# Dirigindo o Desenvolvimento com Testes: ATDD e TDD

## Uma versão atualizada dos materiais submetidos para minhas apresentações no STANZ 2008 e STARWest 2008

Elisabeth Hendrickson,
Quality Tree Software, Inc.
www.qualitytree.com
www.testobsessed.com

Ao invés de submeter os slides desta sessão, eu submeti o seguinte artigo descrevendo a demonstração que farei.

## Dirigindo o Desenvolvimento com Testes

Em Extreme Programming, os programadores praticam Desenvolvimento Orientado a Teste (TDD). Eles começam a desenvolver escrevendo um teste de unidade que falhe, para demonstrar que a base de código existente ainda não contém tal recurso implementado. Uma vez que se tenha um teste de unidade que falhe, eles então escrevem o código de produção para fazer o teste passar.

Quando o teste estiver passando, eles refatoram o código, enxugando e eliminando duplicações para deixar o código-fonte mais legível e melhorar o design. TDD envolve trabalho em passos muito pequenos, um teste em nível de unidade por vez. Apesar do nome, TDD é uma prática de programação, não uma técnica para testes. A prática de TDD resulta numa suíte automatizada de testes unitários, mas tais testes unitários são um efeito colateral e não o objetivo em si. Praticar Desenvolvimento Orientado a Testes está muito mais relacionado a definir as expectativas quanto à funcionalidade de forma concreta a priori, e fazer com que estas expectativas quanto ao comportamento do código guie a implementação que está sob teste.

Como TDD, o Desenvolvimento Orientado a Testes de Aceitação (ATDD) também envolve criar testes antes do código, e tais testes representam expectativas quanto ao comportamento que o software deve ter. Em ATDD, a equipe cria um ou mais testes em nível de aceitação para uma funcionalidade que está sendo atualmente trabalhada. Normalmente estes testes são discutidos e capturados quando a equipe estiver trabalhando com os responsáveis pelo negócio[1] para compreender uma história no

1 Estou usando o termo "responsáveis pelo negócio" para me referir à pessoa ou às pessoas que têm responsabilidade e autoridade para especificar os requisitos para o software em desenvolvimento. Em Scrum, este é o "Product Owner". Em Extreme Programming, este é o "Cliente". Em algumas organizações estas pessoas são Analistas de Negócio, enquanto que em outras, eles são integrantes da Gerência de Produto. Em cada caso, os responsáveis pelo negócio especificam o "o que" enquanto a equipe de implementação especifica o "como". Os responsáveis pelo negócio definem os recursos a serem implementados em termos do comportamento externo verificável; a equipe de implementação toma decisões acerca de detalhes internos de implementação.

 *Traduzido por Marcelo F Andrade http://twitter.com/mfandrade*

backlog.

Quando capturados em um formato suportado por algum framework de automação de testes funcionais, como o FIT ou o FITnesse, os desenvolvedores podem automatizar os testes escrevendo código de suporte (“fixtures”) de como eles implementam a funcionalidade.

Estes testes proveem feedback sobre o quão perto da conclusão da tarefa a equipe está, nos dando uma clara de progresso.

Este artigo explica o ciclo de ATDD em detalhes, oferecendo exemplos de como se parecem os testes de ATDD e de TDD em vários pontos durante o processo de desenvolvimento.

## Introduzindo a Aplicação de Exemplo

A aplicação de exemplo para este artigo é uma variação de um exemplo clássico de login: é um servidor de autenticação baseado em linha de comando escrito em Ruby. No momento, a aplicação de exemplo permite ao usuário fazer duas coisas:

- Criar uma conta com uma senha
- Efetuar o login com um nome de usuário válido e senha

Uma tentativa de se efetuar login com uma conta de usuário inexistente ou com uma senha inválida vai resultar numa mesma mensagem de erro:

```
Acesso Negado
```

Criar uma conta de usuário resulta simplesmente na mensagem:

```
SUCESSO
```

E ao se efetuar o login com sucesso deve-se obter a mensagem:

```
Bem-vindo
```

Então, por exemplo, se eu executar os seguintes comandos, eu deveria obter as seguintes respostas:

```
> login fred senha
Acesso Negado
> criar fred senha
SUCESSO
> login fred senha
Bem-vindo
```

Isto é tudo que nossa pequena aplicação de exemplo faz no momento. Não é muito. Mas é um começo.

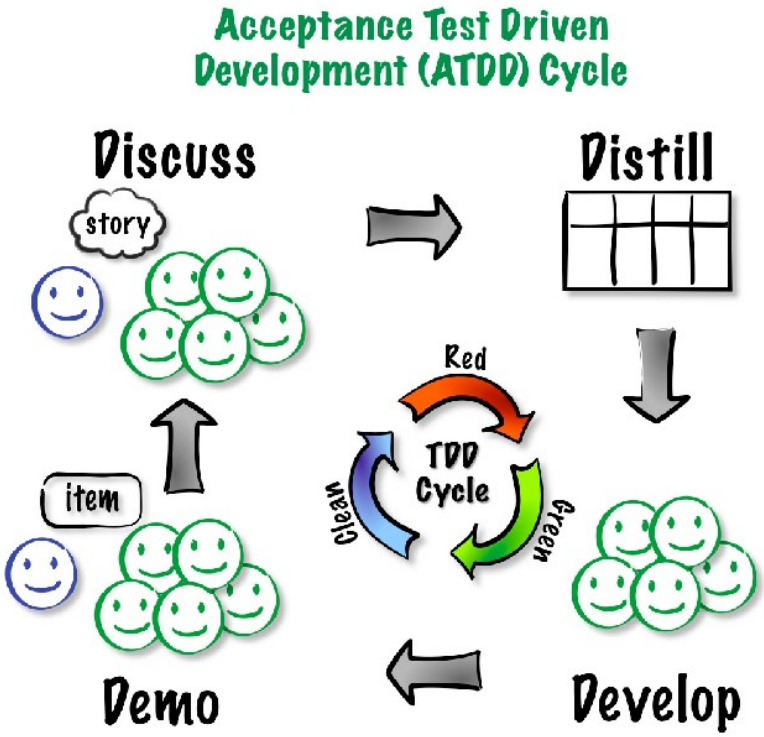
Agora precisamos incluir algumas novas funcionalidades para ela.

## O Backlog

Em nosso projeto de exemplo, a próxima história em nosso backlog priorizado é:

> `Os usuários precisam usar senhas seguras (strings com pelo menos 6 caracteres com pelo menos uma letra, um número e um símbolo)`

## O Ciclo de Desenvolvimento Orientado a Testes de Aceitação (ATDD)




*(Modelo do ciclo ATDD desenvolvido por James Shore com alterações sugeridas por Grigori Melnick, Brian Marick e Elisabeth Hendrickson.)*

## ***Discuta**** os Requisitos

A reunião de planejamento (*Planning Meeting*) é onde discutimos a história sobre senhas seguras. Indagamos os responsáveis pelo negócio, buscando elicitar critérios de aceitação:

> *”O que deve acontecer se um usuário informar uma senha insegura?”*
> *“Você pode nos dar exemplos de senhas que você considera seguras e inseguras?”*
> *“Quais são exatamente os 'símbolos'?”*
> *“E quanto a espaços?”*

* N.T.: *Discuss*

*“E o que fazer com relação a palavras de dicionário com substituições óbvias que atendam aos critérios mais ainda possam ser inseguras, como 'p@ssw0rd'?”*
*“E quanto a contas já existentes?”*
*“Quando você vai considerar que esta funcionalidade está 'funcionando'?”*

Durante a discussão, nós podemos descobrir que mesmo um requisito bem simples esconde inúmeros detalhes. “Hum!”, diria o responsável pelo negócio. “Deveríamos forçar os usuários já existentes que tenham senhas inseguras a atualizar suas senhas no próximo login”. Fazendo as perguntas certas, fazemos também com que o responsável pelo negócio pense cuidadosamente sobre suas expectativas para esta funcionalidade e consiga identificar os critérios de aceitação para a funcionalidade.

Algumas vezes vamos decidir, em equipe, que uma expectativa é, na verdade, uma história nova. “Vamos fazer com que forçar os usuários das contas já existentes atualizem suas senhas inseguras seja uma nova história”, diríamos nós. “Na verdade, isso é uma funcionalidade nova”. O responsável pelo negócio concorda: “Pensando bem, não é tão importante forçar os usuários das contas já existentes a usar senhas seguras. Isso pode ser implementado depois.” Uma consequência natural da discussão sobre os critérios de aceitação para uma funcionalidade num bom nível de detalhes é que isso nos poupa argumentos sobre questões de escopo mais à frente.

Uma vez que tenhamos compreendido o que os interessados no negócio esperam que o software deve fazer, e o que não deve fazer, nós podemos iniciar um esboço dos testes de aceitação junto dos interessados no negócio. Fazemos isto em linguagem natural. Por exemplo:

Senhas válidas que devem resultar em SUCESSO: “p@ssw0rd”, “@@@000dd”, “p@ss w0rd”, “p@sw0d”

Senhas inválidas que devem resultar na mensagem de erro, “Senhas devem ter pelo menos 6 caracteres e conter ao menos uma letra, um número e um símbolo.”: “password”, “p@ss3”, “passw0rd”, “p@ssword”, “@@@000”

## ***Elabore**** os Testes num Formato Adequado a um Framework

Agora que já temos um esboço de nossos testes, nós podemos reescrevê-los num formato adequado para nosso framework de automação de testes. Existem diversos frameworks que suportam definição de testes a priori, como o FIT, Fitnesse, Concordion e o Framework Robot. Neste caso em particular, estou utilizando o Framework Robot.

Nossos testes no Framework Robot são escritos num arquivo HTML parecido com:

| Test Case | Action | Argument |
|---|---|---|
| Verificar senhas válidas e inválidas | Senha deve ser válida | p@ssw0rd |
| | Senha deve ser válida | @@@000dd |
| | Senha deve ser válida | p@wss w0rd |

* N.T.: *Distill*

| | | |
|---|---|---|
| | Senha deve ser inválida | password |
| | Senha deve ser inválida | p@ss3 |
| | Senha deve ser inválida | passw0rd |
| | Senha deve ser inválida | @@@000 |

Claro que há várias maneiras em que poderíamos expressar estes testes, tanto quanto há várias maneiras de capturar as ideias em prosa. Ao relacionar os testes numa tabela, fazemos nosso melhor para expressar a intenção dos testes tão claramente quanto possível sem nos preocuparmos (ainda) sobre como estes testes serão automatizados. Isso significa que devemos focar na essência do teste e ignorar os detalhes de implementação.

Durante a próxima parte do ciclo, quando implementarmos o software, nós associamos os testes com o código.

## ***Desenvolva**** o Código (e Associe-o com os Testes)

Ao implementar o código, se os desenvolvedores estiverem seguindo uma abordagem orientada a testes, eles executam os testes de aceitação e vêem eles falharem. Neste caso, os testes irão falhar com mensagens de erro como estas do Framework Robot:

```
No keywork with name 'Senha deve ser válida' found
```

Esta é uma falha perfeitamente razoável, dado que ainda não temos nada que implemente esta palavra-chave no framework. Expressamos os testes da maneira que quisermos expressá-los, sem nos preocuparmos (ainda) sobre como automatizá-los. Então, agora é a hora de pensar sobre como automatizar os testes escrevendo as palavras-chaves que conectam os testes com o código

Com o Framework Robot, isso significa criar duas palavras-chaves de que precisamos em código de biblioteca.

De maneira semelhante, com o FIT e o Fitnesse, nós poderíamos adicionar código a um Fixture para associar os testes com o código.

Neste caso em particular, porém, como já temos um sistema em execução usando o Framework Robot, nós já temos as palavras-chaves automatizadas para criar uma nova conta e efetuar o login.

Então nós devemos reescrever nossos testes desta maneira:

| Test Case | Action | Argument | Argument |
|---|---|---|---|
| Verificar senhas válidas e inválidas | Criar login | fred | p@ssw0rd |
| | Mensagem deve ser | SUCESSO | |

* N.T.: *Develop*

| | | | |
|---|---|---|---|
| | Tentativa de efetuar login com as credenciais | fred | p@ssw0rd |
| | Mensagem deve ser | Bem-vindo | |
| | Criar login | fred | @@@000d |
| | Mensagem deve ser | SUCESSO | |
| | Tentativa de efetuar login com as credenciais | fred | @@@000d |
| | Mensagem deve ser | Bem-vindo | |
| | *...etc...* | | |

No entanto, isto resulta em testes longos e que ofuscam a essência do que queremos verificar com tanto duplicação. A expressão original dos testes é melhor.

No Framework Robot, podemos criar palavras-chaves a partir de outras palavras-chaves já existentes, assim:

| Test Case | Action | Argument | Argument |
|---|---|---|---|
| Senhas devem ser válidas | [Arguments] | ${password} | |
| | Criar login | fred | ${password] |
| | Mensagem deve ser | SUCESSO | |
| | Tentativa de efetuar login com as credenciais | fred | ${password} |
| | Mensagem deve ser | Bem-vindo | |
| | | | |
| Senhas devem ser inválidas | [Arguments] | ${password} | |
| | Criar login | fred | ${password} |
| | Mensagem deve ser | Senhas devem ter pelo menos 6 caracteres e conter ao menos uma letra, um número e um símbolo. | |
| | Tentativa de efetuar login com as credenciais | fred | ${password} |
| | Mensagem deve ser | Acesso negado | |

O importante a observar aqui é que não há uma sintaxe específica para o Framework Robot, mas ao invés disso, o que precisamos é associar as palavras-chave usadas no teste ao código executável. Diferentes frameworks relacionados a testes ágeis possuem diferentes mecanismos para associar os testes ao código, mas todos eles dispõem de algum mecanismo para fazer tal interrelação entre os testes e o software sob teste.

Agora que implementamos as palavras-chave, podemos executar os testes novamente e obter resultados bem mais significativos. Ao invés de uma mensagem de erro nos dizendo que as palavras-chave não foram implementadas, agora temos testes que falham com mensagens como estas:

```
Expected status to be 'Senhas devem ter pelo menos 6
caracteres e conter pelo menos uma letra, um númro e um
símbolo.' but was 'SUCESSO'.
```

Opa, já tivemos um progresso! Agora temos um teste que está falhando porque a funcionalidade na qual estamos trabalhando ainda não está implementada no software. No momento, as senhas ainda podem ser inseguras. É claro que sabíamos que este teste falharia. Ainda não fizemos nada para fazer o teste passar. Entretanto, executamos o teste mesmo assim. Sempre existe a possibilidade de que tenhamos implementado o teste incorretamente e que ele passe de primeira mesmo quando nenhum código ainda foi escrito. Ver o teste falhar, e nos certificarmos de que ele esteja falhando pelo motivo correto é uma das formas de testarmos nossos testes.

## Implementando Código com TDD

A partir de agora vamos observar como a Jane, a desenvolvedora, implementa a funcionalidade para fazer o teste de aceitação passar.

Primeiramente, a Jane executa todos os testes unitáros para garantir que o código corresponde com todas as expectativas atuais.

Então ela olha o conteúdo dos testes unitários relacionados à criação de uma nova senha. Olhando este pedaço de código em particular, ela encontra uma porção de testes unitários com nomes como:

```
test_valid_returns_true_when_all_conventions_met
test_valid_returns_false_when_password_less_than_6_chars
test_valid_returns_false_when_password_missing_symbol
test_valid_returns_false_when_password_missing_letter
test_valid_returns_false_when_password_missing_number
```

“Que interessante!”, ela pensa. “Parece que já tem algum código escrito para manipular senhas inseguras”. Os testes unitários já estão cumprindo com uma de suas tarefas: documentar o comportamento da base de código existente, como especificações executáveis.

Vasculhando o código um pouco mais a fundo, Jane descobre que ainda que já exista código escrito para determinar se uma senha é válida ou não, este código não está sendo usado. O método para validar uma senha não é chamado em nenhum lugar. É um código morto.

“Ixi...”, Jane pensa consigo mesma. “Código não usado. Argh.”

Revirando o resto dos testes, Jane encontra os testes relacionados à criação de contas de usuário com senha. Ela estuda este teste:

```
def test_create_adds_new_user_and_returns_success
     assert !@auth.account_exists?(“newacc”)
     return_code = @auth.create(“newacc”, “d3f!lt”)
     assert @auth.account_exists?(“newacc”)
     assert_equals :success, return_code
end
```

Jane percebe que o teste simplesmente cria uma nova conta com o nome de usuário “newacc” e senha “d3f!lt” e verifca então se o método retornou “success” como código de retorno.

“Eu aposto que terei um teste que falha se eu fizer um assert da criação de uma conta com uma senha inválida”, Jane diz.

Ela escreve o seguinte teste baseado no original:

```
def test_create_user_fails_with_bad_password
     assert !@auth.account_exists?(“newacc”)
     return_code = @auth.create(“newacc”, “a”)
     assert @auth.account_exists?(“newacc”)
     assert_equals :invalid_password, return_code
end
```

Atente que para escrever este teste, Jane teve de tomar uma decisão sobre o design do projeto, assumindo o que o método “create” deve retornar quando solicitado a criar uma conta com uma senha inválida.

Ao invés de tomar tal decisão escrevendo diretamente código de produção, Jane tomou esta decisão enquanto escrevia testes unitários.  É por isso que TDD é uma técnica mais relacionada a arquitetura e projeto do que a teste de software em si.
Executando os testes unitários, Jane fica satisfeita em ver que seu novo teste unitário falha.  Quando chamado com os parâmetros “newacc” e “a”, o método “create” felizmente retorna “success”.

Jane faz as alterações necessárias no método “create”, usando o método testado mas não usado até então que verifica se as senhas são ou não válidas.

Ela então executa os testes de aceitação novamente, mas ainda vê a mensagem de erro:

```
Expected status to be 'Senhas devem ter pelo menos 6
caracteres e conter pelo menos uma letra, um númro e um
símbolo.' but was 'SUCESSO'.
```

“Okay”, ela diz pra si mesma.  “Eu esqueci de adicionar a mensagem na tabela de mensagens”.

Ela então adiciona a mensagem de erro correta na tabela de mensagens e fica feliz em

ver que o teste de aceitação passa.

A seguir, como Jane e os outros desenvolvedores na equipe praticam integração contínua, ela atualiza sua máquina local com o código mais recente a partir do sistema de controle de versão, faz merges manualmente em todas as alterações que resultaram em conflitos com o código que ela está desenvolvendo, e então executa todos os testes unitários novamente para garantir que tudo continue correto. Quando todos os testes passam, ela faz submete as alterações do código para o repositório.

## Testes Exploratórios e Demonstração

Jane submeteu as alterações no código para suportar a nova funcionalidade, mas ela ainda não terminou.

Ela trabalha com outros na equipe para explorar a funcionalidade. "Vamos tentar usar umas senhas como '&-_\tf00'!", sugere Jack, outro membro da equipe. Jack tenta o comando:

```
> criar wilma &-_\tf00
```

O sistema responde com:

```
-bash: -_tf00: comand not found
```

"Isto é um bug?", Jack pergunta.

"Tenta assim", diz Jane, "colocando aspas na senha", como ela demonstra:

```
> criar wilma "&-_\tf00"
```

O sistema responde com:

```
SUCESSO
```

"Se um usuário entrar com caracteres com significado especial para o shell UNIX, como "&", o shell vai tentar interpretá-los", Jane explica. "Isso acontece em qualquer lugar neste sistema, não apenas no caso desta aplicação". "Talvez precisemos de uma história relacionada a oferecer uma interface diferente que fosse capaz de previnir os usuários de entrar tais caracteres, ou manipulá-los de uma maneira diferente?", ela pergunta.

"Eu acho que deveríamos pensar em falar com o product owner sobre isso", responde Jack.

Como este exemplo demonstra, este tipo de teste exploratório manual é essencial para revelar brechar nos critérios de aceitação e descobrir riscos que ninguém tenha pensado ainda[2].

2 Eu escrevi sobre o uso de Testes Exploratórios em projetos Ágeis em um artigo publicado no *The Art of Agile Development* de James Shore e Shane Warden. Eu recomendo a leitura deste artigo para mais detalhes sobre como

Uma vez que a equipe esteja satisfeita com a implementação que corresponda às expectativas, ela apresenta a funcionalidade para o interessado no negócio, trazendo os riscos em potencial que perceberam durante a implementação e exploração, como a questão do "&" no exemplo acima, e as discussões começam novamente.

## Resultados de ATDD

As equipes que experimentam ATDD normalmente concluem que apenas o ato de se definir testes de aceitação ao discutir requisitos resulta numa melhor compreensão destes requisitos. Os testes em ATDD nos forçam a chegar a um ponto de acordo concreto sobre o exato comportamento que se espera que o software deva ter.

As equipes que seguem o processo todo do começo ao fim, automatizando os testes conforme implementam a funcionalidade, tipicamente concordam que o software resultante é mais testável de uma forma geral, de forma que testes automatizados adicionais são relativamente fáceis de se adicionar. Mais adiante, os testes de regressão automáticos resultantes oferecem pronto feedback valioso sobre expectativas relacionadas a negócio.

## Agradecimentos

As ideias neste artigo têm sido fortemente influenciadas pelas seguintes pessoas, tanto a partir de seus trabalhos publicados quanto em conversas casuais. Este artigo não teria sido possível sem a contribuição delas.

James Shore, Grigori Melnick, Brian Marick, Bas Vodde, Craig Larman, Lasse Koskela, Pekka Klärck, Juha Rantanen, Ran Nyman, Jennitta Andrea, Ward Cunningham, Rick Mugridge, Robert (Uncle Bob) Martin, Alex Chaffee, Rob Mee, Lisa Crispin, Kent Beck, David Astels, Antony Marcano, Joshua Kerievsky, Tracy Reppert e todos os membros da comunidade AA-FTT cujos nomes eu não vou relacionar aqui.

## Recursos

Astels, David (2003). *Test-Driven Development: A Practical Guide*. Prentice Hall PTR.

Beck, Kent (2003). *Test-Driven Development: By Example*. Addison-Wesley.

Crispin, Lisa (2005). "Using Customer Tests to Drive Development." *Methods & Tools*. Summer 2005 Issue. Disponível online em http://www.methodsandtools.com/archive/archive.php?id=23

Koskela, Lasse (2007). *Test Driven: TDD and Acceptance TDD for Java Developers.* Manning Publications.

os Testes Exploratórios incrementam os testes de aceitação. Está fora do escopo desta apresentação discutir sobre Testes Exploratórios em projetos Ágeis.

Marick, Brian (2003). “Agile Testing Directions.” (An explanation of business-facing v. code-facing tests.) Disponível online em http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2

Mugridge, Rick and Cunningham, Ward (2005). *Fit for Developing Software: Framework for Integrated Tests*. Addison-Wesley.

Reppert, Tracy (2004). “Don’t Just Break Software, Make Software: How storytest-driven development is changing the way QA, customers, and developers work.” *Better Software Magazine*. July/August 2004 Issue. Disponível online em http://www.industriallogic.com/papers/storytest.pdf

Watt, Richard J. and Leigh-Fellows, David. “Acceptance Test Driven Planning” http://testing.thoughtworks.com/node/89