Test Driven Development (TDD) e *Mock Objects*

Fudamentos de Testes de Software
PUC Minas – São Gabriel

Definição – Mock Objects

- Objetos simulados que imitam o comportamento de objetos reais de um modo controlado.

- São criados para testar o comportamento de outros objetos.
  - Tal como os *crash test dummies* são usados nos testes de automóveis.

Dummy, Fake, Stub e Mock

- ***Test Double*** ("Dublê"): termo genérico para qualquer objeto falso, utilizado no lugar de um objeto real, para propósitos de testes.
- Tipos de dublês:
  - ***Dummy*** *Objects*: são repassados mas nunca utilizados. Normalmente são usados para preencher listas de parâmetros.
  - ***Fake*** *Objects*: têm implementações funcionais, mas normalmente utilizam algum atalho que os torna inadequados para produção (uma base de dados em memória é um bom exemplo).
  - ***Stubs*****:** providenciam respostas pré configuradas para as chamadas feitas durante os testes, normalmente não respondem a nada que não esteja programado para o teste. *Stubs* também podem gravar informações sobre as chamadas, como um *gateway* que lembra as mensagens que 'enviou', ou talvez apenas quantas mensagens 'enviou'.
  - ***Mocks*****:** são objetos pré-programados com informações que formam uma especificação das chamadas que esperam receber.

Dummy, Fake, Stub e Mock

- Destes tipos de dublês, apenas os *mocks* insistem na verificação do comportamento. Os outros podem, e normalmente utilizam verificação de estados.

- *Mocks* se comportam como os outros na fase de exercício, já que precisam fazer o SUT (*Software Under Test*) acreditar que estão falando com seus reais colaboradores - mas diferem nas fases de *setup* e *verify*.

Quando usar *Mocks*

- Quando o objeto real:
  - ainda não existir ou puder alterar o seu comportamento
    - *Test-Driven-Development*
  - retornar um resultado não determinístico
    - Hora Atual, Temperatura Atual
  - possuir estados difíceis de atingir
    - um erro da rede

Quando usar *Mocks*

- Quando o objeto real:
- precisar incluir informação e métodos exclusivamente para uso nos testes
  - e não para a sua tarefa
- for lento
  - uma base de dados completa que tenha que ser inicializada antes do teste

Detalhes técnicos

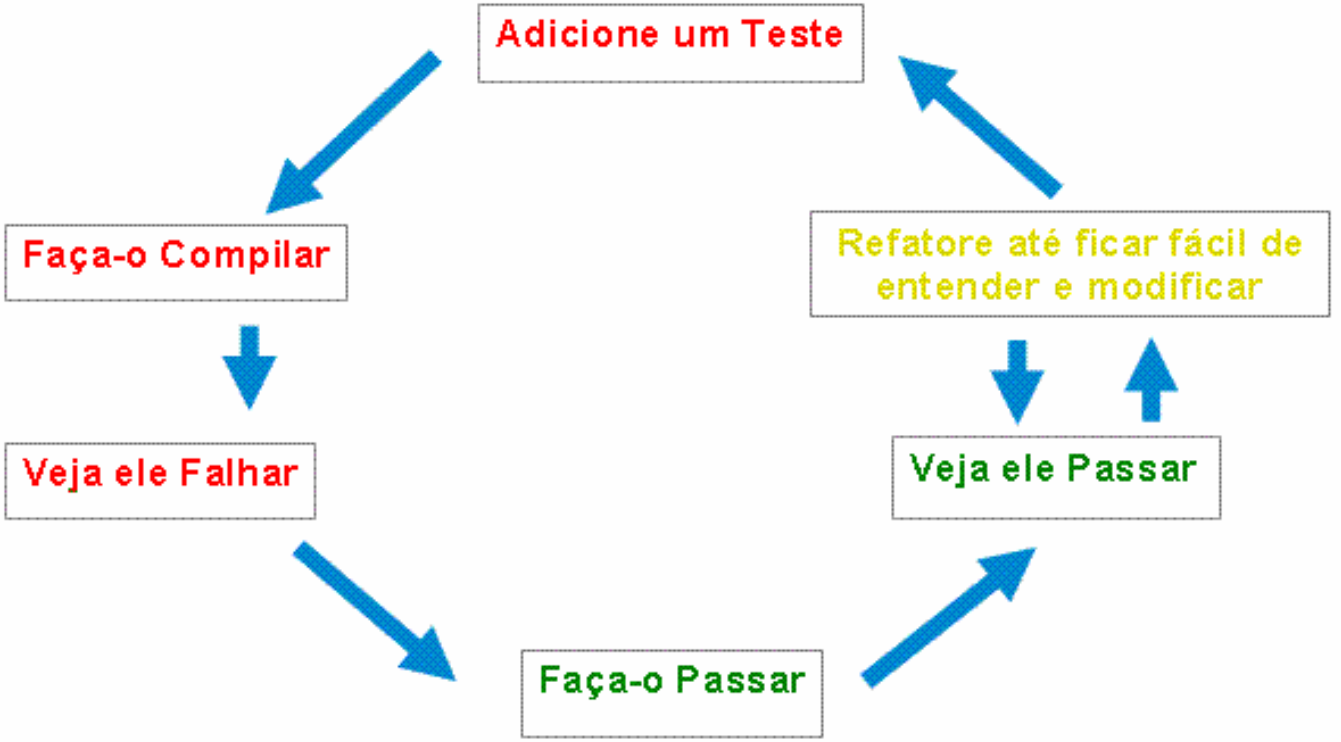
- Permitir que o objeto que o invoca não saiba se está usando um objeto real ou um *mock object*.
- Têm a mesma interface que os objetos que simulam
  - Ambos implementam a mesma interface
  - *Mock object* estende o objeto real

TDD

- *Test Driven Development* (TDD) é uma técnica de desenvolvimento de software que consiste em escrever um teste, escrever um código simples para fazer o teste passar e, ao final, refatorar o código.

TDD

- Mantra do TDD:
  - Escrever um teste
  - Codificar o mínimo possível para o teste passar
  - Refatorar



TDD

- Testes devem ser feitos em **pequenos passos**
- Nunca se codifica sem ter um teste antes!

- Quando encontrar um **bug**, crie um **teste antes** que prove a existência do bug, depois corrija-o.

TDD

- Testes: **documentação executável**!
  - Execute-os **periodicamente**
  - Use **nomes apropriados**
  - Mantenha-os **simples**
- Todas as asserções do JUnit possuem um argumento para descrever o que está sendo testado
  - assertEquals(**"Saldo não é zero!"**, 0, conta.getSaldo());
  - assertNull(**"Conta não é Null!"** , conta);

TDD

- Quando escrevemos um teste antes de codificar estamos pensando em:
  - **design**
  - **teste**
  - **documentação**

TDD

- Ao escrever testes, o que realmente é procurado?
  - testes bem focados: testar partes específicas das aplicações;
  - testes independentes: que os testes rodem em qualquer ordem e a qualquer momento;
  - testes rápidos: que os testes sejam rodados frequentemente.

TDD

- Infelizmente há algumas dificuldades:
  - testes bem focados implicam em uma grande quantidade de testes;
  - torná-los independentes necessita de uma forma eficiente de limpar os objetos utilizados (uma vez que esses objetos podem ser utilizados por outros testes);
  - testar partes isoladas de aplicações complexas não é uma tarefa trivial

Mock Objects

- Mock Objects tem o objetivo de simplificar ou amenizar estes problemas.
- Mock Objects são utilizados quando é difícil ou impossível criar um determinado estado do sistema sendo testado, ou quando o acesso ao recurso é limitado.
- O princípio básico dos Mocks é criar objetos leves e controláveis para simplificar a construção de seus testes.

TDD

- Deve-se utilizar Mocks nas seguintes situações:
  - Deixar os testes mais focados
  - Fazer com que os testes rodem mais rápidos
  - Simplificar o desenvolvimento de aplicações que conversam com hardware, dispositivos remotos, ou mesmo, com outras aplicações.
  - Promover um desenvolvimento baseado em uma interface
  - Encorajar o uso de composição em vez de herança
  - Testar situações incomuns

Exemplo prático

```
public class Convert {
    …

  public static double getTemperatureInCelcius(City city){
    double tempF;
    Sensor sensor = city.getSensor();
    tempF = sensor.getCurrentTemperature();

    //Convert Fahreneit to Celsius degrees
    return (tempF-32.0)*(5.0/9.0);
  }

    …
}
```

Classe City não existe

Resultado não determinístico

Exemplo prático

(continuação)

```
public class Convert {
          …

     public static double getTemperatureInCelcius(City city){
     double tempF;
     Sensor sensor = city.getSensor();
     tempF = sensor.getCurrentTemperature();


     return (tempF-32.0)*(5.0/9.0);
     }
          …
}
```

```
public class MockSensor extends Sensor {
     …
   public double getCurrentTemperature() {
     //MOCK
     return 72.5;
   }
}
```

Resultado conhecido (=22.5ºC)

Exemplo prático

(continuação)

```
public class Convert {
        …

    public static double getTemperatureInCelcius(City city){
    double tempF;
    Sensor sensor = city.getSensor();
    tempF = sensor.getCurrentTemperature();


    return (tempF-32.0)*(5.0/9.0);
    }
        …
}
```

```
public class MockCity implements City {
    …
   private Sensor curSensor;

   public Sensor getSensor() {
      //MOCK
      return curSensor;
   }

   public void setSensor(Sensor s){
      //MOCK
      this.curSensor = s;
   }
}
```

Escrever método para controlar valor de retorno

Exemplo prático

(continuação)

```
public class TestConvert extends TestCase {

   public void testGetTemperatureInCelcius() {
      MockCity city = new MockCity();
      MockSensor sensor = new MockSensor();
      city.setSensor(sensor);

      assertEquals(22.5, Convert.getTemperatureInCelcius(city));
   }
}
```

Em suma…

- Escrever classes *mock*
- Estender classe real
- Implementar interface
  - Implementar **todos** os seus métodos
- Reescrever (*overwrite*) os métodos a simular
- Criar e executar os testes

Frameworks

- Facilitam e tornam mais rápida a criação dos *Mock Objects*
  - Não é necessário escrever o código das classes *mock*
- Permitem criação a partir de:
  - Interfaces
  - Classes
- Permitem definir expectativas flexíveis
  - Número de chamadas a métodos
  - Valores de retorno
  - Valores dos parâmetros
  - Lançamento de exceções

Frameworks

- Integração com jUnit (Java)
- APIs extensíveis
  - Grande parte dos *frameworks*
- Valorizam a qualidade dos testes

Frameworks

- Java
  - jMock (http://www.jmock.org)
  - EasyMock (http://www.easymock.org)
  - rMock (http://rmock.sourceforge.net)
  - MockCreator (http://mockcreator.sourceforge.net)
  - MockLib (http://mocklib.sourceforge.net)
- C#
  - MockLib (http://sourceforge.net/projects/mocklib)
  - Rhino Mocks (http://www.ayende.com/)
  - NMock (http://nmock.org)
  - Microsoft Fakes

Frameworks

- Ruby
  - Mocha (http://mocha.rubyforge.org)
  - RSpec (http://rspec.rubyforge.org)
  - FlexMock (http://onestepback.org/software/flexmock)
- Frameworks para outras linguagens
  - C++
  - Javascript
  - Perl
  - Python
  - ActionScript

25

Exemplo jMock

```
public class Convert {
    …

  public static double getTemperatureInCelcius(City city){
    double tempF;
    Sensor sensor = city.getSensor();
    tempF = sensor.getCurrentTemperature();

    //Convert Fahreneit to Celsius degrees
    return (tempF-32.0)*(5.0/9.0);
  }

    …
}
```

Classe City não existe

Resultado não determinístico

Exemplo jMock

(continuação)

```
public class JMockTestConvert extends MockObjectTestCase {

    public void testGetTemperatureInCelcius() {
        // set up
        Mock mockCity = mock(City.class);
        Mock mockSensor = mock(Sensor.class);

        // expectations
        mockCity.expects(once()).method("getSensor").
                will(returnValue((Sensor) mockSensor.proxy()));
        mockSensor.expects(once()).method("getCurrentTemperature").
                will(returnValue(72.5));

        // execute
        double result = Convert
                .getTemperatureInCelcius((City) mockCity.proxy());

        // test
        assertEquals(22.5, result);
    }

}
```