

5

J820

Test-driven Development

- *Metodologia de desenvolvimento guiada por testes*
- *Objetivo: produzir "código limpo que funciona" [Beck]*
- *Vantagens de produzir código desse tipo (precisa enumerar?)*
 - *Você sabe **quando terminou** (e não precisa se preocupar com uma longa fase de depuração de bugs): previsível!*
 - *Oportunidade de aprender todas as lições que o código ensina (em vez de pegar a primeira solução que aparecer e esquecer as outras)*
 - *Melhora as vidas dos usuários do seu software*
 - *Melhora as vidas dos membros da equipe*
 - *É bom escrever código limpo... mais ainda, se funciona!*

Como escrever código limpo que funciona?

- *Várias forças nos afastam de código limpo e de código que funciona. Como fugir delas?*
 - *Desenvolver com testes automáticos? TDD!*
- *Em TDD...*
 - *Só se escreve uma nova linha de código se algum teste automático falhar*
 - *Elimina-se duplicação o tempo todo*
- *Implicações*
 - *Design: código funcionando fornece feedback e orienta decisões*
 - *Temos que escrever nossos próprios testes: tempo*
 - *Ambiente de desenvolvimento usado deve responder rápido a pequenas mudanças*
 - *Designs elaborados devem facilitar testes: componentes pequenos, desacoplados, altamente coesos*

Mantra TDD: Red/Green/Refactor

- *Ordem geral do desenvolvimento*
 - **Red**: escreva um pequeno teste que não funciona (e talvez nem compile)
 - **Green**: faça o teste rodar rapidamente, cometendo quaisquer pecados necessários no processo
 - **Refactor**: elimine toda a duplicação criada em meramente buscar o sucesso do teste
- *Por que?*
 - Ter trabalho adicional de escrever testes automáticos?
 - Trabalhar em pequenas etapas se você é capaz de dar grandes saltos e resolver tudo rapidamente?
- *Coragem + comunicação + feedback + segurança*
 - Qualidade, tempo, previsibilidade

Exemplo interativo de TDD

- *Extraído do livro: "Test Driven Development", por Kent Beck, parte I (páginas 3 a 87)*
 - *Estes slides apenas apresentam uma visão geral para servir de base à demonstração em sala de aula*
 - *A demonstração será feita de forma interativa usando o livro como guia, mas em "group programming", portanto, pode ter resultados diferentes se a turma seguir outro caminho*
 - *Use o livro para refazer o exemplo posteriormente e explorar novas técnicas*
 - *Os slides mostram apenas os primeiros 3 (dos 17) capítulos. A aula interativa irá um pouco além*

Dinheiro em várias moedas

- O problema
 - Lidar com uma aplicação que realiza operações em dinheiro de moedas diferentes transparentemente
- Exemplo (relatório de quotas de participação)

IBM	1000	quotas a	25 USD:	Total	25000 USD
Novartis	400	quotas a	150 CHF*:	Total	60000 CHF
Total geral:					65000 USD
- Para realizar as operações, precisamos ter uma taxa de câmbio
 - 1 USD = 2 CHF

Requisitos: anotados em nossa lista!



* CHF = Francos Suíços

Elaboração da lista de tarefas

- *Como solucionar este problema?*
 - *Que conjunto de testes irão demonstrar a presença do código que irá gerar corretamente o relatório?*
- *Começamos anotando em nossa lista (to-do list)*
 - *Precisamos poder somar todas as quantidades em moedas diferentes e converter o resultado dadas as taxas de conversão de câmbio*
 - *Precisamos poder multiplicar uma quantidade (preço por quota) por um número (quantidade de quotas) e receber uma quantidade*

```
$5 + $10 CHF = $10 se taxa for 2:1  
$5 * 2 = $10
```

Lista de tarefas

Quando terminar uma, risque-a
Acrescente novas quando necessário

Começaremos com a multiplicação

```
$5 + $10 CHF = $10 se taxa for 2:1  
$5 * 2 = $10
```

- *Que objetos precisamos?*
 - *Não pense em objetos! Pense em testes!*
- *Que testes precisamos?*
 - *Pense em como a operação vai parecer pelo lado de fora (conte uma estória)*
 - *Escreva o teste da forma mais simples*

```
public void testMultiplication() {  
    Dollar five = new Dollar(5);  
    five.times(2);  
    assertEquals(10, five.amount);  
}
```

Detectou problemas?

Não resolva agora. Coloque na lista!

```
...  
Fazer "amount" private  
Não usar inteiros  
Efeitos colaterais
```

O teste não compila!

- *Vamos consertar! Temos 4 erros*
 - *Não existe a classe Dollar*
 - *Não existe o construtor Dollar(int)*
 - *Não existe o método times(int)*
 - *Não existe o atributo amount*
- *Conserte um de cada vez, só para compilar (não preencha métodos e construtores ainda)*

```
class Dollar {  
    Dollar(int amount) {}  
    void times(int multiplier) {}  
    int amount;  
}
```

- *Agora podemos rodar o teste e vê-lo **falhar***

A falha é um avanço!

- Barra **vermelha** nos impele para buscar a barra **verde**!
 - Temos uma medida concreta de falha
 - O problema mudou de "Me dê uma solução para o problema das múltiplas moedas!" para "Faça este teste funcionar, e depois faça os outros funcionarem!"
 - Muito mais simples: reduzimos o "escopo do medo"
- **Faça o teste passar!**
 - A meta neste momento não é ter a resposta perfeita, mas fazer o teste passar. Qual a menor mudança que podemos fazer para alcançar isto?

```
class Dollar {  
    Dollar(int amount) {}  
    void times(int multiplier) {}  
    int amount = 10;  
}
```

Trapaça! Mas funciona!
Temos uma barra verde!



Está verde, mas não está pronto!

- *Precisamos generalizar antes de prosseguir. O ciclo é o seguinte*
 - *Acrescente um pequeno teste*
 - *Rode todos os testes e assegure que falham*
 - *Faça uma pequena mudança*
 - *Rode todos os testes e assegure que passam*
 - *Refatore o código para remover duplicação (remover a dependência entre o teste e o código - a duplicação é o "sintoma" que indica essa dependência)*
- *Onde está a duplicação?*
 - *Está nos dados! Não vê um **10** nos dois lugares (fora do assert)?*

Removendo duplicação

- Se fizermos o seguinte

```
class Dollar {  
    Dollar(int amount) {}  
    void times(int multiplier) {}  
    int amount = 5 * 2;  
}
```

- O "10" some, revelando 5 e 2 duplicados no teste e no código acima
- Não há como remover a duplicação em uma etapa.
 - Podemos começar movendo a definição do valor de amount para o interior de times()

```
class Dollar {  
    int amount;  
    Dollar(int amount) {}  
    void times(int multiplier) {  
        amount = 5 * 2;  
    }  
}
```

← Os testes passam!
Barra verde!

Removendo duplicação (2)

- E agora, como podemos conseguir um **5** e eliminar o **5** do interior de `times()`?

- Este foi o valor passado ao construtor, no teste!
Podemos guardá-lo na variável de instância **amount**

```
class Dollar {  
    int amount;  
    Dollar(int amount) {  
        this.amount = amount;  
    }  
    ...  
}
```

- E o **2**? Foi passado como parâmetro de `times()` no teste, então está em **multiplier**

```
...  
void times(int multiplier) {  
    amount = amount * multiplier;  
}  
}
```

Terminamos!

- Rodamos o JUnit e... **barra verde!** Só para remover mais um pouco de duplicação (variável amount), rescrevemos a operação em times() para usar o operador *=

```
class Dollar {
    int amount;
    Dollar(int amount) {
        this.amount = amount;
    }
    void times(int multiplier) {
        amount *= multiplier;
    }
}
```

- Rodamos o teste. Tudo OK. Agora podemos riscar um item da lista!

```
$5 + $10 CHF = $10 se taxa for 2:1
$5 * 2 = $10
Fazer "amount" private
Nao usar inteiros
Efeitos colaterais
```

- *Vencemos uma etapa, mas há muito a fazer ainda*
 - *Começamos com **duas** tarefas na lista, acrescentamos mais três, resolvemos uma. Saldo: faltam **quatro**!*
- *Para solucionar o teste, nós*
 - *Escrevemos uma história com um trecho de código que dizia como esperávamos que funcionasse uma operação*
 - *Fizemos o teste compilar com stubs (métodos vazios)*
 - *Fizemos o teste rodar cometendo pecados terríveis*
 - *Gradualmente generalizamos o código, substituindo constantes com variáveis*
 - *Adicionamos itens a nossa lista de tarefas em vez de resolver tudo de uma vez*

Por que ir tão devagar?

- *Por que não resolver todos os problemas de uma vez?*
 - *Se eu já sei fazer contas, por que escrever código a conta-gotas?*
- *TDD não é sobre escrever código passos minúsculos! É sobre **ter a capacidade de** escrever código em passos minúsculos!*
 - *Você não vai escrever código no dia-a-dia com passos tão pequenos, mas, se você pode dar passos tão pequenos, pode dar passos maiores!*
 - *E, se a coisa ficar difícil, reduzir o tamanho dos passos quando for necessário!*

Resumo de um ciclo TDD

- **Escreva um teste**
 - *Escreva uma história: pense em como você gostaria que uma determinada operação funcionasse no seu código*
 - *Invente a interface que você gostaria de ter e inclua todos os elementos necessários para contar sua história*
- **Faça o rodar**
 - *Fazer a barra ficar verde é a atividade mais importante!*
 - *Se você já souber de uma solução limpa e óbvia, escreva! Se é óbvia mas vai levar um minuto, coloque-a na lista e volte ao problema principal, que é fazer a barra ficar verde em alguns segundos*
- **Faça direito**
 - *Agora que o sistema está funcionando, melhore o código, removendo a duplicação introduzida para fazê-lo rodar rapidamente, e faça-o ficar verde rapidamente*

Objetivo: conseguir código limpo que funciona!

Objetos degenerados

- *Resolvemos um problema, mas há alguns efeitos colaterais que persistem na nossa implementação*
 - *O estado do objeto persiste após a execução.*

```
Dollar five = new Dollar(5); // five contém 5
five.times(2); // five agora contém 10
five.times(3); // resulta em 30, e não 15
```
- *É possível descobrir isto acrescentando mais um teste, que falha*

```
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

Esta é a história que estou contando que mostra a forma como eu gostaria que tudo funcionasse.

Não está funcionando!

Solução

- *Uma solução seria sempre devolver um objeto novo de times().*

```
public void testMultiplication() {
    Dollar five = new Dollar(5);
    Dollar product = five.times(2);
    assertEquals(10, product.amount);
    product = five.times(3);
    assertEquals(15, product.amount);
}
```

- *O teste sequer compila. É preciso mudar a interface da implementação também*

```
class Dollar {
    ...
    Dollar times(int multiplier) {
        amount *= multiplier;
        return null;
    }
}
```

*Agora compila, e
falha. Ótimo!*

- *Para fazer o teste rodar, precisamos que ele retorne um objeto Dollar com o resultado correto*

```
class Dollar {  
    ...  
    Dollar times(int multiplier) {  
        return new Dollar(amount *= multiplier);  
    }  
}
```

- *Com isto, eliminamos mais um item da lista*

```
$5 + $10 CHF = $10 se taxa for 2:1  
$5 * 2 = $10  
Fazer "amount" private  
Nao usar inteiros (arredondamento)  
Efeitos colaterais
```

Implementação "óbvia"

- *Para resolver o primeiro teste da lista, começamos com uma implementação falsa e gradualmente chegamos à implementação real*
- *Neste teste, digitamos o que achávamos ser a solução e "rezamos" enquanto os testes rodaram. Por que tivemos sorte e os testes rodaram, seguimos adiante*
- *Usamos duas estratégias para rapidamente alcançar a barra verde*
 - *"Fake it" - retorne uma constante e gradualmente a substitua com variáveis até ter o código real*
 - *"Obvious Implementation" - arrisque uma implementação correta (se falhar, volte ao "Fake It")*

Value Object

- O objeto Dollar deve ser tratado como um valor, ou seja, se for criado um Dollar(5), esperamos que ele seja igual a outro Dollar(5)
 - Podemos implementar Dollar como um objeto de valor, ou Value Object
 - Para isto, precisamos implementar equals()
 - Se formos colocar Dollars em um HashMap, precisaríamos implementar hashCode() também

```
$5 + $10 CHF = $10 se taxa for 2:1  
$5 * 2 = $10  
Fazer "amount" private  
Nao usar inteiros (arredondamento)  
Efeitos colaterais  
equals()  
hashCode()
```

Implementar equals()

- *Como implementar equals()*
 - *Não pense nisto! Pense como testar a igualdade entre dois Dollars. Pense no teste!*

```
public void testEquality() {  
    assertTrue(new Dollar(5).equals(new Dollar(5)));  
}
```

- *O teste falha. Criamos uma implementação falsa só para obter a barra verde*

```
public boolean equals(Object obj) {  
    return true;  
}
```

Triangulação

- Poderíamos resolver o problema com implementação óbvia, mas vamos fazer uma triangulação
 - Esperamos que \$5 seja igual a \$5, mas também que \$5 seja diferente de \$6. Colocamos isto no teste

```
public void testEquality() {  
    assertTrue(new Dollar(5).equals(new Dollar(5)));  
    assertFalse(new Dollar(6).equals(new Dollar(5)));  
}
```

- O teste falha! Agora somos forçados a uma implementação genérica!

```
public boolean equals(Object obj) {  
    Dollar dollar = (Dollar)obj;  
    return amount == dollar.amount;  
}
```


Mais tarefas

- Enquanto pensamos no teste de igualdade, outras perguntas surgem na mente
 - Como comparar Dollars com outros objetos (Francos, por exemplo)
 - Como comparar com null?
- Não resolva isto agora. Coloque na lista!

```
$5 + $10 CHF = $10 se taxa for 2:1  
$5 * 2 = $10  
Fazer "amount" private  
Nao usar inteiros (arredondamento)  
Efeitos colaterais  
equals()  
hashCode()  
Igualdade com null  
Igualdade com objetos
```

- *Algumas sugestões para ir além (seguindo mais ou menos o roteiro do livro)*
 - *Comparar Dollars com Dollars e fazer amount private*
 - *Suportar operações com Francos*
 - *Lidar com Dollars e Francs como objetos similares*
 - *Comparar Francs com Dollars*
 - *Lidar com câmbio*
 - *Multiplicação, soma e subtração em comum*
 - *Abstração e generalização*

[Beck] Kent Beck, "*Test-Driven Development by Example*", Addison-Wesley, 2003



Curso J820
Produtividade e Qualidade em Java:
Ferramentas e Metodologias

Revisão 1.1

© 2002, 2003, Helder da Rocha
(helder@acm.org)

 argonavis.com.br