Modulo II
Mock Objects

Prof. Ismael H F Santos

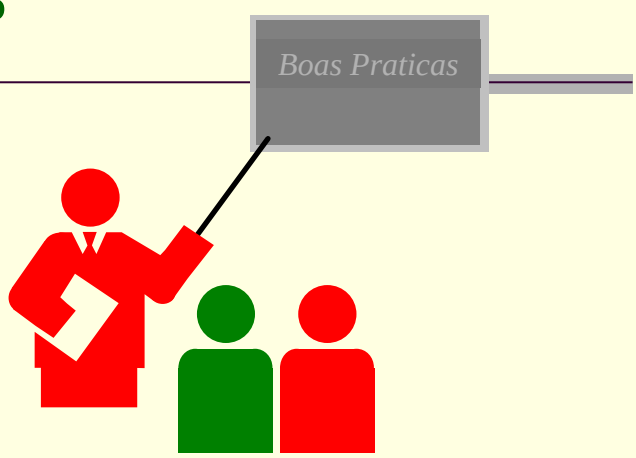
Bibliografia

- *JUnit in Action*

Agenda:

- Mock Objects

MA-XP

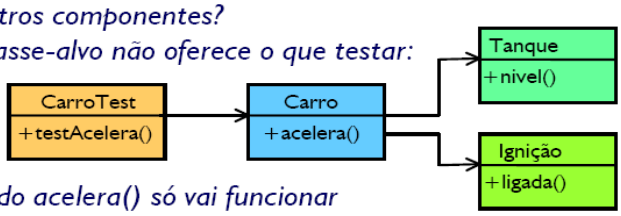


Como lidar com testes difíceis

- *Testes devem ser simples e suficientes*
  - ***XP**: design mais simples que resolva o problema; sempre pode-se escrever novos testes, quando necessário*
- *Não complique*
  - *Não teste o que é responsabilidade de outra classe/método*
  - *Assuma que outras classes e métodos funcionam*
- *Testes difíceis (ou que parecem difíceis)*
  - *Aplicações gráficas: eventos, layouts, threads*
  - *Objetos inacessíveis, métodos privativos, Singletons*
  - *Objetos que dependem de outros objetos*
  - *Objetos cujo estado varia devido a fatores imprevisíveis*
- *Soluções*
  - *Alterar o design da aplicação para facilitar os testes*
  - *Simular dependências usando proxies e stubs*

Dependência de código-fonte

- *Problema*
  - *Como testar componente que depende do código de outros componentes?*
  - *Classe-alvo não oferece o que testar:*



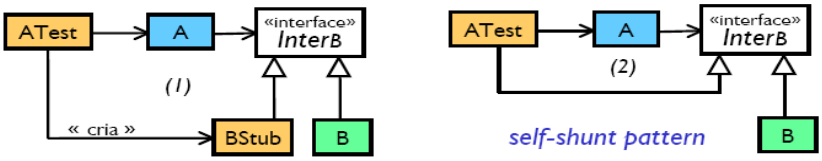
- *Método acelera() só vai funcionar se nível() do tanque for > 0 e ignição estiver ligada()*
- *Como saber se condições são verdadeiras se não temos acesso às dependências?*

```
public void testAcelera() {
    Carro carro =
              new Carro();
    carro.acelera();
    assert???(???);
}
```

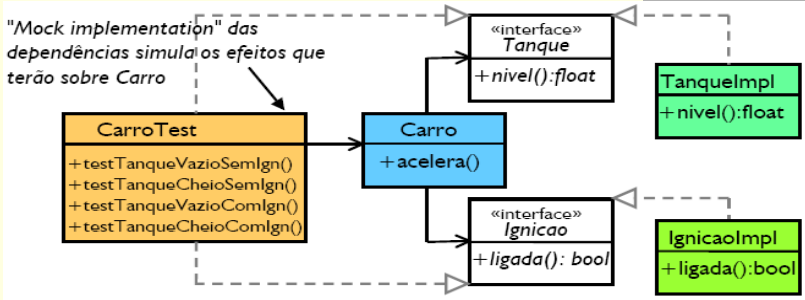
Stubs: objetos "impostores"

- *É possível remover dependências de código-fonte refatorando o código para usar interfaces*

- *Agora B pode ser substituída por um stub*
  - *BStub está sob controle total de ATest (1)*
  - *Em alguns casos, ATest pode implementar InterB (2)*



Dependência: solução usando stubs



- *Quando criar o objeto, passe a implementação falsa*
  - *carro.setTanque(new CarroTest());*
  - *carro.setIgnicao(new CarroTest());*
- *Depois preencha-a com dados suficientes para que objeto possa ser testado*

Dependências de servidores

- Usar **stubs** para simular serviços e dados
  - É preciso implementar classes que devolvam as respostas esperadas para diversas situações
  - Complexidade muito grande da dependência pode não compensar investimento (não deixe de fazer testes por causa disto!)
  - Vários tipos de stubs: mock objects, self-shunts.
- Usar **proxies** (mediadores) para serviços reais
  - Oferecem interface para simular comunicação e testa a integração real do componente com seu ambiente
  - Não é teste unitário: teste pode falhar quando código está correto (se os fatores externos falharem)
  - Exemplo em J2EE: **Jakarta Cactus**

Mock Objects

- **Mock objects** (MO) é uma estratégia de uso de stubs que não implementa nenhuma lógica
  - Um mock object não é exatamente um stub, pois não simula o funcionamento do objeto em qualquer situação
- Comportamento é controlado pela classe de teste que
  - Define comportamento esperado (valores retornados, etc.)
  - Passa MO configurado para objeto a ser testado
  - Chama métodos do objeto (que usam o MO)
- Implementações open-source que facilitam uso de MOs
  - EasyMock (tammofreese.de/easymock/) e MockMaker (www.xpdeveloper.com) geram MOs a partir de interfaces
  - Projeto MO (mockobjects.sourceforge.net) coleção de mock objects e utilitários para usá-los

Exercicio

- 1. A classe *Exercicio.java* possui quatro métodos vazios:

```
int soma(int a, int b)
long fatorial(long n);
double fahrToCelsius(double fahrenheit);
String inverte(String texto);
```

  Escreva, na classe *ExercicioTest.java*, test-cases para cada método, que preencham os requisitos:
  - ***Soma***: 1+1 = 2, 2+4 = 4
  - ***Fatorial***: 0! = 1, 1! = 1, 2! = 2, 3! = 6, 4! = 24, 5! = 120
    A fórmula é n! = n(n-1)(n-2)(n-3)...3*2*1
  - ***Celsius***: -40C = -40F, 0C=32F, 100C=212F
    A fórmula é F = 9/5 * C + 32
  - ***Inverte*** recebe "Uma frase" e retorna "esarf amU"
- Implemente um teste e execute-o (o esqueleto de um deles já está pronto). Ele deve falhar.
- Implemente os métodos, um de cada vez, e rode os testes até que não falhem mais (tarja verde), antes de prosseguir.