectativas de comportamento que o software deveria ter. Ao invés de criar testes focados em código com a perspectiva do desenvolvedor (que é o foco do TDD), o ATDD prega que sejam desenvolvidos testes de aceitação com uma visão direta de negócio voltado ao ponto de vista do usuário.

Os testes de aceitação definem quais as saídas ou interações com outros sistemas que um software deve ter a partir de um determinado conjunto de entradas, sem se importar com os detalhes de implementação que levam a estes comportamentos.

O desenvolvimento orientado a testes no nível de aplicação pelos usuários foi brevemente comentado por [Beck 2002], sendo apresentado como um problema social a ser resolvido, pois daria uma nova responsabilidade aos usuários, o que poderia exigir treinamento e adicionar custos.

Visando minimizar os impactos para o usuário, alguns autores iniciaram pesquisas com o objetivo de orientar o desenvolvimento pelos testes de aceitação, utilizando-os como ferramenta de comunicação, definição e validação de requisitos [Hendrickson 2008b] [Watt e Leigh-Fellows 2004].

Antes de iniciar a discussão a respeito do ATDD é necessário posicioná-lo dentre alguns dos tipos de testes existentes. Durante o desenvolvimento do software, os tipos mais comumente empregados [Watt e Leigh-Fellows 2004] são de unidade, funcional e de acetação, os quais são resumidamente explicados abaixo:

- **de unidade**: testam pequenas unidades do software, como métodos, funções e procedimentos. Eles são especialmente importantes para garantir a integridade do software após alterações e refatorações de código, sendo muito utilizados em TDD. Os testes unitários e de aceitação são complementares, não sendo recomendado que um deles seja suprimidos após a adoção do outro.
- **funcional**: ainda no nível de testes de desenvolvimento, os testes funcionais são utilizados para garantir que os componentes do software interoperam corretamente. Os testes funcionais são normalmente baseados em algum cenário de aplicação cenários de
- **de aceitação**: os testes de aceitação são a base deste trabalho e são detalhados ao longo dele. Resumidamente pode-se dizer que eles especificam o critério de aceitação de uma estória. Eles exigem participação ativa do usuário, tanto na sua

[1] Definição de Pronto, utilizada em métodos ágeis para descrever os itens a serem atendidos para que uma determinada atividade seja considerada pronta.

discussão quanto na escrita de testes. O desafio principal no desenvolvimento dos testes de aceitação é garantir que os requisitos estejam totalmente claros no início da iteração, evitando erros no desenvolvimento devido a falhas no entendimento dos requisitos.

Diferente do TDD comumente usado em *Extreme Programming* [Beck 2000] e que é orientado a garantir a qualidade interna do software, o ATDD é focado em garantir a qualidade externa, ou seja, aquela que é diretamente perceptível pelo usuário. Estas técnicas são complementares, pois garantem a qualidade sob perspectivas diferentes. Os detalhes a respeito do TDD não são cobertos neste trabalho.

Assim como no TDD, o ATDD é uma prática de programação e não apenas de teste, pois o principal foco é o alinhamento de expectativas antecipadamente para guiar o desenvolvimento do software.

Após a discussão a respeito dos requisitos, eles são convertidos em testes de aceitação, em linguagem próxima da natural, pois facilita o entendimento por parte dos clientes e *stakeholders*. Esta discussão é realizada para cada *user story* no *backlog*. São exemplos de ferramentas que podem ser utilizadas no ATTD: FitNesse [2], Robot Framework [3], Concordion [4], Cucumber [5], JBehave [6].

Como os testes são desenvolvidos em paralelo com o software a ser testado, eles servem como indicadores de quanto falta ser implementado para que o software atinja o *status* de pronto.

Como mencionado em [Rebelo 2014], enquanto Craig Larman e Bas Vodde (2010a) , em seu artigo, consideram a aplicação do ATDD como um fluxo de 3 etapas: debater, desenvolver e revisar; Hendrickson incluiu mais um passo: refinar. O ciclo de ATDD proposto por Hendrickson (2008b) é mostrado na figura 2 [Rebelo 2014].

A seguir são detalhadas as etapas da figura 2 :

- Debater (*Discuss*): Nesta etapa se discutem os requisitos e alinham-se as expectativas. As estórias de usuário (*user stories*) são refinadas em um *workshop* ou em uma reunião de preparação do *backlog* do produto (*backlog refinement*), antes da reunião de planejamento da iteração/*sprint*. Em ambos os casos, os participantes são uma equipe multifuncional, o *Product Owner* e, algum outro interessado que potencialmente tem mais informações sobre as estórias.
- Refinar (*Distill*): Nesta etapa os requisitos são refinados e convertidos para a linguagem suportada pelo *framework* de teste.
- Desenvolver (*Develop*): Neste ponto é realizado o desenvolvimento propriamente dito, o qual pode utilizar TDD para garantir a qualidade interna.
- Demonstração (*Demo*): Na etapa final desta iteração do desenvolvimento da estória, o software funcionando é mostrado para os clientes, que opinam sobre o resultado final, levantando alterações e garantindo que o software atende completamente as suas expectativas.

[2] http://www.fitnesse.org
[3] http://robotframework.org
[4] http://concordion.org
[5] http://cucumber.io
[6] http://jbehave.org


**Figura 2. Ciclo de desenvolvimento ATDD.**

Este trabalho foca no detalhamento das etapas de Discussão e Refinamento, as quais estão diretamente ligadas ao levantamento de requisitos e definição dos testes. Apesar de não serem tratadas neste trabalho, as etapas de desenvolvimento e demonstração são muito importantes para o ATDD. Como em outros métodos ágeis elas garantem que o que está sendo entregue atende aos requisitos de qualidade do produto e maximizam a sua aderência às necessidades dos clientes.

### 2.3.1. Vantagens do ATDD

Hendrickson (2008) afirma que o ato de definir os testes de aceitação no início de um ciclo de desenvolvimento aumenta consideravelmente o entendimento das necessidades dos usuários. O processo força todos os envolvidos a atingirem um consenso a respeito do que o software deve exibir. Times que seguem todo o processo, automatizam os seus testes e com o conhecimento adquirido desenvolvem módulo de software que são mais testáveis, facilitando assim a adição de novos testes no futuro. Como vantagens adicionais, ela ainda cita o uso dos testes para a garantia de não regressão, *feedback* rápido a respeito de falhas de implementação e de entendimento dos requisitos.

Markus Gärtner (2012) , descreve como vantagem do ATTD o seguinte:

> *Um analista de qualidade e um programador que colaboram juntos para atingir a meta da equipe, em relação a automação de teste, agregam muito valor quando iniciam a abordagem do ATDD [Gartner 2012].*

Rebelo (2014) adiciona como vantagem do ATDD a obtenção de um entendimento mais claro e refinado dos requisitos, possibilitando um acordo entre ambas as partes do que será desenvolvido durante uma iteração/*sprint*. No final, o resultado estará alinhado às expectativas do cliente.

Por aumentar a comunicação entre clientes e desenvolvedores, o ATDD minimiza

os equívocos no entendimento dos requisitos evitando grandes retrabalhos no futuro.

A forma padrão de levantamento dos requisitos em metodologias ágeis é a discussão entre o *product owner* e os desenvolvedores [NetObjectives 2013]. A principal forma de esclarecimento é a resposta das dúvidas dos desenvolvedores pelo *product owner*. O problema que decorre disto é que nem sempre temos total clareza daquilo que sabemos e do que sabemos que não sabemos e deveria ser perguntado. Se ao discutir os requisitos, os envolvidos não conseguem identificar que há falhas no entendimento, toda a implementação, testes e demonstrações futuras desta funcionalidade são afetadas e em muitos casos identificadas apenas após o produto de software ser colocado em produção. A proposta do ATDD é através do aumento da comunicação entre os envolvidos e do maior envolvimento dos usuários nos testes aumentar a quantidade de perguntas feitas para reduzir a quantidade de falhas no entendimento das expectativas dos usuários. A perspectiva da discussão muda de saber o que é necessário implementar para qual o comportamento o software deve ter para atender as necessidades do usuário, em termos de entradas e saídas no formato utilizado no domínio da aplicação. A diferença entre as perguntas define a perspectiva pela qual são discutidos os requisitos e o comprometimento do usuário e dos *stakeholders* com a qualidade do que será entregue.

## 2.4. Aplicações do ATDD

Uma motivação para focar o desenvolvimento em testes de aceitação é buscar a resposta para a questão: "Os desenvolvedores sabem quando o software está pronto?"[Watt e Leigh-Fellows 2004]. A conclusão que ele chega é que a pergunta é simples, mas nem sempre a resposta é óbvia. Os testes de aceitação são meios eficientes de expressar os requisitos em uma forma que não gera ambiguidades.

O desafio principal é desenvolver e manter estes testes. A forma mais efetiva encontrada por eles foi tornar os testes não apenas da definição das funcionalidades, mas de todo o processo de desenvolvimento. Há dois pontos importantes a respeito dos testes de aceitação: importância e dificuldade [Watt e Leigh-Fellows 2004].

A importância dos testes de aceitação se dá pelo fato de que as estórias descrevem o requisito sucintamente e os testes descrevem detalhadamente qual deve ser o comportamento do software para atender a este requisito, definindo claramente o ponto no qual a implementação entregue pode ser considerada pronta. Os testes servem como um contrato entre clientes e desenvolvedores. Nas discussões a respeito dos testes são definidos comportamentos do software, os quais implicam em mais ou menos esforço.

Por outro lado, a definição e desenvolvimento de testes de aceitação são atividades de difícil adoção. O primeiro ponto é a visão de que os testes de aceitação tem como objetivo apenas testar e garantir a qualidade após a implementação. No ATDD, os testes são uma ferramenta de descrição de requisitos e não apenas uma outra técnica de desenvolvimento de testes caixa preta. Os custos maiores dos testes de aceitação, se comparados aos testes unitários que são desenvolvidos e mantidos pelos desenvolvedores do software, se devem ao fato de envolverem uma quantidade maior de pessoas na sua definição:

- Clientes: no ATDD, os clientes escrevem parte dos testes em linguagem natural. Mesmo assim, um certo nível de treinamento é necessário para que estes testes possam ser automatizados. Em contrapartida, a confiança dos clientes no software

aumenta, uma vez que eles sabem quais os itens do software estão sendo testados e de que forma.

- Gerentes: devido ao tempo necessário para o retorno do investimento inicial para a implementação do ATDD, os gerentes precisam ter a confiança na técnica e patrocinar a sua adoção.
- Testadores: ao contrário dos testes unitários, os testes de aceitação exigem uma quantidade maior de habilidades que extrapolam as que os desenvolvedores normalmente tem, exigindo participação ativa dos testadores no desenvolvimento dos testes.
- Desenvolvedores: ao adotar o ATDD, os desenvolvedores precisam ficar responsáveis por mais um nível de testes, os quais tem o seu custo, uma vez que os desenvolvedores são os responsáveis pelas camadas de automação dos testes. Este custo de desenvolvimento e manutenção pode desencorajar os desenvolvedores a adotar o ATDD. Sendo assim, é necessário que eles tenham sempre em mente os custos relacionados a falhas nos requisitos e o nível de retrabalho associado.

Uma das formas conhecidas de reduzir os custos relacionados à adoção do ATDD é a utilização de *frameworks* de testes, os quais geram uma economia de tempo considerável, pois entregam pronta a infraestrutura necessária para a geração e execução dos testes.

### 2.5. Diferenças entre ATDD e BDD

O BDD (*Behaviour Driven Development*) [North 2006] e o ATDD são duas técnicas que se assemelham muito e por isso constantemente se tem discutido ao longo do tempo as suas similaridades e diferenças [Rebelo 2014]. Como elas são muito similares nos seus objetivos, este documento utilizará ambas como sinônimos, embora ainda exista alguma discussão a respeito, mas foge do escopo deste trabalho.

North define o BDD da seguinte forma:

> *O BDD é uma prática ágil que permite uma melhor comunicação entre desenvolvedores, analistas de qualidade, áreas de negócio e pessoas não-técnicas, durante um projeto de software, descrevendo um ciclo de iterações com saídas bem definidas e resultando na entrega de software testado e que funciona [North 2006].*

Janet Gregory também decide tratá-los como sinônimos:

> *Ao chamarmos de BDD, ou ATDD, ou Especificação por Exemplo, queremos o mesmo resultado - um entendimento comum compartilhado do que será construído para tenta 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 estória ou funcionalidade. A equipe entenderá suficientemente o escopo para então iniciar a codificação e os testes [Gregory 2010].*

Outro termo que reflete uma técnica muito semelhante a estas duas é a Especificação por Exemplos [Adzic 2011], que prega que ao ao invés de criar documentos que

expressam requisitos de uma forma burocrática, muito formal e demasiadamente detalhista devem ser criados documentos expressos em uma linguagem mais natural, focada em comportamento onde o próprio modelo de escrita, além de ser facilmente entendido por qualquer pessoa no projeto, possui um padrão onde pode ser automatizado. Ainda há referências aos termos *Example-Driven Development* (EDD) e *Story Test-Driven Development* (SDD) [Wikipedia 2015].

## 2.6. Questões em aberto no teste de software

Apesar de ser um assunto antigo, ainda há uma quantidade razoável de questões em aberto na área de teste de software [Alrmuny 2014]. Em seu artigo Alrmuny listou alguns pontos em aberto:

- Diferença entre teoria e prática: aproximar os resultados dos diversos estudos relacionados a testes de software com o que é aplicado na indústria de software. A realização de mais estudos empíricos aplicados a cenários reais de desenvolvimento pode reduzir esta distância, mas os custos relativos à modificação dos processos e o receio do compartilhamento de dados de defeitos em cenários reais podem dificultar a realização destes estudos.
- Efetividade indeterminada dos critérios de cobertura: existem diversas propostas de critérios de avaliação de cobetura, mas ainda não está bem estabelecida uma forma precisa de medir a sua efetividade. Existem estudos que buscam realizar a comparação entre métodos de análises de cobertura, mas os seus resultados ainda não são suficientes para guiar a aplicação prática na indústria.
- Dependência do contexto: relacionado ao item anterior, os estudos realizados para a análise de cobertura são em sua grande maioria úteis para o contexto no qual foram desenvolvidos, não sendo genéricos o suficiente para ser adotados como uma padrão.
- Falta de estudos empíricos bem formados: os estudos empíricos se caracterizam por ser caros e trabalhosos, sendo ainda afetados por fatores de difícil controle como validade do sistema sob estudo em relação a sistemas reais, indisponibilidade de dados de falha sendo muitas vezes os dados gerados através de injeção artificial de falhas, forma de geração dos casos de teste e por fim o fator humano devido ao qual os resultados dos estudos são fortemente afetados pela experiência dos testadores envolvidos.
- Evolução do software e paradigmas de desenvolvimento emergentes: sistemas cada vez mais complexos exigem constante adaptação das técnicas de testes, chegando ao ponto de certos domínios de aplicação exigirem técnicas de teste específicas. específicas
- A combinação de diferentes critérios de cobertura e de técnicas de testes garantem melhor a qualidade do software? Diversos estudos empíricos buscaram mostrar que a combinação de técnicas de avaliação de cobertura de testes melhora a qualidade do software. O problema a ser resolvido é que em muitos casos os estudos não avaliam individualmente de forma adequada as técnicas existentes para identificar as que tem desempenho melhor em determinados domínios antes de combiná-las.
- Uso de oráculos de teste: um oráculo é um programa usado para decidir se a resposta de um outro programa a um conjunto de casos de teste é correta ou não e sendo assim a sua escolha afeta diretamente os resultados das avaliações.

- Como reduzir o custo da automação de testes garantindo o desempenho necessário aos times de desenvolvimento e também como fazer para que os resultados em termos de qualidade devidos à automação possam ser corretamente mensurados.
- Inviabilidade de requisitos de testes, nos quais nenhuma combinação de entradas gera a cobertura esperada. Este fator está diretamente relacionado com a avaliação de cobertura dos testes, pois a correta mensuração de cobertura permite identificar quando alguns requisitos de testes são inatingíveis. Um requisito de teste inatingível deve ser identificado o mais cedo possível para evitar desperdício de recursos.

Estes itens serviram como base para a elaboração deste trabalho. Toda a estratégia de testes é baseada na automação e na implementação dos testes certos para as funcionalidades certas, dependendo diretamente da participação dos usuários ou seus representantes. A diferença entre teoria e prática é tratada ao basear as análises em um cenário real, o qual apresenta uma quantidade maior de variáveis fora de controle, como entrada e saída de pessoas do projeto e outras iniciativas de qualidade em paralelo. Apesar disso, os cenários reais tem uma vantagem maior que é a aplicabilidade direta dos resultados para cenários semelhantes. A identificação de requisitos de testes inviáveis é tratada nas reuniões de planejamento de testes, as quais por comunicação direta entre os clientes e os desenvolvedores espera-se que esta identificação ocorra. Estes pontos associados à avaliação da cobertura de forma empírica diretamente pelos *stakeholders* garantem uma melhoria significativa no foco e nos resultados obtidos na qualidade do software e no atendimento dos requisitos.

## 3. Trabalhos Relacionados

Alguns trabalhos mencionam rapidamente os testes de aceitação como ferramenta de descrição de requisitos [Cao e Ramesh 2008] [Gallardo-Valencia e Sim 2009] [Paetsch et al. 2003].

Ricca et al. (2009) caracterizou empiricamente a contribuição dos testes automáticos na fase inicial de levantamento de requisitos por meio de testes de aceitação para especificar o comportamento desejado do sistema para esclarecer os requisitos vindos do cliente usando tabelas como entrada do *framework* Fit tornando-as especificações executáveis dos requisitos do sistema. Os experimentos foram feitos com estudantes de universidades de Trento e Torino, buscando entender em que momentos as tabelas Fit foram suficientes para definir os requisitos e em quais casos são necessário artefatos adicionais para a correta compreensão dos requisitos. O estudo concluiu que houve um aumento de 400% no entendimento do requisito quando comparado ao uso apenas de descrição textual, sem aumento do esforço.

Melnik et al. (2006) usa o método experimental para comprovar que os clientes em conjunto com um desenvolvedor ou analista ligado ao projeto são capazes de utilizar testes de aceitação executáveis para comunicar e validar os requisitos funcionais de negócio, embora muitos clientes mostraram dificuldades de aprendizado da técnica proposta. Foram propostas algumas abordagens para contornar o problema. Este trabalho menciona diversos termos que na visão dos autores se referem a desenvolvimento orientados a testes de aceitação, como *story-test-driven development*, *specification by example* e *functional tests*. Este estudo mostrou que é possível a adoção dos testes de aceitação como ferramenta de requisitos, mas com a ressalva que nas avaliações da túnica pelas pessoas

envolvidas foi observada uma razoável rejeição ao seu uso. O estudo foi conduzido com oito duplas das quais cinco eram formadas por clientes e desenvolvedores e três eram formadas apenas por desenvolvedores. Os melhores resultados obtidos em termos de código e especificação foram observados em algumas das duplas mistas, embora o pior resultado também tenha vindo de uma dupla mista. Isso mostrou a importância da participação dos usuários no processo, sendo decisiva para o seu sucesso ou fracasso.

Haugset e Hanssen (2008) apresenta um estudo de caso de aplicação dos testes de aceitação como ferramenta de descrição de requisitos por uma consultoria de software norueguesa. Foi apresentada uma visão geral dos estudos existentes à época, mostrando os principais pontos que necessitavam de mais estudos para aumentar a aplicabilidade da técnica. Neste estudo foi apontado que é um pouco inapropriado para os clientes expressarem constantemente os requisitos na forma de testes de aceitação, fazendo com que a sua participação. Na maioria dos casos, a escrita de testes a partir dos requisitos foi deixada a cargo dos desenvolvedores. Este trabalho chama a atenção para a necessidade de avaliação prévia do custo de escrita dos testes para que os benefícios sejam suficientes para compensá-los. Além disso, é importante compreender que os testes escritos antes de ter o software que eles testará, gerará uma especificação executável que para se tornar um código de testes livre de erros. Como vantagem foi visto que os testes de aceitação aumentam a confiança entre os desenvolvedores de um mesmo código compartilhado, aumentando a colaboração entre eles. Este trabalho contribuiu significativamente para a análise dos testes de aceitação, mas não descreveu a forma de conectar os testes de aceitação com uma metodologia de desenvolvimento orientado a eles.

Bjarnason et al. (2011) apresenta na forma de estudo de caso as formas como as metodologias ágeis buscam resolver os desafios da Engenharia de Requisitos tradicional e quais os novos desafios elas trazem. Alguns problemas dos métodos tradicionais como lacunas na comunicação e excesso de escopo são tratados pelos métodos ágeis com times multifuncionais, detalhamento gradual e incremental, estórias de usuário e listagem única e contínua de requisitos que permite garantir uma carga de trabalho constante e priorizada. A transição das organizações para os métodos ágeis exige uma adequação completa da estrutura e não apenas das equipes de desenvolvimento, pois exige uma participação maior dos outros setores e o entendimento de que nem todos os requisitos iniciais serão garantidamente atendidos, mas que os mais priorizados certamente serão. Como desafios foram identificadas a necessidade de equipes multifuncionais com conhecimento abrangente nas necessidades dos usuários, requisitos e teste. Este estudo aponta para a necessidade da avaliação dos impactos das metodologias ágeis em produtos de software com ciclo de vida mais longo em termos de qualidade e facilidade de manutenção.

Haugset e Stalhane (2012) discute o uso do ATDD impacta na engenharia de requisitos e estende o *framework* proposto por [Cao e Ramesh 2008] para mostrar como é possível a utilização conjunta de práticas ágeis que facilitam a comunicação e o entendimento e o foco em documentação da engenharia de requisitos tradicional. Ele reforça que metodologias ágeis que possuem menos controles formais necessitam de desenvolvedores disciplinados e clientes dedicados, sempre dando preferência à comunicação direta entre as partes. O artigo buscou responder qual o papel do ATDD na engenharia de requisitos. Segundo a opinião dos autores, o ATDD está posicionado entre a engenharia de requisitos tradicional e a engenharia de requisitos ágil, herdando aspectos de ambos, documentando

os requisitos mas de forma iterativa por meio dos testes executáveis desenvolvidos por intensa comunicação com os clientes. Além de ser a especificação dos requisitos os testes também garantem a verificação automática e servem como documentação forçadamente atualizada pelos desenvolvedores e clientes, pois os desvios em relação a isso resultarão em falhas apontadas pelos testes.

Fontela e Garrido (2013) se refere ao ATDD como uma evolução do TDD, propondo a análise dos benefícios do ATDD do ponto de vista da garantia de qualidade do software para reduzir os riscos da refatoração do código, melhorando a sua legibilidade e preservando o atendimento dos seus requisitos. É descrito um método prático e completo de aplicação do ATDD com esse objetivo. Múltiplas camadas de testes com análise de cobertura dos testes foram definidas para identificar os pontos de falhas nos testes após as refatorações. Este artigo mostrou uma forma mais segura de fazer as refatorações, realimentando o desenvolvimento de software de maneira positiva e segura por meio de testes em diferentes níveis. Foi desenvolvida uma ferramenta específica para a análise de cobertura em diversas camadas.

Este trabalho se diferencia dos demais pelo objetivo de preencher a lacuna de documentação dos método ágeis usando o ATDD associado ao início do desenvolvimento da documentação de usuário antes da implementação. Esta abordagem aumenta o entendimento dos desenvolvedores a respeito do que deve ser implementado pois aos desenvolver a documentação eles precisam explicar o funcionamento daquilo que será desenvolvido. Isto poderia ser feito de diversas outras formas como registrar essa explicação em documentação interna da empresa, mas além de não ver a funcionalidade entregue do ponto de vista do usuário essa abordagem iria contra os princípios ágeis. A abordagem proposta visa melhorar a descrição de requisitos e reduzir o tempo de projeto, uma vez que a documentação de usuário é entrege em conjunto com o software desenvolvido, pois ela já nasce com o projeto e é revisada ao longo dele.

## 4. Metodologia e Proposta

A fim de estudar os impactos da utilização do ATDD como metodologia de desenvolvimento de software esta seção descreve a metodologia utilizada para a avaliação, aplicando o ATDD em um ambiente de desenvolvimento de uma empresa de tecnologia real. Na parte final da seção são analisados os resultados de forma qualitativa focando nos efeitos do ATDD na redução da quantidade de defeitos detectados e na melhoria da comunicação de requisitos entre os envolvidos no projeto de software.

### 4.1. Metodologia

Por se tratar do estudo dos impactos da aplicação do ATDD em um ambiente de desenvolvimento real que não o utilizava, foi decidido adotar pesquisa qualitativa na forma de pesquisa-ação [Wainer 2007]. A abordagem qualitativa foi decidida pelo pouco tempo para a aquisição dos dados e a pesquisa-ação devido à necessidade de atuar no ambiente de observação para iniciar a aplicação da técnica.

A implantação do ATDD em uma organização depende de um mudança de cultura que pode levar meses ou anos, dependendo do tamanho da empresa e da aceitação pelos membros das equipes de desenvolvimento.

Para iniciar incrementalmente a adoção do ATDD foi escolhida uma equipe de 6 pessoas para desenvolver um projeto piloto e permitir a avaliação da metodologia para de acordo com os resultados obtidos propor ou não a adoção desta técnica pelas demais equipes da empresa.

Como a pesquisa-ação depende fortemente da aceitação dos envolvidos para que seja bem sucedida, a abordagem utilizada foi a de introdução gradual dos conceitos do ATDD, evitando formas impositivas de adoção desta nova organização do processo de desenvolvimento de software. Os conceitos foram sendo apresentados e adequados à realidade da equipe a partir do *feedback* dos seus membros.

Como os testes funcionais já estavam implantados como prática de desenvolvimento da equipe, a proposta formulada foi reordenar as práticas já estabelecidas que descreviam os requisitos em alto nível, como documentação de usuário, casos de uso e descrição em linguagem natural dos testes automáticos.

Os passos para uma pesquisa-ação definido por [Baskerville 1999], foram aplicadas como segue:

1. infraestrutura cliente-sistema: baseada nos requisitos do sistema a ser desenvolvido, apenas solicitando permissão para a execução do estudo.
2. diagnóstico: o diagnóstico inicial foi feito com base nas ferramentas de relatos de *bugs* e de revisão de código. Foi avaliada uma versão anterior do software para poder ao término da versão com ATDD realizar a comparação.
3. planejamento e tomada da ação: foi realizado um planejamento baseado treinamentos, exposição da metodologia, *feedback* dos desenvolvedores e avaliação de resultados, com acompanhamento diário das atividades para garantir a correta aplicação da metodologia.
4. avaliação: a avaliação dos resultados foi realizada afim de gerar conhecimento a partir deles. Os resultados foram levemente diferentes dos esperados, sendo a análise de requisitos pouco impactada pelo ATDD, mas a clareza sobre as implementações aumentou consideravelmente.
5. aprendizado: como a análise se baseou em construção de uma teoria, sendo parte da construção de conhecimento após o estudo. Além disso, foram realizadas avaliações informais ao longo da execução para entender o impactos da aplicação do ATDD na motivação da equipe.

O método inicial de coleta dos dados projetado foi a contagem de defeitos e a comparação entre versões, mas esta não se mostrou a melhor abordagem devido à diferença de duração das duas versões e a baixo ocorrência de defeitos. Essa análise seria quantitativa, mas com uma quantidade pequena de dados para analisar, o que forçou o estudo de novas abordagens.

Sendo assim, a abordagem foi modificada para utilizar observação como técnica de aquisição de dados [Seaman 1999]. A aquisição de dados por observação em uma versão passada foi possível devido à análise de documentação e ao uso dos registros do sistema de revisão de código utilizado, o qual armazena histórico das discussões relativas às entregas em análise.

A partir do histórico das discussões foi possível coletar os dados necessários de forma qualitativa através da classificação das observações feitas pelos revisores, que são

os membros da equipe de desenvolvimento que não desenvolveram o código em revisão. Este método de revisão é conhecido como revisão pelos pares [Davies e Berrow 1998]. Esta classificação gerou uma grande quantidade de classes que precisaram ser refinadas para focar nos itens mais importantes e garantir a remoção de classificação duplicadas.

Essa classificação seguiu o método de comparação constante descrito por [Seaman 1999] para a geração de teoria. Havia inicialmente uma suposição de que o ATDD amplia o conhecimento de requisitos tanto pelos desenvolvedores quanto pelos clientes, tornando-os mais claros e fazendo com que o produto entregue se atenda melhor às necessidades do cliente. Como este não era o único efeito possível da aplicação do ATDD, a geração de teoria é mais adequada, pois permite chegar a conclusões a partir dos dados sem focar especificamente na comprovação de algo. Conforme será visto na análise do contexto organizacional, algumas das conclusões não eram esperadas no início do estudo.

## 4.2. Contexto Organizacional

Nesta seção será descrito a forma como o ATDD foi apresentado para uma equipe de desenvolvimento na busca da melhoria do processo de desenvolvimento, buscando melhorar o entendimento dos requisitos. A equipe formada por cinco desenvolvedores, um testador e dois *stakeholders*.

### 4.2.1. Situação Anterior ao Uso do ATDD

A equipe de desenvolvimento que foi escolhida para executar o projeto piloto com o uso do ATDD já trabalhava com métodos Ágeis e desenvolvendo testes unitários e testes funcionais automáticos. Isso facilitou a adoção do ATDD.

O primeiro passo neste trabalho foi realizar uma análise da metodologia de desenvolvimento em uso. A sequência de desenvolvimento seguida até então era a mostrada na figura 3.

A entrada da *user story* no *backlog* ocorre a partir dos requisitos de produto em alto nivel ou de necessidades expressas pelos *stakeholders*. A descrição das *user stories* é realizada em reuniões de planejamento as quais ocorrem no início da iteração Scrum. Esse detalhamento visa listar os critérios de aceitação e esclarecer as dúvidas dos desenvolvedores com relação aos requisitos. A etapa de implementação é realizada com desenvolvimento orientado a testes unitários de módulos. Nesta etapa há a revisão do código gerado e dos testes no nível do módulo. Após a implementação, são desenvolvidos testes automáticos os quais são executados para garantir a qualidade do código gerado. Em caso de indentificação de falhas em alguma destas etapas o desenvolvimento retorna à parte da implementação para a correção do problema e ajuste dos testes. Ao final do processo, a documentação de usuário é gerada e o software gerado é apresentado aos *stakeholders* para demonstração das funcionalidade entregues e aprovação.

Apesar de ter uma grande preocupação com a qualidade, o foco do desenvolvimento era a implementação. Isto trazia resultados perceptíveis no produto final mais cedo, mas nem sempre isto resultava em redução do tempo de desenvolvimento da solução completa.

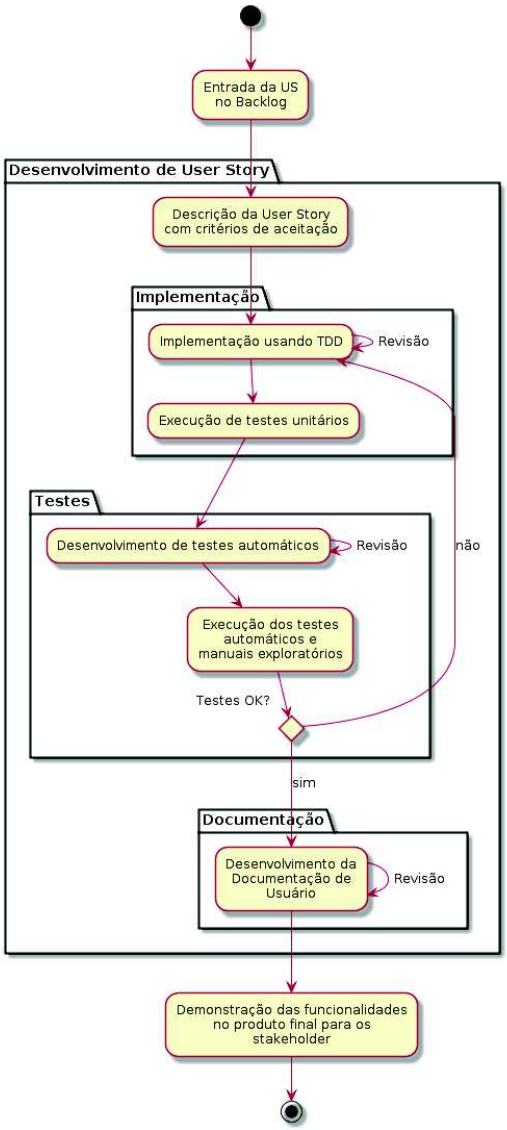


**Figura 3. Fluxo de desenvolvimento anterior.**

A primeira versão das funcionalidades era entregue mais rapidamente, mas em alguns casos o software entregue, apesar de estar livres de erros, não atendia completamente os requisitos quando apresentadas aos *stakeholders*, gerando retrabalho e troca de contexto por parte dos desenvolvedores.

### 4.2.2. Implantação do ATDD

Baseado nos princípios do ATDD, a primeira etapa da adoção da metodologia foi propor a discussão e elaboração dos testes de aceitação automáticos antes do início da implementação. Neste ponto, como relatado em outros trabalhos [Haugset e Hanssen 2008], surgiu a primeira dificuldade, pois era necessário equalizar o conhecimento de todos para o entendimento dos testes.

Como os testes até então eram desenvolvidos para serem analisados e entendidos

pelos desenvolvedores, sendo em muitos casos distantes da linguagem natural cujo entendimento é mais fácil por pessoas sem conhecimento de programação que é o caso de grande parte dos *stakeholders*. Optou-se por utilizar a sintaxe *Given-When-Then* [North 2006] devido ao seu extenso uso para testes de aceitação.

Outro complicador é o tempo demandado pela análise e discussão dos testes, o qual os *stakeholders* não dispunham. Durante todo o desenvolvimento da versão de software um dos objetivos foi otimizar o tempo de interação com os *stakeholders* para garantir a disponibilidade deles ao longo de todo o projeto.

Iniciou-se o projeto desta forma, reunindo o time de desenvolvimento antes, buscando propor a primeira versão dos testes baseados nos requisitos conhecidos. Esta primeira versão era então apresentada aos *stakeholders* para refinamento e esclarecimento de requisitos.

Esta abordagem economizou tempo dos *stakeholders* como esperado, mas gerou um retrabalho grande por parte dos desenvolvedores, pois em alguns casos os requisitos não haviam sido bem entendidos. Para reduzir este retrabalho, foram estudadas outras formas de interação com os *stakeholders*. Documentos de requisitos seriam uma opção, mas iria contra os princípios ágeis de desenvolvimento baseados em mais comunicação e menos documentação. Optou-se por ao invés disso inverter também a ordem de elaboração da documentação de usuário por parte dos desenvolvedores. Isso foi feito porque a documentação de usuário em projetos de produto de longo prazo é algo obrigatório. Neste caso, havia dois tipos de documentação: de comandos e de aplicação. A documentação de comandos tem como objetivo, descrever cada item de interação do usuário com o sistema, enquanto a documentação de aplicação visa descrever um cenário de uso real do sistema, o qual resolve um problema específico do usuário.

Iniciando a etapa de desenvolvimento pelo documento de aplicação pelo produto de software, foi possível para os desenvolvedores definir de forma clara o problema a ser resolvido pela funcionalidade. Sendo este documento de grande importância para os *stakeholders*, foi mais fácil obter a sua atenção do que com a especificação de testes funcionais. O ponto inicial do documento é uma representação gráfica do cenário de aplicação e uma descrição textual justificando questões tecnológicas de interesse do usuário, como normas que são atendidas, e definindo quais são as aplicações para a qual se destina aquela funcionalidade.

Neste ponto se percebe um diferença entre os usuários dos equipamentos de redes e usuários de sistemas de uso geral, eles possuem conhecimento técnico de áreas de engenharia relacionadas ao produto, tornando importante que seja explicado de maneira mais técnica como o produto deve funcionar.

Ao fazer esta inversão na ordem desenvolvimento foram percebidas vantagens, como o entendimento maior dos requisitos por parte dos desenvolvedores, garantia da entrega completa da funcionalidade incluindo a documentação de usuário. Entre os desenvolvedores também foi percebido um aumento no sentimento de propriedade sobre a tecnologia, pois além de saber como ela deve funcionar eles passaram a entender melhor qual a sua finalidade e aplicação para o cliente podendo priorizar o desenvolvimento e a profundidade dos testes de acordo com a criticidade do produto no cenário de aplicação do cliente.

A partir de uma conversa prévia com os *stakeholders* é gerada a primeira versão do documento de aplicação, o qual é enviado para revisão para que os demais membros da equipe e os *stakeholders* possam avaliar a sugerir melhorias. Após a revisão permite que se inicie a discussão a respeito dos testes. Esta primeira versão não é completa do ponto de vista do usuário, pois as imagens de telas capturadas do software e de interação com a linha de comando precisam ser capturadas do software em execução. Ela pode receber protótipos destas saídas do software, mas certamente precisarão ser constantemente atualizadas durante todo o ciclo de desenvolvimento do software.

Após a documentação de aplicação é gerado o documento de descrição de comandos ou de interação, pois ele provê os protótipos das telas necessários para o desenvolvimento do software e para complementar a documentação de aplicação.

Neste ponto o uso do ATDD tem mais subsídios para que os testes possam começar a ser escritos, pois os requisitos já estão mais claros devido à documentação escrita. É importante salientar que neste ponto teoricamente não há esforço maior sendo aplicado do ponto de vista do projeto, pois a escrita destas documentações já é de responsabilidade dos desenvolvedores, ou seja, apenas a ordem foi alterada. Como podemos supor, como as atividades de documentação não são bem aceitas pelos desenvolvedores o desempenho da escrita de documentação antes do código foi abaixo do esperado no início pois os desenvolvedores argumentavam que havia dificuldade em documentar algo que eles não tinham desenvolvido. Neste ponto foi identificado que em muitos casos o conhecimento a respeito dos requisitos era construído ao longo da implementação, o que em uma primeira análise levava a crer que algumas vezes os desenvolvedores trabalhavam em modo de tentativa e erro, pois implementavam o que haviam entendido e ao apresentarem para os *stakeholders* identificavam as diferenças entre o entendimento de ambos quanto às necessidades dos usuários. Claramente, isto é um ponto de retrabalho, que em muitos casos não era identificado como tal, pois ficava dentro do prazo de implementação.

As vantagens do ATDD foram sendo percebidas ao longo do projeto, como por exemplo ao definir o nome usado para identificar uma determinada interface. Inicialmente, pensando de forma apenas programática, pensou-se que um nome é uma *string* e que isso seria requisito suficiente para definir os requisitos. Por se tratar de uma interface de linha de comando foi percebido que caracteres especiais como barras e pontuação poderiam ser interpretados como fim de comando. Ao identificar isso durante a análise dos casos de teste, foram estabelecidas duas possibilidades: tratar todos estes casos especiais ou restringir os caracteres válidos para nomear a interface. A ideia inicial dos desenvolvedores foi tratar os casos especiais, mas ao fazer a estimativa para garantir tanto o desenvolvimento quanto os casos de teste quanto a implementação foi percebido que o retorno de investimento do esforço necessário não compensaria. Como os *stakeholders* estavam envolvidos na definição dos casos de testes, eles definiram que não era um requisito do produto suportar esses caracteres especiais nesta identificação de interface.

Outro bom exemplo das vantagens do ATDD foi percebida quando os desenvolvedores ao pensar nos testes que precisariam ser desenvolvidos em uma *user story* a ser desenvolvida no ciclo de desenvolvimento seguinte, perceberam que para a execução correta dos testes seria necessária a aquisição de um equipamento adicional, o qual possuía prazo de entrega de 15 dias o que inviabilizaria a finalização da US naquele ciclo. A *user story* foi despriorizada e o departamento responsável pela aquisição do equipamento foi

notificado com antecedência de 21 dias.

Muitos outros exemplos das vantagens do ATDD na análise de requisitos durante o projeto poderiam ser citados aqui.

Além do ATDD, o desenvolvimento da documentação de usuário antes da implementação permitiu que os *stakeholders* corrigissem o comportamento do sistema antes da sua implementação minimizando o retrabalho. Essa documentação também serviu de ferramenta de comunicação interna entre os desenvolvedores das *user stories* e o outros envolvidos no projeto como revisores, testadores e *stakeholders*. Ao inverter a ordem da documentação não foram feridos os princípios ágeis pois esta documentação tem como objetivo intensificar a comunicação entre os envolvidos. A sequência de atividades desenvolvida durante o desenvolvimento das *user stories* no contexto do projeto utilizando o ATDD é mostrada na figura 4. Este fluxo de trabalho foi desenvolvido durante a adoção do ATDD no contexto deste trabalho e é baseado no fluxo de ATDD descrito na figura 2 que foi proposta por Hendrickson (2008) e citada por Rebelo(2014).

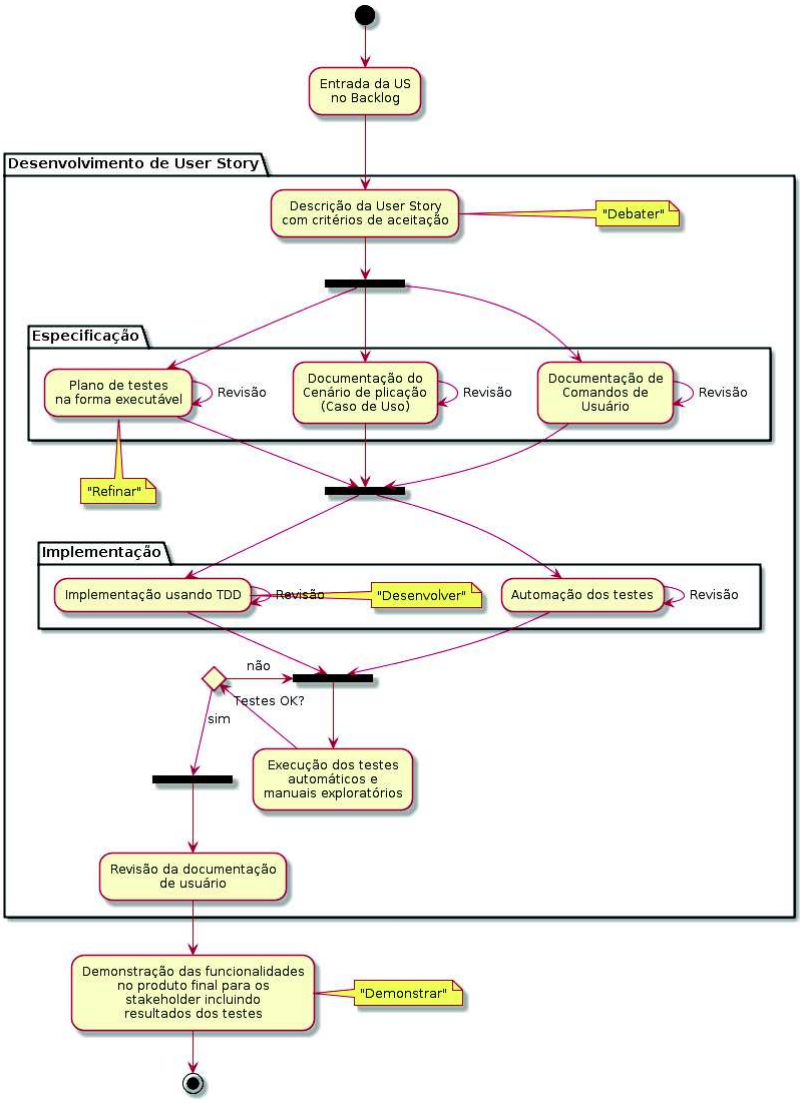


**Figura 4. Fluxo de desenvolvimento ATDD**

Neste diagrama, pode-se ver que a *user story* inicia a partir de uma necessidade discutida entre o *Product Owner* e os *stakeholders*. Tendo essa necessidade identificada, a equipe de desenvolvimento debate a repeito da US para garantir que ela está suficientemente clara para a equipe desenvolvimento e que não há bloqueios para que ela seja desenvolvida. Essa é a etapa de Debate do ATDD.

A próxima etapa do ATDD é o Refinamento feito por meio de colaboração e comunicação direta entre os envolvidos na elaboração e no processo de revisão. Esta etapa é onde o cenário de aplicação, os requisitos e o critério de aceitação são detalhados. A implementação somente inicia quando estas etapas estiverem concluídas ou tão adiantadas a ponto de não representarem riscos de alterações significativas.

Neste ponto inicia o desenvolvimento tanto do software em si usando TDD quanto da automação dos testes funcionais. Ambas as implementações são baseadas em software existente então a quantidade de trabalho em cada ramo depende da diferença entre o software existente e o necessário por esta US em ambos os casos.

Após a implementação é realizada a execução final dos testes sobre o software entregue. Ao executar sem erros os testes permitem que seja realizada a demonstração final para os *stakeholders* a fim de validar a entrega da *user story*.

Apesar do diagrama demonstrar as etapas de forma sequencial e mencionar diretamente a presença dos *stakeholders* no início e no fim do desenvolvimento, os melhores resultados são obtidos quando os *stakeholders* participam de forma consultiva em todo o processo de desenvolvimento analisando versões beta da implementação, validando o conjunto de testes e revisando a documentação de usuário. Além disso, por ser baseado em métodos ágeis, este método de desenvolvimento é incremental mesmo na ???

Comparando com o método de desenvolvimento anterior mostrado na figura 3, há a inversão entre o desenvolvimento de documentação e testes com relação à implementação. Adiciona-se a isso o fato deste desenvolvimento ser paralelizado. Isso traz as vantagens mencionadas de melhor análise e entendimento dos requisitos, buscando sempre o desenvolvimento das features que realmente agregam valor so usuário o mais cedo possível dentro do projeto.

### 4.2.3. Reuniões de planejamento de testes

Após as primeiras iterações com o uso do ATDD foi percebido que o desenvolvimento ainda estava iniciando antes das descrições dos testes serem completadas. Ao questionar os desenvolvedores sobre os motivos, eles relataram que os *stakeholders* nem sempre estavam acessíveis no momento do início do trabalho na US, em função disso também foi identificada a recorrência de *user stories* que levavam muito mais tempo do que o projetado. Para minimizar o problema foi proposto então que antes das reuniões de *review* e *planning* do Scrum fosse feita uma reunião de refinamento das *user stories* focada exclusivamente em testes. Nestas reuniões, os *stakeholders* traziam priorizadas as *user stories* a serem desenvolvidas nas próximas iterações. Analisando sequencialmente estas *user stories* na ordem de prioridade e ampliando os seus critérios de aceitação na forma de itens, os quais eram facilmente convertidos em um ou mais casos de testes a serem descritos pelos desenvolvedores e revisados pelos *stakeholders*.

Essa iniciativa resultou no aumento da quantidade de casos de teste desenvolvidos e em um melhor entendimento das necessidades dos usuários, pois uma vez que a forma como seria testado funcionalmente o software, os desenvolvedores podiam prever uma quantidade maior de comportamentos no software e focar naquilo que era mais importante para os clientes. Essa abordagem visava otimizar o tempo dos *stakeholders* que não estavam conseguindo atender a todas as requisições dos desenvolvedores.

### 4.3. Descrição do Cenário em Análise

Para realizar as análises foram escolhidas duas versões subsequentes do mesmo software. A versão na qual foi aplicado o uso do ATDD foi desenvolvida em 4 meses. A versão imediatamente anterior foi desenvolvida em 9 meses. Essa diferença de tempo se deve à diferença de escopo das versões sendo a primeira uma versão na qual foram desenvolvidas funcionalidades importantes do produto e a segunda uma versão onde inúmeras melhorias nas funcionalidades foram desenvolvidas. Esta diferença de escopo e tempo foi levada em consideração na análise dos resultados.

A primeira métrica utilizada para comparar as versões foi a quantidade de inconformidades apontadas pela equipe de testes. Essas inconformidades apesar de não gerarem impacto para os clientes, pois foram identificadas internamente na empresa, necessitaram de ações corretivas por parte do time de desenvolvimento que já estavam trabalhando em novas funcionalidades. Estas correções podem ter dois custos relacionados, retrabalho e troca de contexto. Ambos causam queda no desempenho da equipe de desenvolvimento.

Como a diferença entre os valores de inconformidades identificadas entre as duas versões foi bastante significativo,como será mostrado no seção a seguir foi decidido utilizar uma segunda metodologia de análise dos dados, para identificar a interação entre os desenvolvedores, os testadores e o *Product Owner* foi decidida utilizar a estratégia de classificação para a construção de teoria mencionada anteriormente.

A fonte dos dados para esta análise complementar foi o sistema de revisão de código utilizado pela equipe de desenvolvimento. Este sistema de revisão foi utilizado em ambas as versões, com e sem o uso do ATDD. Como no momento do desenvolvimento não havia o objetivo de utilizar o sistema de revisão como fonte de dados, a utilização dele como ferramenta de observação não gerou impacto nos resultados.

Partindo de uma *User Story*:

| |
|---|
| **Como um** operador de equipamento de telecomunicações<br>**Eu gostaria** de ajustar o relógio do sistema<br>**Para** ter os registros de eventos sincronizados |

Esse exemplo simples demonstra que apenas a descrição da *User Story* pode não ser suficiente para a correta implementação de uma funcionalidade do software. Ao iniciar a discussão a respeito dos testes desta *User Story*, surgiram alguns casos de testes:

- Ajustar o relógio para hora, minuto e segundos válidos sucesso.
- Tentar ajustar o relógio com a hora inválida esperando mensagem de erro.
- Repetir o teste anterior com minuto e segundo inválidos.

Ao analisar esses testes positivos e negativos [Board) 2012] básicos durante uma

reunião de planejamento de testes, logo surgiram algumas preguntas relacionadas a novos casos de testes, buscando saber se eles eram validos ou não:

- O formato de hora é de 12 ou 24 horas?
- Tem suporte a *timezone*?
- Ajuste automático para o horário de verão?
- A data será ajustada pelo mesmo comando?
- Como é o relacionamento entre o ajuste do relógio e o *log* do sistema?
- É possível ajustar o relógio manualmente com o protocolo SNTP [**?**] habilitado?

Perguntas como essas suscitam o desenvolvimento de novos testes, mas mais do que isso buscam definir melhor os requisitos do software que será implementado. Enquanto a *user story* que apenas ajusta um relógio exige um esforço de implementação diferente de uma que atende todos os requisitos levantados pelo segundo conjunto de perguntas, tanto no desenvolvimento quanto nos testes.

Usando a ferramenta *Robot Framework*, os desenvolvedores e o *stakeolders* puderam escolher o formato de arquivo e linguagem de descrição de testes utilizados para descrever os testes. Neste caso foi escolhido o formato de arquivo texto ao invés de HTML. Como linguagem foi escolhida inicialmente a linguagem *keyword driven* fornecida pelo *Robot Framework*, pela liberdade de escrita que ele propicia, mas ao longo do tempo, por influência de testadores da empresa, os testes foram migrando para o estilo *Given-When-Then* mais usado em BDD. Apesar de usar a linguagem utilizada no BDD, ainda assim este trabalho utiliza o termo ATDD, pois a linguagem usada no BDD foi uma preferência da equipe de desenvolvimento e não uma obrigatoriedade para que o ATDD seja aplicado. Na figura 5 é mostrado um exemplo de caso de teste usando *Robot Framework*.

```
1  *** Test Cases ***
2  Set valid clock
3      Given the system clock is     '01:23:45'
4      When I set the clock to     '23:59:50'
5      Then there is no error message
6      And the new clock value is     '23:59:50'
7
8  Set invalid hour
9      Given the system clock is     '11:22:33'
10     When I set the clock to     '33:02:03'
11     Then the follow error message is shown     'Wrong time format'
12     And the clock value still unchanged
13
14 Set of timezone change the hour
15     Given the system clock is     '11:22:00'
16     And timezone is set to     -3
17     When I set the timezone to     -1
18     Then the hour change to 13
19
20 Clock set is blocked when SNTP is enable
21     enable sntp
22     set clock     '01:02:03'
23     check error message     'SNTP is enabled'
24
```

**Figura 5. Exemplo de teste usando *Robot Framework*.**

### 4.4. Medições Realizadas e Análise de Resultados

Definida a métrica inicial de quantidade de inconformidades, a aquisição foi bastante simples, pois o registro delas pelas equipes de desenvolvimento e testes já estava definida no processo de desenvolvimento da empresa. Através de uma simples aplicação de filtros por versão foi suficiente para chegar aos dados mostrados na figura 6

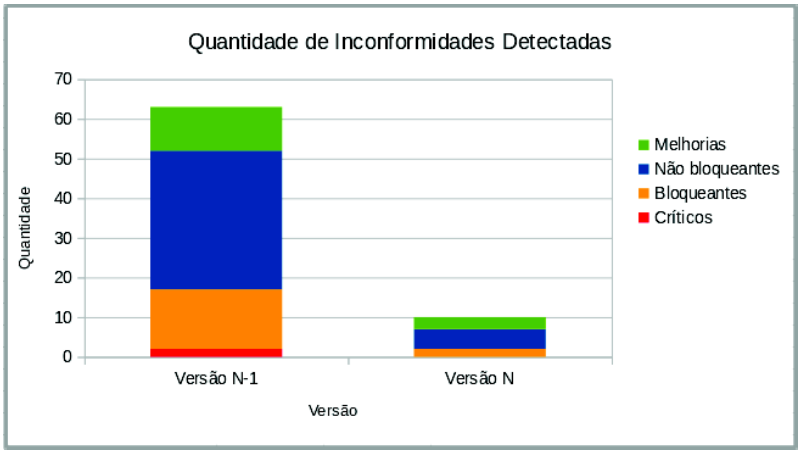


**Figura 6. Quantidade de inconformidades por versão.**

Uma fator que deve ser levado em consideração ao analisar o gráfico é que a versão na qual foi aplicado o ATDD teve duração de 4 meses enquanto a versão sem ATDD teve duração de 9 meses. Para facilitar a comparação entre as versões, foi gerado o gráfico da figura 7, o qual tem a quantidade de inconformidades ponderada como se ambas as versões tivessem 9 meses. A versão N-1 é a versão anterior sem ATDD e a versão N é aquela na qual o ATDD foi aplicado.

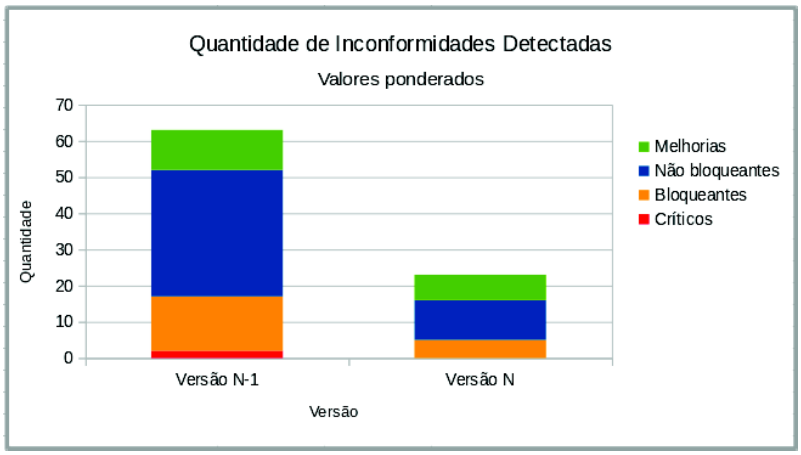


**Figura 7. Quantidade de inconformidades ponderada pela duração da versão.**

Como se pode perceber, mesmo com a ponderação a diferença de inconformidades entre as duas versões foi significativa. Para garantir a validade destes dados para a análise, foi realizada uma avaliação pelos desenvolvedores com o objetivo de definir que outros fatores poderiam ter ocasionado esta variação. Apesar de não poder se afirmar de forma definitiva, foi levantada a possibilidade de que a a natureza das *user stories* se alterou possa ter simplificado o desenvolvimento, pois a versão na qual foi aplicado o ATDD

foi uma versão de melhorias, com uma quantidade menor de funcionalidades complexas. A partir dessa hipótese foi utilizada a abordagem focada nos resultados das revisões mencionada anteriormente.

A aplicação da construção de teoria por observação [Seaman 1999] se baseia no fato de que apenas a análise da quantidade de inconformidades não seria suficiente para a análise desejada na aplicação do ATDD. A partir disto, foi iniciada a aquisição de dados por observação. Essa análise foi realizada no sistema de revisão de código utilizado pela equipe. Todas as observações realizadas pelos revisores foram classificadas manualmente utilizando termos que pareciam melhor descrever a classe no momento da análise das observações. Essa classificação livre visou não gerar um vício de origem onde o classificador consultaria uma lista previamente definida para classificar cada item. Essa lista poderia fazer com que o classificador não encontrasse novas classes, não conhecidas pelos desenvolvedores da lista.

A primeira classificação gerou 38 classes diferentes de observações feitas pelos revisores como mostrado na figura 8.

| qtde | classe | qtde | classe |
|---|---|---|---|
| 1 | abstração | 1 | execução de testes |
| 27 | analise estática | 1 | falta de clareza de requisitos |
| 1 | aprovação | 110 | formatação |
| 15 | arquitetura | 1 | generalização |
| 1 | arquitetura de testes | 1 | histórico |
| 1 | bug | 120 | implementação |
| 1 | clareza | 2 | implementação testes |
| 35 | código desnecessário | 16 | interface de usuário |
| 6 | código duplicado | 1 | legibilidade |
| 4 | compatibilidade | 27 | manutenibilidade |
| 2 | complexidade | 2 | merge |
| 1 | conflito de código | 1 | modificação de requisitos |
| 1 | conflito de merge | 81 | padrão de código |
| 1 | dependências | 2 | portabilidade |
| 1 | desempenho | 1 | proteção código |
| 145 | documentação | 1 | redução de complexidade |
| 5 | duplicação de código | 68 | requisitos |
| 1 | efeitos colaterais | 48 | testes |
| 2 | entendimento requisitos pelo revisor | 1 | tratamento erro |

**Figura 8. Classificação inicial das revisões.**

Essa grande quantidade de classes dificultava a análise, pois haviam muitas classes com uma quantidade pequena de ocorrências, enquanto outras eram sinônimas entre si. Das classes mostradas na figura 8, algumas não foram alteradas, como **análise estática**, **código desnecessário**, **documentação**, **formatação de código**, **interface de usuário** e **padrão de código**. Arquitetura, abstração e portabilidade foram agrupadas em **arquitetura**, **código duplicado** e duplicação de código eram sinônimos, sendo agrupadas no primeiro termo, todos os itens relacionados a **requisitos** e **testes** foram agrupados em classes com estes dois nomes. As duas classes com maior agrupamento foram **implementação** e **manutenibilidade**. As classes novas e antigas são mostradas na figura 9

Com esse agrupamento as classes ficaram com as quantidades mostradas na figura 10. Esta nova quantidade de classes facilitou a comparação dos resultados das duas versões. Definidas as classes foram separados os resultados das duas versões como mostrado na figura 11. Os resultados são mostrados em valores percentuais para compensar o efeito da duração diferente das versões na quantidade.

| qtde | classe antiga | qtde | classe agrupada | qtde | classe antiga | qtde | classe agrupada |
|---|---|---|---|---|---|---|---|
| 27 | análise estática | 27 | análise estática | 16 | interface de usuário | 16 | interface de usuário |
| 1 | abstração | 18 | arquitetura | 4 | compatibilidade | 39 | manutenibilidade |
| 15 | arquitetura | | | 2 | complexidade | | |
| 2 | portabilidade | | | 27 | manutenibilidade | | |
| 35 | código desnecessário | 35 | código desnecessário | 1 | legibilidade | | |
| 6 | código duplicado | 11 | código duplicado | 1 | redução de complexidade | | |
| 5 | duplicação de código | | | 1 | clareza | | |
| 145 | documentação | 145 | documentação | 1 | efeitos colaterais | | |
| 110 | formatação | 110 | formatação | 1 | generalização | | |
| 120 | implementação | 130 | implementação | 1 | histórico | | |
| 1 | aprovação | | | 81 | padrão de código | 81 | padrão de código |
| 1 | bug | | | 68 | requisitos | 72 | requisitos |
| 1 | conflito de código | | | 2 | entendimento requisitos revisor | | |
| 1 | conflito de merge | | | 1 | falta de clareza de requisitos | | |
| 1 | desempenho | | | 1 | modificação de requisitos | | |
| 2 | merge | | | 2 | implementação de testes | 52 | testes |
| 1 | tratamento de erro | | | 48 | testes | | |
| 1 | proteção código | | | 1 | execução de testes | | |
| 1 | dependências | | | 1 | arquitetura de testes | | |

**Figura 9. Agrupamento de classes para simplificação da análise.**

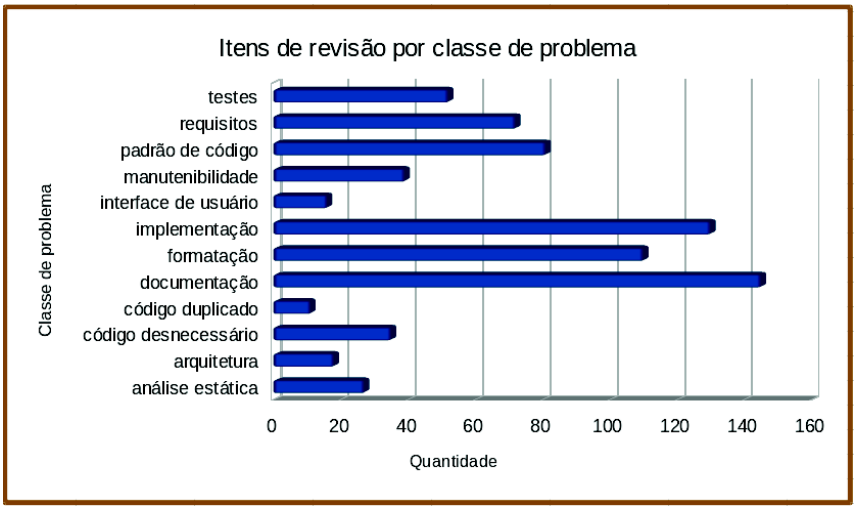


**Figura 10. Agrupamento de classes de observações em revisão.**

Analisando de forma qualitativa o gráfico da figura 11, é possível perceber que houve uma redução na representatividade de itens como implementação e formatação do código na quantidade de itens apontados em revisão. Ao mesmo tempo, houve um aumento na preocupação com a padronização do código (a forma com a qual ele é escrito), com os testes e com a interface com o usuário. A variação é perceptível nos dados, mas ao acompanhar o trabalho da equipe de desenvolvimento foi percebido que esta preocupação ultrapassava os limites da revisão. Durante o trabalho nesta versão com ATDD foi evidente a mudança de cultura na equipe, pois ao invés de focar apenas no código que seria desenvolvido, os membros da equipe passaram a fazer primeiro a se perguntar quais requisitos deveriam ser atendidos, quais os cenários nos quais o cliente mais utilizaria o produto e quais os testes deveriam ser feitos para que estes requisitos fossem verificados de forma automática, garantindo que a sua qualidade se mantivesse ao longo de toda a vida do produto.

Para que os resultados pudessem demonstrar de forma mais clara a transformação ocorrida na qualidade do software desenvolvido utilizando o ATDD estudos quantitativos de is longo prazo precisariam ser realizados focando também na percepção de qualidade

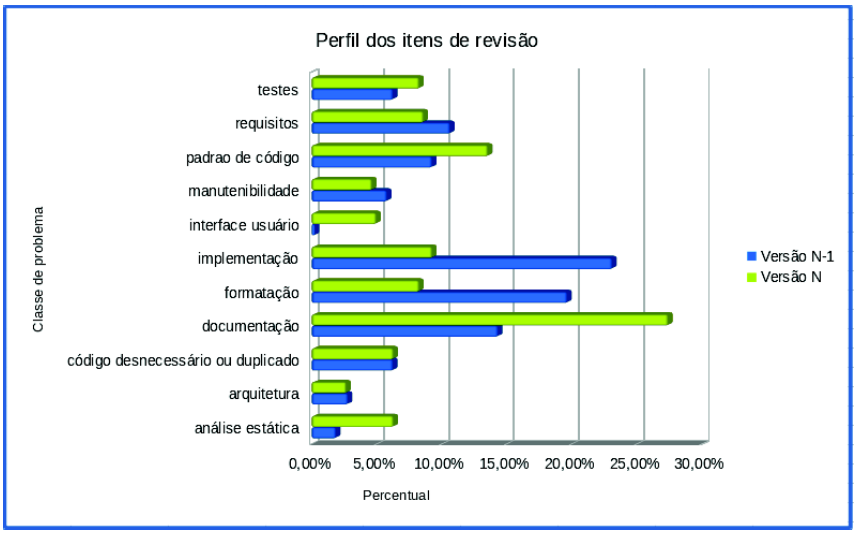


**Figura 11. Agrupamento de classes de revisão por versão.**

por parte dos *stakeholders* e usuários do produto. Mesmo assim, os resultados aqui apresentados já são um indicativo forte de que o ATDD transforma a cultura de desenvolvimento de uma empresa, não sendo apenas uma ferramenta de testes, mas uma poderosa ferramenta de comunicação e refinamento de requisitos. Como a mudança de cultura é algo subjetivo, é um pouco difícil conseguir a comprovação de que esta cultura realmente mudou e de que esta mudança tem efeitos permanentes na organização.

## 5. Conclusão e Trabalhos Futuros

Este trabalho apresentou o ATDD como ferramenta de análise de requisitos e comunicação com os *stakeholders*. Demonstrou que a sua aplicação pode alterar a cultura da empresa, alternando o foco da implementação para a análise de requisitos e forma ágil. Além disso, foi apresentada uma forma de minimizar a falta de documentação característica da aplicação de métodos através do desenvolvimento da documentação de usuário como ferramenta de requisito, o que faz com que a documentação nasça com o projeto, evitando que ela seja postergada indefinidamente. Além dos efeitos mensuráveis, algumas percepções a respeito dos benefícios do ATDD foram apresentadas, como por exemplo a mudança na forma como os desenvolvedores veêm o resultado do seu trabalho e a importância que isso tem para o cliente.

A principal contribuição deste trabalho é a demonstração de que não apenas os testes podem ter a sua ordem de desenvolviment alterada, mas qualquer outro artefato obrigatório do processo, que ao ser desenvolvido de forma antecipada melhora a comunicação entre usuários e desenvolvedores.

Este trabalho se limitou a analisar um domínio de aplicação específico em um tempo curto de desenvolvimento (3 meses). Não foram analisadas o efeito que teriam outras metodologias de desenvolvimento na parte de implementação. Além disso, a principal análise realizada neste trabalho se baseia na percepção dos desenvolvedores refletidas na revisão de código realizada por eles. A alteração do grupo de desenvolvedores pode trazer reflexos do fator humano mencionado anteriormente.

Como trabalhos futuros há algumas direções a seguir, realizar a análise quantita-

tiva do uso do ATDD, desenvolver uma forma de tornar as percepções dos desenvolvedores e cliente algo mensurável, adequar o uso do ATDD para cenários de aplicação onde os requisitos sejão mais estáticos, como no desenvolvimento de novas tecnologias de HW, preservando as funcionalidades existentes.

## Referências


Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Alrmuny, D. Z. (2014). Open problems in software test coverage. *Lecture Notes on Software Engineering*, 2(1):121–125.

Andersson, J., Bache, G., e Sutton, P. (2003). Xp with acceptance-test driven development: A rewrite project for a resource optimization system. In *XP*, pages 180–188.

Baskerville, R. L. (1999). Investigating information systems with action research. *Commun. AIS*, 2(3es).

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., e Thomas, D. (2001). Manifesto for agile software development. `http://www.agilemanifesto.org/`. Acessado em 03-05-2016.

Bjarnason, E., Wnuk, K., e Regnell, B. (2011). A case study on benefits and side-effects of agile practices in large-scale requirements engineering. In *Proceedings of the 1st Workshop on Agile Requirements Engineering*, AREW '11, pages 3–1, New York, NY, USA. ACM.

Board), I. I. S. T. Q. (2012). Standard glossary of terms used in software testing.

Cao, L. e Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software*, 25(1):60–67.

Crispin, L. e Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 1 edition.

Davies, R. e Berrow, T. (1998). An evaluation of the use of computer supported peer review for developing higher-level skills. In *Selected Papers from the CAL 97 Symposium on Symposium*, CAL '97, pages 111–115, Elmsford, NY, USA. Pergamon Press, Inc.

Fontela, C. e Garrido, A. (2013). Connection between safe refactorings and acceptance test driven development. *IEEE Latin America Transactions*, 11(5):1238–1244.

Gallardo-Valencia, R. E. e Sim, S. E. (2009). Continuous and collaborative validation: A field study of requirements knowledge in agile. *Managing Requirements Knowledge, International Workshop on*, 0:65–74.

Gartner, M. (2012). *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*. Addison-Wesley Professional, 1st edition.

Gregory, J. (2010). Atdd vs. bdd vs. specification by example vs ... `http://janetgregory.ca/`

atdd-vs-bdd-vs-specification-by-example-vs/. Acessado em 02-05-2016.

Haugset, B. e Hanssen, G. K. (2008). Automated acceptance testing: A literature review and an industrial case study. *AGILE Conference*, 0(0):27–38.

Haugset, B. e Stalhane, T. (2012). Automated acceptance testing as an agile requirements engineering practice. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5289–5298.

Hendrickson, E. (2008a). Acceptance test driven development (atdd): an overview. http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/. Accessed: 2016-04-28.

Hendrickson, E. (2008b). Driving development with tests: Atdd and tdd. http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf. Accessed: 2016-04-28.

Larman, C. e Vodde, B. (2010a). Acceptance test-driven developmen with robot framework. http://wiki.robotframework.googlecode.com/hg/publications/ATDD_with_RobotFramework.pdf. Accessed: 2016-04-29.

Larman, C. e Vodde, B. (2010b). *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley Professional, 1st edition.

Lucia, A. e Qusef, A. (2010). Requirements engineering in agile software development. *Journal of Emerging Technologies in Web Intelligence*, 2(3).

Melnik, G., Maurer, F., e Chiasson, M. (2006). Executable acceptance tests for communicating business requirements: Customer perspective. In Chao, J., Cohn, M., Maurer, F., Sharp, H., e Shore, J., editors, *AGILE*, pages 35–46. IEEE Computer Society.

NetObjectives (2013). Acceptance test-driven development. http://www.netobjectives.com/acceptance-test-driven-development. Acessado em 02-05-2016.

Nogueira, E. (2016). O que é atdd - acceptance test driven development. http://www.qualister.com.br/blog/o-que-e-atdd---acceptance-test-driven-development. Acessado em 02-05-2016.

North, D. (2006). Introducing bdd. http://dannorth.net/introducing-bdd/. Accessed: 2016-06-25.

Paetsch, F., Eberlein, A., e Maurer, F. (2003). Requirements engineering and agile software development. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '03, page 308, Washington, DC, USA. IEEE Computer Society.

Rebelo, P. (2014). Acceptance test-driven development (atdd), passo a passo. http://www.infoq.com/br/articles/atdd-passo-a-passo. Accessed: 2016-04-28.

Ricca, F., Torchiano, M., Di Penta, M., Ceccato, M., e Tonella, P. (2009). Using acceptance tests as a support for clarifying requirements: A series of experiments. *Inf. Softw. Technol.*, 51(2):270–283.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572.

Wainer, J. (2007). *Atualização em informática 2007*, chapter Métodos de pesquisa quantitativa e qualitativa para a ciência computação, pages 221–262. Sociedade Brasileira de Computação e Editora PUC-Rio.

Watt, R. J. e Leigh-Fellows, D. (2004). *Extreme Programming and Agile Methods - XP/Agile Universe 2004: 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004. Proceedings*, chapter Acceptance Test Driven Planning, pages 43–49. Springer Berlin Heidelberg, Berlin, Heidelberg.

Wikipedia (2015). Acceptance test-driven development –- wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Acceptance_test-driven_development\&oldid=692553117`. Acessado em 02-05-016.