Desenvolvimento orientado a testes: Benefícios, técnicas e limitações

ICARO RAMON VERAS CAMELO[1], ANTÔNIO DE BARROS SERRA[2]

[1 2] Instituto Federal de Educação, Ciência e Tecnologia - IFCE
{icarorvc}@gmail.com, {prof.serra}@gmail.com

Palavras Chaves: desenvolvimento, software, engenharia de software, testes, tdd.

Resumo. *Uma abordagem que vem sendo largamente explorada no desenvolvimento de software é o desenvolvimento orientado a testes (Test Driven Development - TDD). O TDD propõe uma abordagem diferente do modelo tradicional em relação aos testes. Decorrente do manifesto ágil, mais precisamente do XP (Extreme Programming), o TDD propõe a criação de testes unitários antes da codificação e não só isso, propõe que o design do software seja incremental e evolutivo [Koskela 2007]. Codificar testes unitários antes do código final garante que seu código só fará aquilo que deve fazer e ajuda a identificar situações de erros em potencial antes mesmo de codificarmos uma linha sequer de código, não sendo necessário gastar horas e horas procurando a raiz do problema nem novos bugs após manutenção, bastando apenas executar toda a suíte de testes e verificar se todos eles foram bem sucedidos. Esse artigo pretende mostrar os benefícios do uso do TDD, as técnicas aplicadas e as limitações.*

Abstract. *Test driven development is an approach which is being widely exploited in software development. TDD (Test Driven Development) purposes a different approach of the traditional method regarding tests. Originated from agile manifest, specifically from XP (Extreme Programming) TDD purposes coding unit tests before the production code, and not only that, says that software design must be incremental and evolutive. Coding unit tests before the production code guarantee that your code is going to do just what it must do and help to identify potential errors before to code even one single line, simply by running the whole test suite and verifying that they were all well succeeded. This article intends demonstrate the benefits, techniques and limitations of TDD.*

1. Introdução

Uma abordagem que vem sendo largamente explorada no desenvolvimento de software é o desenvolvimento orientado a testes (Test Driven Development - TDD). O TDD propõe uma abordagem diferente do modelo tradicional em relação aos testes, propondo a criação de testes unitários antes da codificação e não só isso, propõe que o design do software seja incremental e evolutivo [Koskela 2007]. De acordo com estudos encomendados pelo Departamento de Comércio americano publicados em 2002 [NIST 2002], falhas de software são comuns e tão danosas que chegam a custar aproximadamente 60 milhões de dólares por ano.

Para demonstrarmos os benefícios do desenvolvimento orientado a testes, utilizaremos exemplos de um sistema de mudanças residenciais, um sistema baseado em camadas no padrão (MVC – Model, View, Controller), utilizando frameworks JSF, Spring e Hibernate.

A organização do artigo se dá em oito seções, contando com a introdução. A segunda seção explica a razão da necessidade de testes, seguido das características de um bom teste, explanados na seção três. Na seção quatro, é explanado o (Test Driven Development - TDD), seguido de boas práticas de programação e técnicas de testes, seções cinco e seis, respectivamente. Na seção sete explicamos as limitações do uso do TDD e por fim tratamos dos resultados práticos na seção oito.

2. Por que testar? (Motivação)

Desenvolver software sem testes pode definir o sucesso ou o fracasso de um software, a perda ou ganho de um cliente importante para empresa e gerar grandes problemas como: escalabilidade ruim, alto acoplamento e erros após manutenção. Esses são pontos que preocupam - ou pelo menos deveriam preocupar - a maioria dos desenvolvedores de software ao construir um produto, seja ele para o cenário corporativo ou não.

Segundo Myers [Myers 2004] uma das principais causas da fraca abrangência de testes é o fato de muitos programadores começarem a testar com a falsa definição do termo "teste", onde os mesmos encaram que a fase de testes é uma fase onde não se agrega valor ao software e sim uma fase onde será atestado que o software faz aquilo que deveria fazer.

De acordo com estudos encomendados pelo Departamento de Comércio americano publicados em 2002 [NIST 2002], falhas de software são comuns e tão danosas que chegam a custar aproximadamente 60 milhões de dólares por ano. O estudo também mostra que embora nem todos os erros possam ser removidos, mais de um terço deles poderiam ser eliminados por uma infraestrutura de testes aperfeiçoada, que permitiria uma prévia e mais efetiva identificação e remoção dos erros nos softwares.

De acordo com Arden Bement, diretor do *National Institute of Standard and Technologies – NIST (Instituto Nacional de Padrões e Tecnologias),* o impacto de erros de softwares é enorme pois todo os negócios nos Estados Unidos agora dependem de software para o desenvolvimento, produção, distribuição e suporte pós-vendas de produtos e serviços.

3. O que faz um teste ser bom ou ruim?

Não existe uma regra geral ou método pelo qual podemos mensurar com exatidão se um teste é bom ou ruim, mas existem alguns fatores que podem ser relevantes para assegurarmos que os testes são de boa qualidade.

De uma maneira generalizada, qualquer tipo de teste que precisa ser realizado, não pode ser executado e verificado pelo programador que codificou o programa. Isso se dá porque o programador, bem como qualquer pessoa que executa uma atividade, já está sugestionada a seguir determinados passos, passos estes geralmente os mesmos utilizados na atividade de codificação.

Um bom teste precisa necessariamente ser isolado, ou seja, não deve depender de nenhum outro teste anterior, nem mesmo de dados contidos no banco de dados, embora todo e qualquer código adicionado precisa necessariamente ser testado por todos os testes anteriores, garantindo assim que o novo código não faz os testes anteriores falharem.

Além disso, um bom teste deve cobrir 100% do código implementado, ou seja, se o código em produção foi implementado com determinados comportamentos previstos, como por exemplo o tratamento de exceções, os testes unitários devem ser construídos de tal forma que vislumbre tal comportamento.

4. O que é Test Driven Development (TDD)?

*Test Driven Development – TDD (*Desenvolvimento orientado a testes) é uma abordagem que vem sendo largamente explorada no desenvolvimento de software. O TDD propõe uma abordagem diferente do modelo tradicional em relação aos testes. Decorrente do manifesto ágil, mais precisamente do *XP (Extreme Programming)* o TDD propõe a criação de testes unitários antes da codificação e não só isso, propõe que o design do software deve ser incremental e evolutivo [Koskela 2007].

Diferente do modelo tradicional onde primeiramente modelávamos todo o sistema, geralmente utilizando (UML – *Unified Model Language*), depois implementávamos esse modelo e por fim testávamos o código implementado, o TDD segue basicamente o ciclo abaixo:

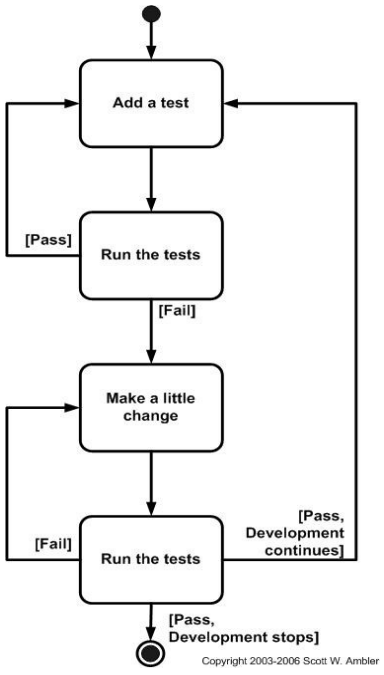

Figura 1: Ciclo do TDD.

a) **Adicione um teste que falhe**: Pense no que o código deveria fazer e escreva um teste unitário para testar essa funcionalidade/comportamento. Nesse

momento não será necessário se preocupar com criação de classes ou métodos. Mantenha-se focado no teste, após tê-lo codificado, escreva o código necessário para executá-lo, para que ele compile e falhe. Não existe uma quantidade limite para criação de testes.

b) **Faça o teste passar**: Escreva o código para que o teste unitário criado previamente seja executado com sucesso. Mesmo certo de que o seu método deve fazer algo a mais, mantenha-se focado em fazer o teste ser bem sucedido.

c) **Refatore o código:** Após o teste ter sido executado com sucesso, verifique o código e tente melhorá-lo. Assegure que o mesmo pode ser reescrito de uma maneira mais elegante, utilizando as boas práticas de programação.

Com essa dinâmica, o TDD propõe que antes de qualquer código final seja adicionado, um teste unitário deve ser criado. Esse pequeno ciclo difere da forma comum utilizada ao desenvolver software no sentido de que o design da aplicação é definido no final, após vários testes e após várias refatorações de código.

5. Refactoring

Um dos princípios do desenvolvimento orientado a testes é a necessidade de refatoração de código. Quando desenvolvemos orientados a testes, estamos focados em implementar o código necessário para fazer com que os nossos testes sejam bem sucedidos. No entanto, à medida que inserimos código novo para produzir esse resultado, muitas vezes criamos código complexo, duplicado e geralmente com uma arquitetura mal definida. A refatoração não adiciona código novo ao código existente, apenas reestrutura-o. De acordo com Martin Fowler [Fowler 1999], *refactoring* é uma ferramenta poderosa que nos auxilia nos seguintes pontos:

a) Melhorar o design do software;

b) Fazer o software mais fácil de entender;

c) Encontrar bugs;

d) Escrever código mais rapidamente;

Segundo ele, devemos refatorar o código quando adicionamos um novo método/função, quando precisamos corrigir um bug e quando fazemos uma revisão de código.

Existem também alguns princípios que podem nos auxiliar no processo de refatoração do código. O artigo de Robert Martin [Martin 2005] propõe cinco princípios de boas práticas de design de software, portanto, podemos utilizá-los no processo de refatoração do código. São eles:

The Single Responsibility Principle (Princípio de responsabilidade única): Uma classe deve ter um, e somente um, motivo para ser alterada.

The Open Closed Principle (Princípio aberto-fechado): Você deve ser capaz de estender o comportamento uma classe, mas não modificá-lo.

The Liskov Substitution Principle (Princípio da substituição de Liskov): Uma classe derivada deve ser substituível por suas classes base.

The Dependency Inversion Principle (Princípio de Injeção de Dependência): Dependa de abstrações, não de concretização.

The Interface Segregation Principle (Princípio de segregação de interface): Codifique interfaces com fina granularidade, específicas para quem vai utilizá-las.

6. Técnicas de testes

Existem basicamente dois tipos de padrões de testes no TDD: o padrão de barras vermelhas (*red bars*) e o padrão de barras verdes (*green bars*). Essas nomenclaturas fazem alusão ao estado do testes, se bem sucedidos ou mal sucessidos de acordo com as cores das barras na IDE *(Integrated Development Environment* – Ambiente de desenvolvimento integrado*)* que variam de acordo com o sucesso do teste unitário, entre bem sucedidos (verde) ou mal sucessidos (vermelhos).

Para os exemplos das técnicas utilizadas no TDD, utilizaremos linguagem de programação Java, JUnit [Massol 2004] (http://www.junit.org), framework de teste unitário Java, versão 4.8 e utilizamos também o framework Mockito (http://mockito.org/), para *mock* de objetos.

Para seguir as boas práticas na realização de testes unitários, criamos *packages* (pacotes) individuais para código em produção e testes unitários.

O sistema utilizado nos exemplos a seguir é um gerenciador de mudanças residenciais.

6.1) **Baby steps**: Desenvolver teste unitário, testar um comportamento/situação por vez, codificar e refatorar, esse é o princípio por trás da técnica *baby steps*.

No exemplo abaixo, criamos uma classe de teste chamada UsuarioTest e um método para verificar se um usuário está ativo ou inativo no sistema:

```
/** Javadoc */
/*Classe de testes da classe Usuario*/
public class UsuarioTest {
	@Test
	public void isAtivo(){
	Usuario usuario = new Usuario();
	Assert.assertTrue(usuario.isAtivo());
	}
}
```

Figura 2: Classe de teste UsuarioTest e método "isAtivo".

Note que sequer temos a classe "Usuario" e que o teste acima nem sequer compila. Precisamos fazer com que o nosso teste compile, então devemos criar a classe chamada "Usuario" e depois disso criar um método chamado "isAtivo".

Escrever testes utilizando a técnica *baby steps* nos permite manter a linha de raciocínio e testar cada elemento necessário para implementar toda a solução para o problema, além de só implementarmos aquilo que realmente precisamos implementar. Cada pequeno teste representa um passo em direção ao nosso objetivo, ou seja, código mais testado possível.

Essa técnica em especial é utilizada implicitamente nas outras técnicas e sem dúvida é a principal técnica utilizada em todo o TDD.

6.2) **Fake it**: A ideia dessa técnica é bastante simples: Qual será a sua primeira implementação após ter escrito um teste unitário que está falhando? Resposta: Utilizar constantes para fazer o teste bem sucedido. Em seguida refatore o código e substitua constantes por variáveis.

Ainda utilizando o exemplo acima, implementamos o método "isAtivo" da classe "Usuario" utilizando a técnica *Fake it:*

```
/** Javadoc */
public class Usuario {
	public boolean isAtivo() {
		return false;
	}
}
```

Figura 3: Método "isAtivo" retornando constante.

Parece razoável pensar: Por que implementar código que não faz o que deveria fazer, somente para fazer com que o teste seja bem sucedido? Simples. Primeiro: Nos proporciona um melhor controle do escopo. Segundo: Nos alerta de que precisamos de mais testes e nos mantém focados no problema, já que a mente dos programadores não para de pensar em diversas situações de erro. Terceiro: essa técnica nos ajuda no aspecto psicológico. Ter os testes bem sucedidos nos ajuda a refatorar o nosso código com mais confiança.

No exemplo dado, percebemos a necessidade de criar um atributo chamado "ativo", substituímos a constante e após algumas refatorações do código, temos:

```
/** Javadoc */
public class Usuario {
	private boolean ativo;
	public boolean isAtivo() {
		return ativo;
	}
	//Getters and Setters (...)
```

Figura 4: Classe Usuario e método "isAtivo" refatorada.

6.3) **One to Many**: O que fazemos se precisamos implementar operações que trabalham com coleções? Primeiramente implementamos o método sem pensar em coleção e após alguns *refactorings,* fazemos o método funcionar com coleções.

O exemplo a seguir mostra um método que precisa retornar o valor total de todos os itens da mudança declarados pelo usuário:

Refatoramos a inicialização do objeto Mudanca, já que precisamos utilizá-lo em vários métodos dos nossos testes, como mostra a figura 5:

```
/** Javadoc */
@Before
public Mudanca inicializarMudanca(){
Mudanca mudanca = new Mudanca();
mudanca.setCode(1l);
mudanca.setStatusMudanca(StatusMudanca.INICIAD
A);
mudanca.setTipoMudanca(TipoMudanca.RESIDENCIAL
);
return mudanca;
}
```

Figura 5: Método inicializa objeto "Mudanca".

Criamos o método para testar apenas um item:

```
/** Javadoc */
@Test
public void precoItensMudancaTest(){
//Utilizando método para inicializar Mudança.
Mudanca mudanca = inicializarMudanca();
mudanca.addItem(new Item(1l,"Cama",1000);

BigDecimal total = new
MudancaServiceImpl().calcularValorTotal(mudanc
a.itens[0]);

Assert.assertEquals(1000,total);
}
```

Figura 6: Método testa apenas um item da coleção.

A seguir, implementamos o teste:

```
/** Javadoc */
public BigDecimal calcularValorTotal(Item
item){
return item.getvalorDeclarado();
}
```

Figura 7: Método implementado para apenas um item.

Após o teste ser bem sucedido, modificamos o teste para testar uma coleção, sem alterar o teste anterior, que é o cerne de outra técnica chamada *Isolation Change*:

```
/** Javadoc */
@Test
public void precoItensMudancaTest(){
//Utilizando método para inicializar Mudança.
Mudanca mudanca = inicializarMudanca();
mudanca.addItem(new Item(1l,"Cama",1000);
mudanca.addItem(new Item(2l,"Mesa",450);
BigDecimal total = new
MudancaServiceImpl().calcularValorTotal(mudanc
a.itens[0],mudanca.itens);

Assert.assertEquals(1450,total);
}
```

Figura 8: Método testa vários itens da coleção.

Em seguida a implementação do novo teste:

```
/** Javadoc */
public BigDecimal calcularValorTotal(Item item
, List<Item> itens){
        BigDecimal total = 0;
        for(Mudanca item : itens){
        total =
total.add(item.getvalorDeclarado());
        }
        return total;
}
```

Figura 9: Método refatorado testa vários itens da coleção.

Após refatoração do código, finalmente temos o código final:

```
/** Javadoc */
public BigDecimal
calcularValorTotal(List<Item> itens){
        BigDecimal total = 0;
        for(Mudanca item : itens){
        total =
total.add(item.getvalorDeclarado());
        }
        return total;
}
```

Figura 10: Método concluído após refatoração.

O exemplo acima é um exemplo muito rico, pois além da técnica *One to Many*, utilizamos o princípio chamado *Isolation Change* (Isolação de mudança), uma vez que adicionamos um parâmetro ao teste sem remover o parâmetro anterior, deixando-o livre para implementarmos novas mudanças sem afetar o caso de teste.

6.4) **Mocks**: Como se testa um objeto que precisa ou depende de um recurso complexo? Crie uma versão falsa desse objeto. Existem mecanismos que nos auxiliam a utilizar essa técnica, nesse artigo, utilizaremos o framework supracitado, Mockito.

Nos exemplos a seguir, temos a classe de serviço "MudancaServiceImpl" e precisamos recuperar os itens contidos na mudança.

Antes de executar o nosso teste principal, precisamos configurar a classe para que antes de iniciar seja criado o objeto falso:

```
/** Javadoc */
public class MudancaServiceImplTest {
MudancaServiceImpl serviceImpl;
@Before
public void init() {
serviceImpl = new MudancaServiceImpl();
serviceImpl.setDao(mock(MudancaDao.class));
        }
}
```

Figura 11: Inicialização do objeto falso MudancaDao.

Após inicializar o objeto falso, temos a implementação do teste:

```
/** Javadoc */
@Test
public void getItensUsuario() {
Mudanca mudanca = new Mudanca();
mudanca.setCode(1l);
mudanca.setStatusMudanca(StatusMudanca.INICIAD
A);
mudanca.setTipoMudanca(TipoMudanca.RESIDENCIAL
);
List<MudancaItem> itens;

when(serviceImpl.getDao().findItensUsuario(mud
anca).thenReturn(null));

Item i = new Item(3l, "Jarro",200);
Item i2 = new Item(4l, "Colchão",300);

when(serviceImpl.getDao().findItens()).thenRet
urn(Arrays.asList(i, i2));

itens = serviceImpl.getItensUsuario(mudanca);

assertEquals(2, itens.size());
}
```

Figura 12: Método "getItensUsuario" com objeto falso .

Com essa técnica também eliminamos outra frequente fonte de erros no desenvolvimento de software: o banco de dados.

7. Limitações

Quando adotamos uma nova prática ou uma nova abordagem é necessário conhecer os contextos nos quais essa abordagem é mais ou menos útil. É preciso também identificar quais são os limites onde podemos operar com essa nova prática, bem como seus pontos fortes e pontos fracos. No caso do TDD existem alguns limites, não somente ligados à área técnica, mas também relacionados a pessoas, onde em determinadas situações não é interessante utilizá-lo. É preciso também deixar claro que isso varia de caso a caso, time para time.

Podemos enumerar alguns desses limites do próprio TDD e de alguns limites para aplicação do TDD:

7.1) **Time não acredita na ideia**: Definitivamente esse é o ponto principal. Se a maioria da equipe não acredita no potencial e na eficácia do TDD, se pensa que testar o software é uma fase enfadonha onde se verifica que o código em produção está fazendo o que deveria fazer é quase impossível obter os benefícios que o TDD traz.

7.2) **Psicológico do desenvolvedor:** Começar a utilizar TDD não é uma tarefa simples. No início é difícil se desligar da maneira tradicional a qual a maioria dos desenvolvedores está acostumada e assimilar a ideia de escrever código parcial, ou seja, escrever código que você sabe que não será definitivo, como o exemplificado na técnica *Fake it.*

7.3) **Tempo curto para o aprendizado**: Se o projeto em questão está com prazos curtos e o nível de complexidade e criticidade das funcionalidades for alto, é necessário analisar a viabilidade do uso do TDD, haja vista os fatores relacionados ao tempo de assimilação da nova abordagem e a produtividade do time de desenvolvimento.

7.4) **Não é possível automatizar testes:** Existem cenários onde não existe disponibilidade de automatização de testes, como por exemplo em softwares desenvolvidos em COBOL, devido a diversos fatores. Apesar de ser uma linguagem de programação antiga, ao contrário do que muitos pensam, ainda existem muitos softwares escritos em COBOL [Mitchell 2006].

8. Conclusão

Nesse artigo foi possível perceber que é factível orientar o código desenvolvido a testes utilizando a abordagem do TDD.

Existem outros testes que nos auxiliam no alcance de alta qualidade de software, como os testes funcionais, testes de integração e integração contínua do código fonte.

Baseado nas referências citadas, percebemos que quando boas práticas de programação e técnicas de testes são aplicadas corretamente, consequentemente produzimos software com código limpo, sucinto, bem estruturado e o mais importante: bem testado.

Referências

[Beck 2002] Beck, K. “Test-Driven Development By Example”, Addison-Wesley Professional, 2002

[Mitchell 2006] Robert L. Mitchell. “Cobol: Not dead yet”. Disponível em: http://www.computerworld.com/s/article/266156/Cobol_Not_Dead_Yet. Acesso em: 25/03/2010

[Myers 2004] - Glenford J. Myers - The Art of Software Testing. Wiley Second edition.

[NIST 2002] National Institute of Standards and Technology- “Software Errors Cost U.S. Economy $59.5 Billion Annually”. Disponível em: http://www.nist.gov/public_affairs/releases/n02-10.htm. Acesso em: 21/03/2010.

[Martin 2005] R. Martin – “Principles of Oriented Object Design”. Disponível em: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. Acesso em: 21/03/2010.

[Massol 2004] Vincent Massol e Ted Husted. “JUnit in Action”. Greenwich: Manning Publications Co, 2004.

[Koskella 2007] Lasse Koskella - “Test Driven: TDD and Acceptance TDD for Java Developers”. Manning Publications.

[Fowler 1999] Martin Fowler - “Refactoring Improving the Design of Existing Code ”. Manning Publications.