Componentes de Teste

Criado: set/2010
Últ. Atualiz.: abr/2013

Tópicos

- Noção de drivers, stubs e mock objects
- Estrutura da implementação de um caso de teste
- Padrões para construção de stubs e mocks

Referências

R.Binder. Testing OO Systems. Addison Wesley, 1999, c.16-19.

- Onde encontrar tutoriais sobre JUnit:
  - http://open.ncsu.edu/se/tutorials/junit/
  - www.cs.uoregon.edu/education/classes/05S/cis410sm/lecture-slides/JUnitTutorial.ppt
  - www.cs.wm.edu/~noonan/cs301/labs/**junit**/**tutorial**.html
  - supportweb.cs.bham.ac.uk/documentation/**tutorial**s/docsystem/build/**tutorial**s/**junit**/**junit**.pdf
  - www.cs.wm.edu/~noonan/cs301/labs/**junit**/**tutorial**.html
  - ...

Mais referências

Vincent Massol e Ted Husted. Junit in Action, cap7. Manning Publications, 2003.

Martin Fowler. “Mocks aren't stubs”. Atualizado em jul/2004 no seguinte endereço: /www.martinfowler.com/articles/mocksArentStubs.html

Sobre padrões para definir mocks:

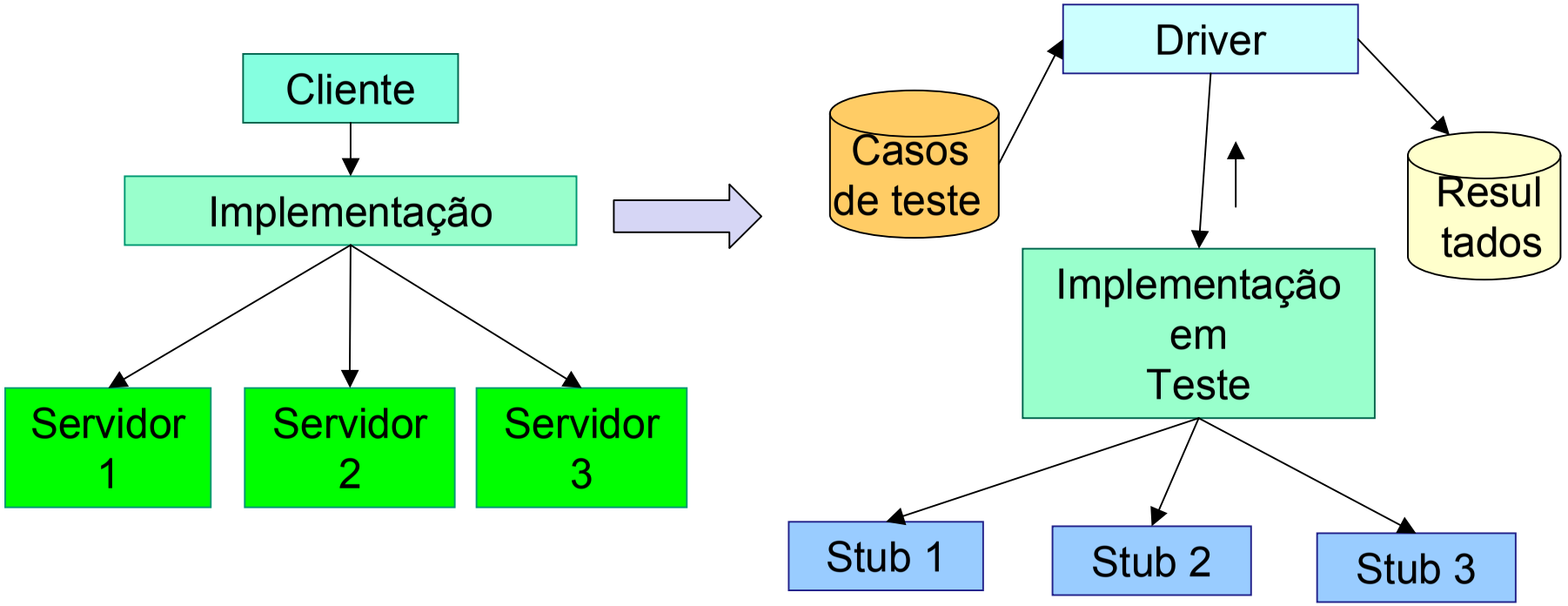
G. Meszaros. : A Pattern Language for Automated Testing of Indirect Inputs and Outputs using XUnit. PLOP 2004. Obtained in jan/2006 at: **http://testautomationpatterns.com/TestingIndirectIO.html**

S. Gorts. Unit testing with hand crafted mocks. Last updated on sept/2004. Obtained at: **http://refactoring.be**.

Componentes de teste (1)

- **Driver**
  - Programa ou classe que aplica os casos de teste ao componente em teste
  - Faz o papel de cliente do componente em teste (**CeT**).
- **Stub**
  - Implementação temporária, mínima, de um componente usado pelo CeT, com o objetivo de melhorar a controlabilidade e observabilidade do CeT durante os testes. Faz o papel de servidor do CeT.
- **Test Harness**
  - Sistema que compreende os drivers, stubs, CeT e outras ferramentas de apoio aos testes.

Componentes de testes (2)

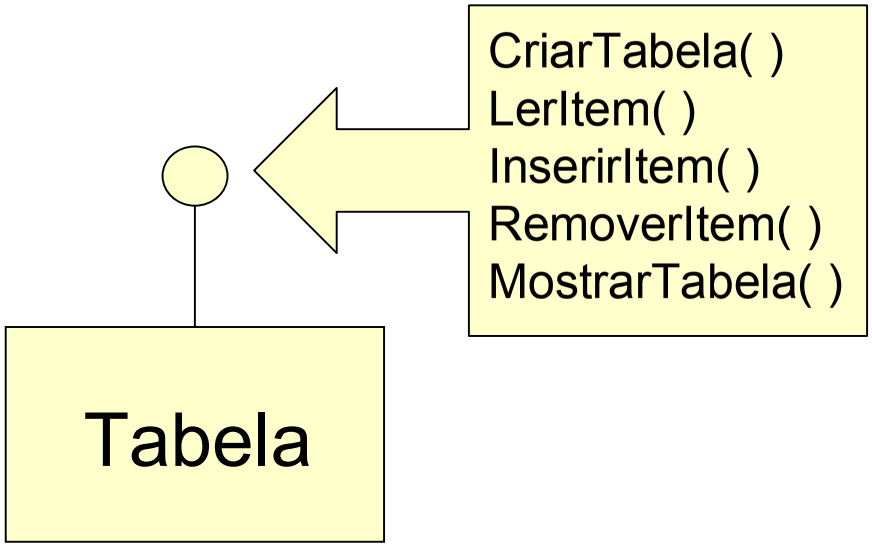


Componentes de testes (3)

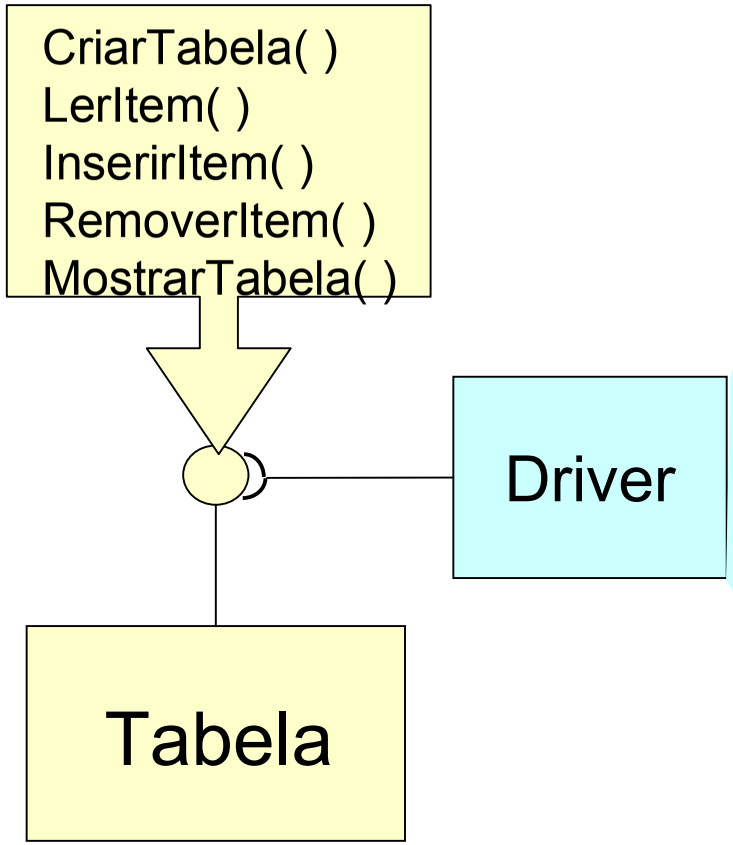
- Devem ser mais simples e mais rápidos de desenvolver do que as unidades substituídas
- Grau de facilidade ou dificuldade de construí-los depende da qualidade do projeto:

acoplamento ↗
coesão ↘
⇨ dificuldade ↗

Exemplo


CriarTabela( )
LerItem( )
InserirItem( )
RemoverItem( )
MostrarTabela( )
Tabela

Exemplo - Driver




```
type TabInt = array [ 1 .. N, 1 .. M ] of
     integer;
...
var Tabela: TabInt,
       x: integer;
...
criaTab ;
leItem ( x );
insereItem (x );
mostraTab ;
....
```

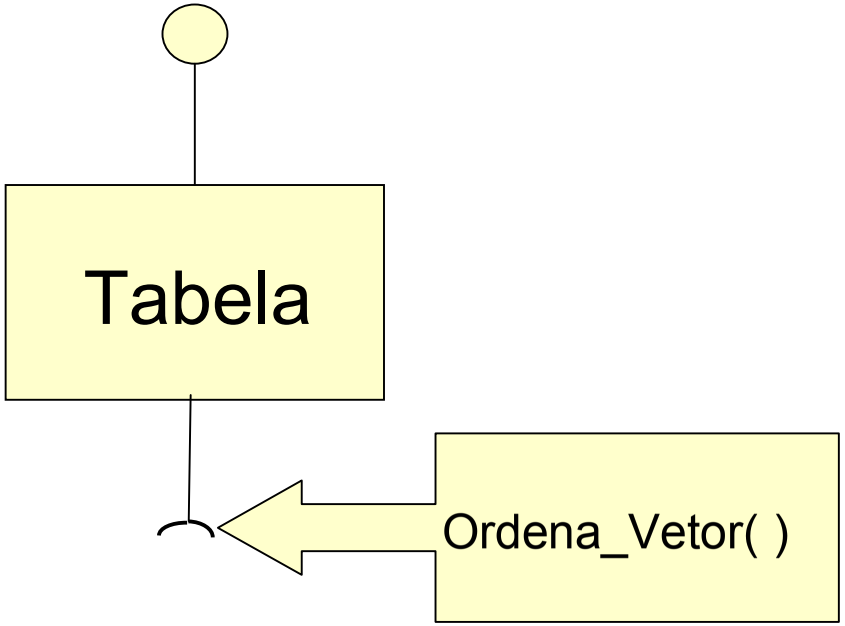
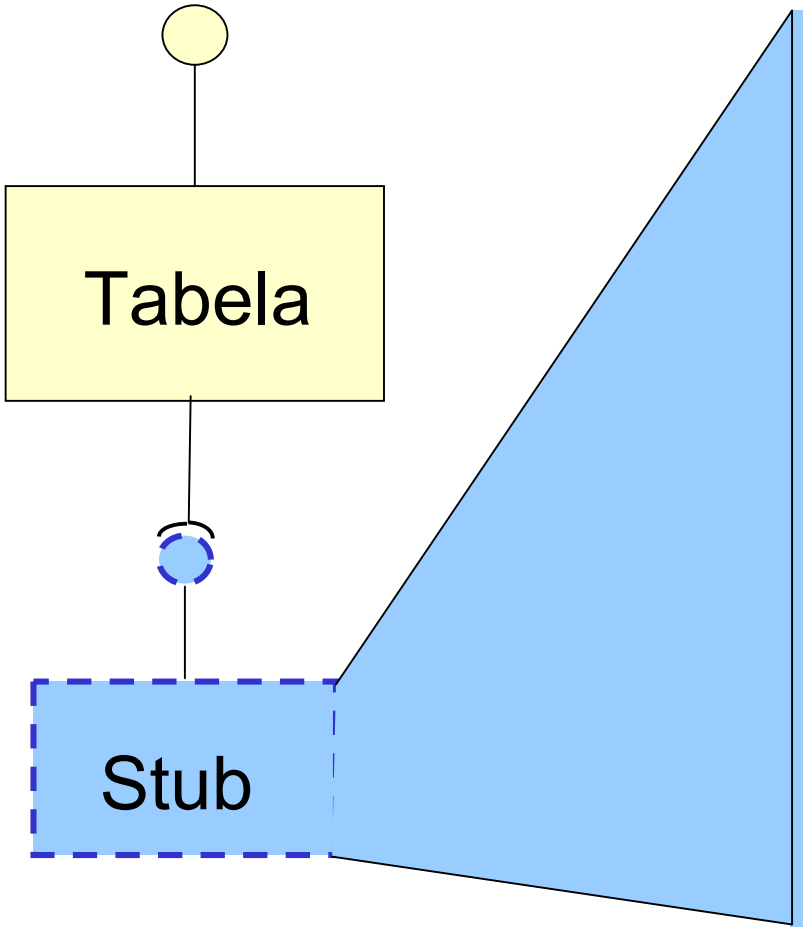


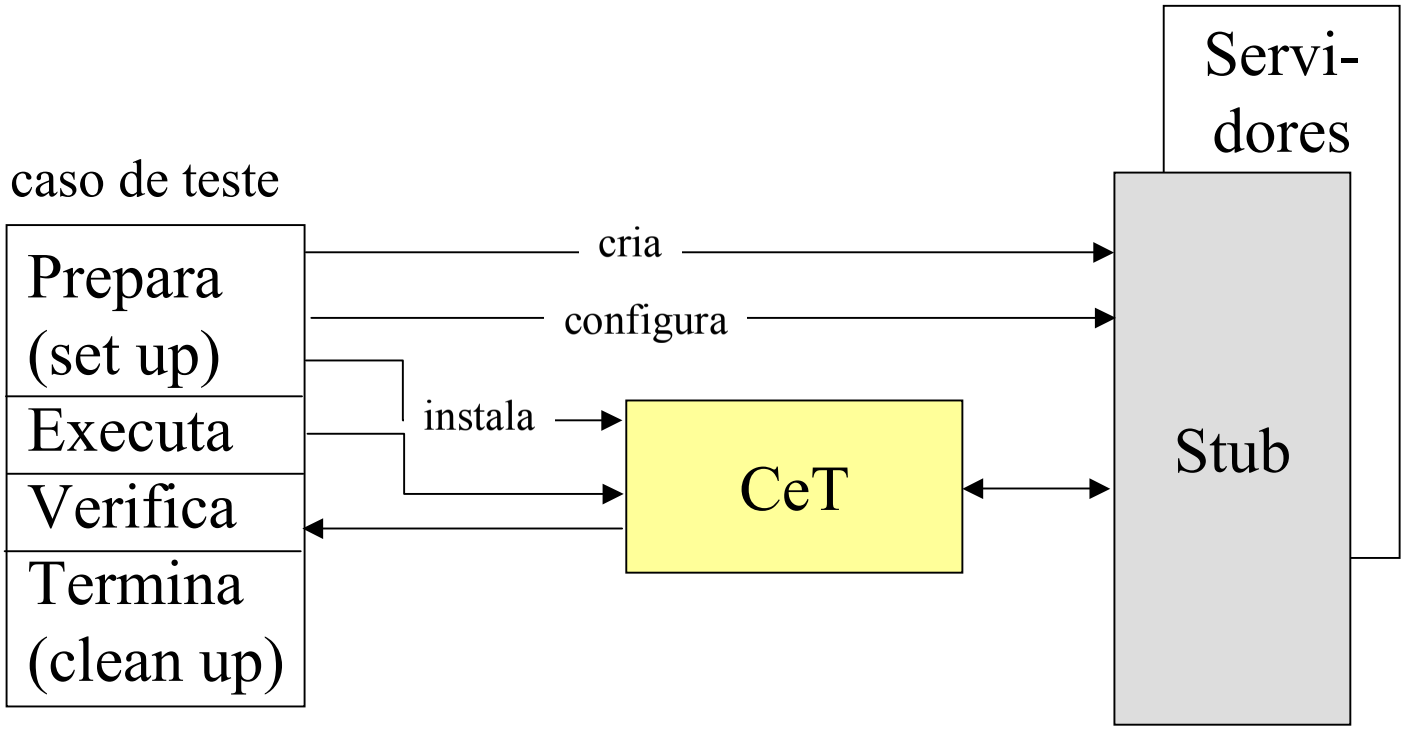
Tabela
Ordena_Vetor( )

Exemplo: stub




```
type VetorInt = array [1 .. N] of
    integer;
...
procedure Ordena_Vetor (a : VetorInt);
...
begin
   write ("Valores fornecidos");
    for i := 1 to N do write (a [ i ] );
    write ("Forneça os valores
    ordenados");
    for i := 1 to N do read (a [ i ] );
end;
```

Estrutura de testes (xUnit)



Fases da execução de um caso de teste

- **Preparação** (*set up*):
  - Cria o que for necessário, configurando os stubs de acordo para que o caso de teste execute conforme o esperado.
- **Execução**:
  - Interage com o CeT, aplicando os testes gerados e observando os resultados obtidos.
- **Verificação**:
  - Compara os resultados obtidos com os esperados.
- **Término** (*clean up* ou *tear down*):
  - Termina a execução do CeT e deixa o ambiente de execução de testes no mesmo estado em que estava antes da realização do caso de teste.

Estrutura de um caso de teste (2)

- **Preparação** (*set up*):
  - Cria o que for necessário, configurando os stubs de acordo para que o caso de teste execute conforme o esperado.
- **Execução**:
  - Interage com o CeT, aplicando os testes gerados e observando os resultados obtidos.
- **Verificação**:
  - Compara os resultados obtidos com os esperados.
- **Término** (*clean up* ou *tear down*):
  - Termina a execução do CeT e deixa o ambiente de execução de testes no mesmo estado em que estava antes da realização do caso de teste.

Execução de um caso de teste

- Aplica os dados de entrada criados
- Dados de entrada podem ser fornecidos:
  - Via teclado pelo usuário
  - Por outro sistema ou dispositivo
  - Via arquivos ou banco de dados
- Não esquecer também de outras fontes de entrada:
  - estado do sistema
  - ambiente de execução

Verificação de um caso de teste

- Determina se a saída observada = saída esperada (especificada)
- Saídas podem ser produzidas de diferentes formas:
  - Exibição na tela
  - Dados armazenados em arquivos ou bancos de dados
  - Envio para outros sistemas (via rede ou outro meio de comunicação)
  - Alteração do estado do sistema ou do ambiente de execução

☞ Não esqueça que

  - Colapso (crash), aborto de execução, blocagem (*deadlock*, *livelock*), exceções, ...
    também são possíveis saídas observadas!

Mock Objects

- Criados pela comunidade XP (em 2000)
  - ***Tim Mackinnon, Steve Freeman,*** Philip Craig. "*Endo-Testing: Unit Testing with Mock Objects*" (www.cs.ualberta.ca/~hoover/cmput401/XP-Notes/xp-conf/Papers/4_4_MacKinnon.pdf), apresentada no evento XP2000.(disponível emt www.mockobjects.com).
- Objetivo:
  - Sistematizar a geração de stubs
  - Desenvolver uma infra-estrutura para criação de mocks e incorporação dos mesmos aos Testes de Unidade.

Bibliotecas

- *Mock Objects* (ou *mocks*) servem para emular ou instrumentar o contexto (serviços requeridos) de objetos da CeT.
- Devem ser simples de implementar e não duplicar a implementação do código real.
- Bibliotecas de mocks podem ser usadas para criar stubs: existem várias APIs para esse fim:
  - MockObjects (www.mockobjects.com)
  - EasyMock (www.easymock.com)
  - MockMaker (www.mockmaker.org)
  - djUnit (http://works.dgic.co.jp/djunit/)
  - ...

Algumas razões para criar mocks

- Adiar decisão sobre a plataforma a ser usada
  - Esta é uma outra diferença entre mocks e stubs → poder criar uma classe que tenha o comportamento esperado, sem se comprometer com nenhuma plataforma específica. Ex.: para testar acesso a BD, cria-se um mock com a funcionalidade mínima que se espera do BD, sem precisar usar um BD específico.
- Lidar com objetos difíceis de inicializar na fase de preparação (set up)
  - Testes de unidade que dependam de um estado do sistema que é difícil de preparar, especialmente quando ainda não se tem o resto do sistema, podem usar mocks. O mock emula o estado de sistema, sem a complexidade do estado real. Dessa forma, o mock poderia ser utilizado por vários casos de teste que necessitem que o sistema esteja neste estado.
- Testar um objeto em condições difíceis de serem reproduzidas
  - Por exemplo, para os testes em presença de falhas do servidor: o mock pode implementar um proxy do servidor, que apresente um defeito pré-estabelecido quando for usado em determinados casos de teste.

Mocks x stubs

- Mocks são voltados para testes classes. Stubs, em princípio, podem ser usados em qqr linguagem (OO ou não).
- Segundo Martin Fowler, mocks e stubs não são sinônimos:
  - Mocks podem servir para colocar o objeto da CeT no estado desejado para os testes.
  - Um stub é uma implementação alternativa da interface do objeto substituído.
  - Um stub é mais passivo, geralmente retornando dados pré-estabelecidos pelos casos de teste para a CeT.
  - **Mocks podem verificar se o servidor foi chamado adequadamente** ⇒ contêm verificação embutida (assertivas)

Exemplo: classe em teste e uma servidora

```
classe ClasseEmTeste
   Servidora serv;
   metodo( )
     serv.executa( )
   end
end

classe Servidora
   executa( )
      # código complexo
end
```

http://www.floehopper.org/articles/2006/09/11/the-difference-between-mocks-and-stubs

Exemplo de stub: pseudo-código

```
classe ClasseDeTeste implementa Test::Unit::TestCase
   classe ServidoraStub
   executa( )
      retorna X
   end
end
// exemplo_uso_Stub
   ServidoraStub servidora
   classeTeste = ClasseEmTeste.new(servidora)
   assert_equal X, classeTeste.metodo
end
end
```

http://www.floehopper.org/articles/2006/09/11/the-difference-between-mocks-and-stubs

Exemplo de mock: pseudo-código

```
classe ClasseDeTeste implementa Test::Unit::TestCase
    classe ServidoraMock
      atributo: call_count
     ...
          call_count = 0
  // métodos
    execute( )
          call_count +=1 // conta nº de chamadas ao método
          end
    get_call_count ( )
       ...
 end
    // exemplo_uso_Mock
  servidora = ServidoraMock.new
  classeTeste = ClasseEmTeste.new(servidora)
 // verifica nº de chamadas ao método servidor
  assert_equal 1, servidora.get_call_count
 end
end
```

http://www.floehopper.org/articles/2006/09/11/the-difference-between-mocks-and-stubs

Outro exemplo : o mock

```
// Usado no teste do método: canUserLogin( User, String ) , para substituir
// o método validatePassword, chamado pelo método em teste.
public class MockUser implements User {
   ...
// Prepara o que retornar quando validatePassword for chamado
public void setValidatePasswordResult( boolean result ) {
    expectedCalls++;
   this.returnResult = result; }

// Implementação do mock de  validatePassword
public boolean validatePassword( String password ) {
   actualCalls++;
   return returnResult; }

public boolean verify() {
   return expectedCalls == actualCalls; }
... }
```

Interface da classe substituída

Determina nº esperado de chamadas ao método substituído

Conta chamadas ao método substituído

Verifica se chamadas de acordo com o esperado

Uso do mock: o caso de teste

```
// Caso de teste usando o MockUser criado anteriormente
public void testCanUserLogin() {
    MockUser user = new MockUser();
    user.setValidatePasswordResult( true );                    preparação

// usa objeto em teste já criado: ot
     boolean result = ot.canUserLogin( user, "foobar" );       execução

     assertTrue("Expected to validate user " + "password \"foobar\"", result );
     assertTrue("MockUser not used as expected", user.verify());
                                                               verificação
}
```

Padrões

- G. Meszaros definiu diversos padrões de projeto para mocks.
- S.Gorst definiu vários **idiomas** (padrões de código) para serem usados em stubs. Entre eles:
  - *Responder*:
    - Usado para fornecer entradas válidas para o CeT.
  - *Saboteur*:
    - Usados para fornecer entradas inválidas ou lançar exceções para o CeT.

Responder

- **Problema**: como fazer para que um objeto da CeT receba valores esperados de um servidor durante os testes.
- **Solução**: uso de uma classe que pode ser configurada para responder com um objeto *Responder* a cada chamada de método.

Responder

```
import java.util.ArrayList;
import java.util.List;
 public class ResponderCommunicator
     implements Communicator {
     private List _responses = new ArrayList( );
    public void open( ) throws
     CommunicationException { }

    public void close( ) throws
     CommunicationException { }

  public String communicate(String message)
     throws CommunicationException {
     if ( !_responses.isEmpty( ) ) return
     (String)_responses.remove(0);
     throw new CommunicationException("use
     setResponse to define responses"); }

 public void setResponse(String response) {
     _responses.add(response); } }
```

Implementa mesma interface da classe servidora

Contém lista de respostas a serem fornecidas a cada chamada da classe servidora

Verifica a cada chamada, se a lista de respostas = Φ → lança exceção

Método que configura a lista de respostas

Saboteur

- **Problema**: como exercitar o comportamento do CeT em situações de erro.
- **Solução**: uso de um sabotador, i.e, mock que retorne condições de erro.

Saboteur

Flags que indicam se é para retornar erro ou não

A cada chamada, verifica o *flag*. Se *true,* lança exceção.

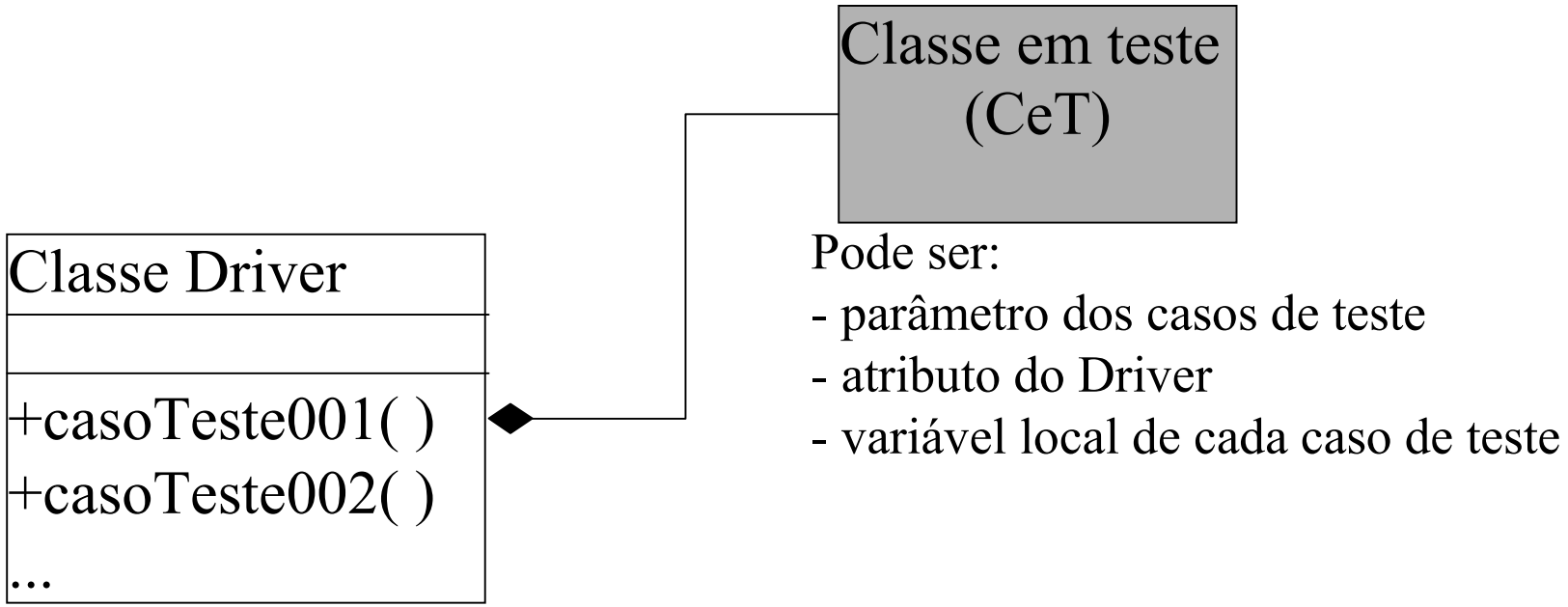
Métodos usados pelos casos de teste para inicializar os *flags*.

```
public class SaboteurCommunicator implements
    Communicator {
  private boolean _openIsSabotaged;
  private boolean _closeIsSabotaged;
  private boolean _communicateIsSabotaged;
  public void open( ) throws CommunicationException {
   if ( _openIsSabotaged ) throw new
   CommunicationException("open( ) sabotaged"); }
  public String communicate(String message) throws
   CommunicationException {
   if ( _communicateIsSabotaged ) throw new
   CommunicationException("communicate( )
   sabotaged");
  return null; }
  public void close( ) throws CommunicationException {
   if ( _closeIsSabotaged ) throw new
   CommunicationException("close( ) sabotaged"); }
  public void sabotageOpen( ) {
       _openIsSabotaged = true; }
  public void sabotageClose( ) {
     _closeIsSabotaged = true; }
  public void sabotageCommunicate( ) {
   _communicateIsSabotaged = true; } }
```

Mais padrões

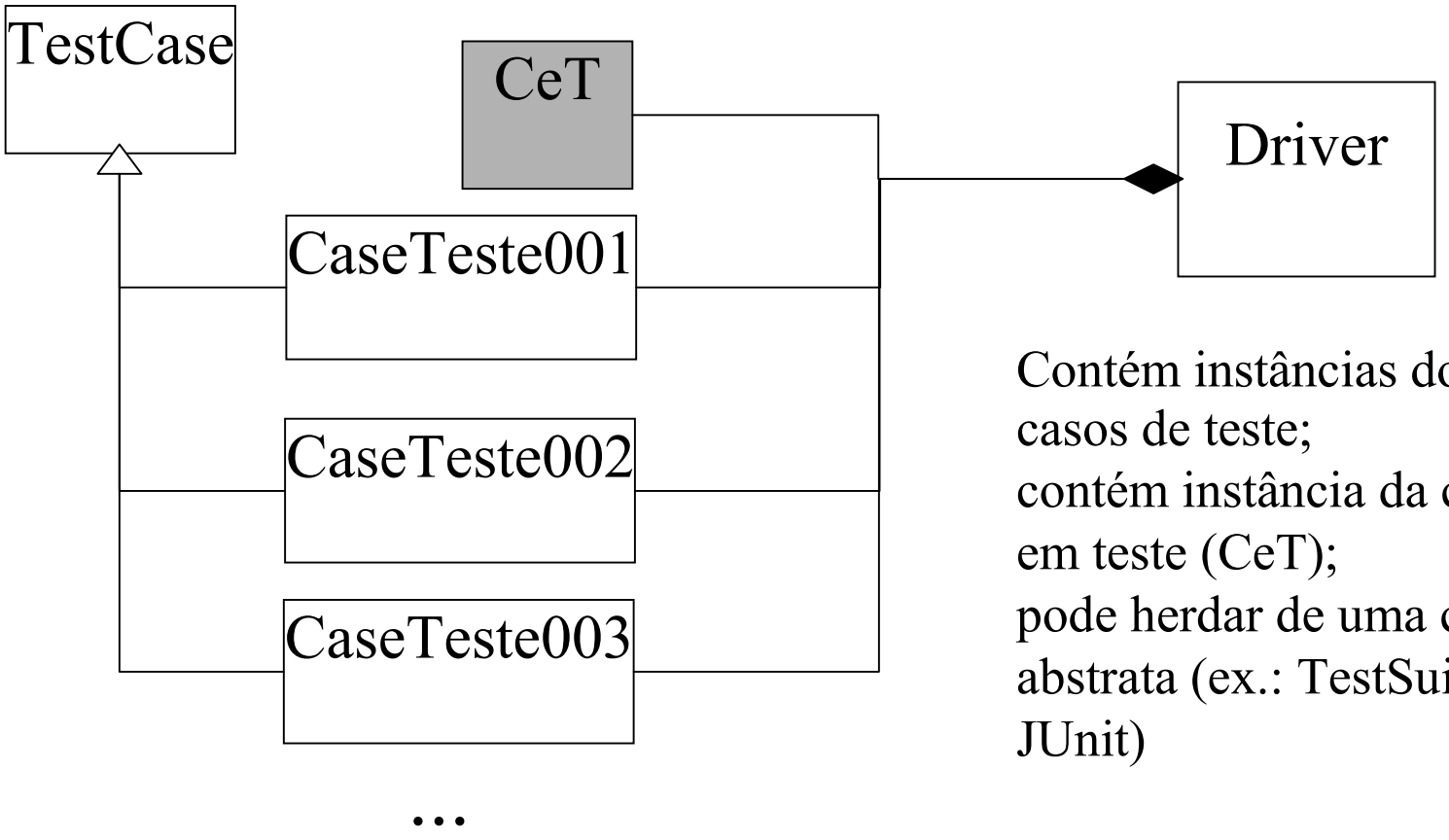
- Diversos autores propuseram padrões para a implementação de componentes de testes:
  - Drivers
  - Stubs
- Os padrões mostrados aqui são para sw OO, mas a criação de drivers e stubs também é necessária em outros paradigmas.

Driver (1)



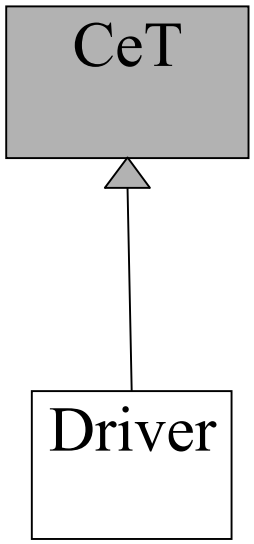
Cada caso de teste = método da classe driver
o driver pode conter um número arbitrário de métodos

Driver (2)



Contém instâncias dos casos de teste;
contém instância da classe em teste (CeT);
pode herdar de uma classe abstrata (ex.: TestSuite, do JUnit)

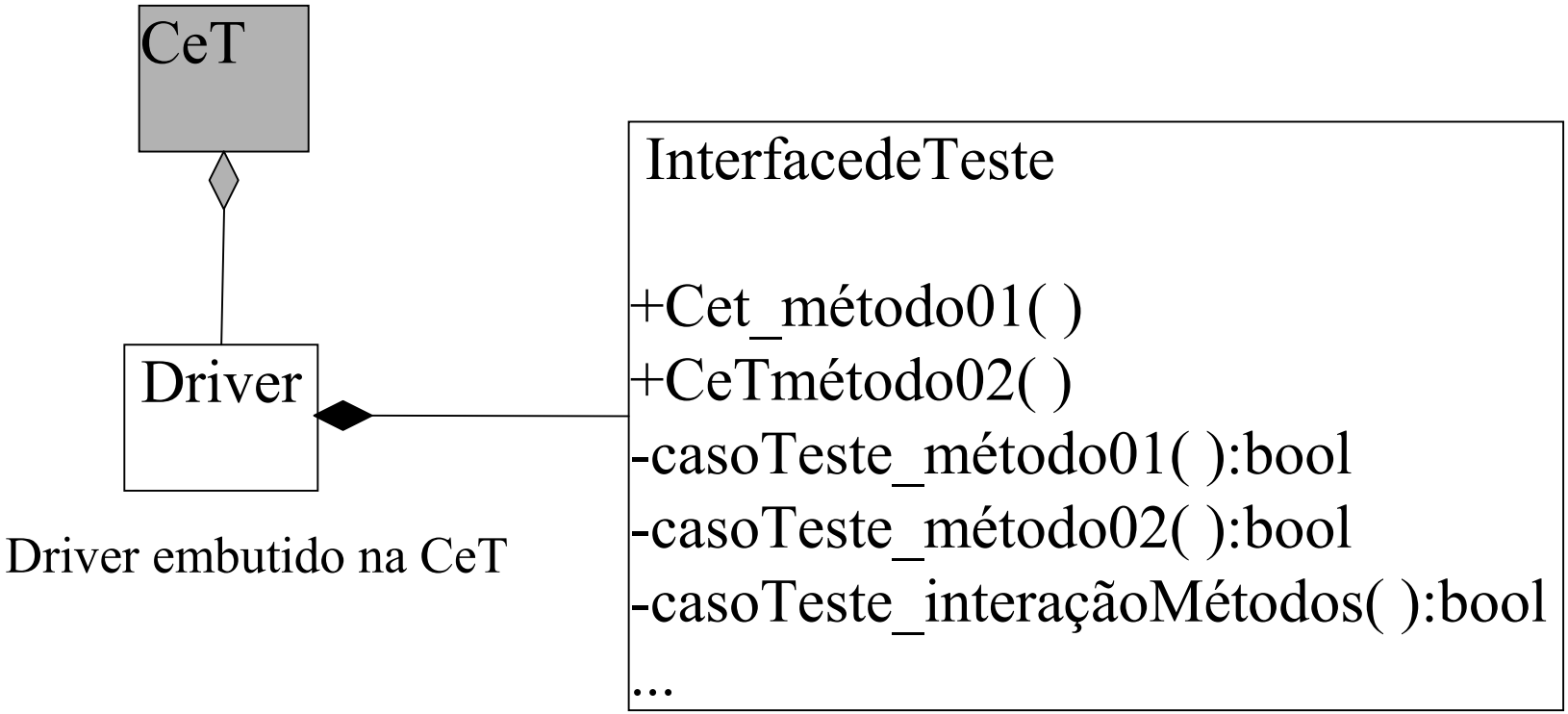
Driver (3)



Driver é subclasse da CeT

Facilita acesso a características não-privadas da CeT;
Útil para testar classes abstratas, implementando métodos virtuais

Driver (4)

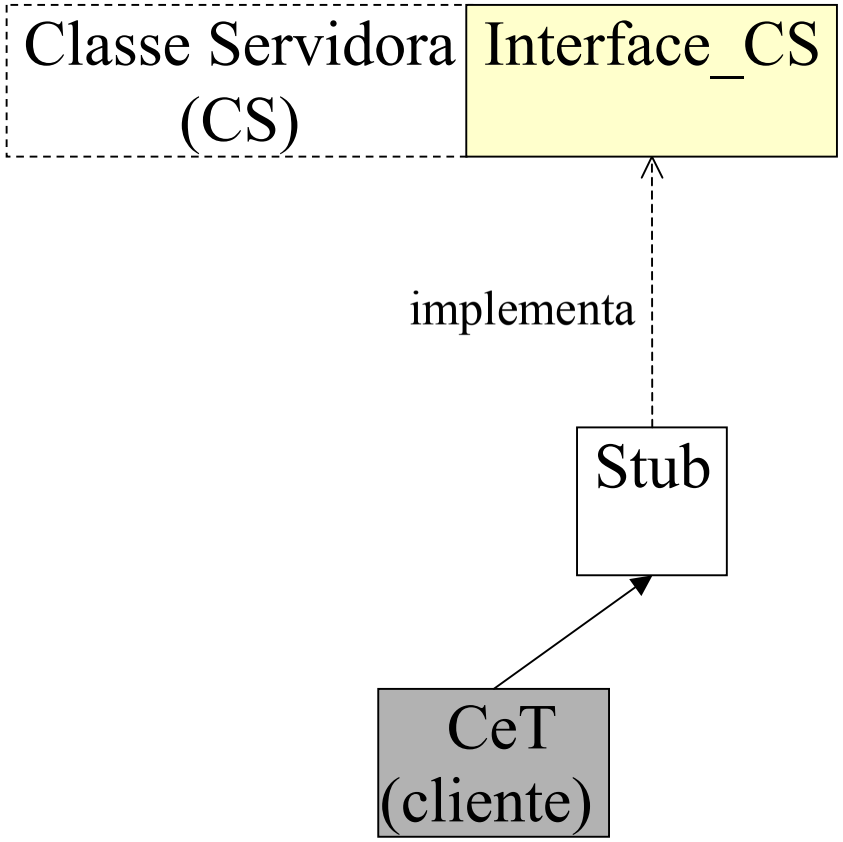


Driver embutido na CeT

Stubs

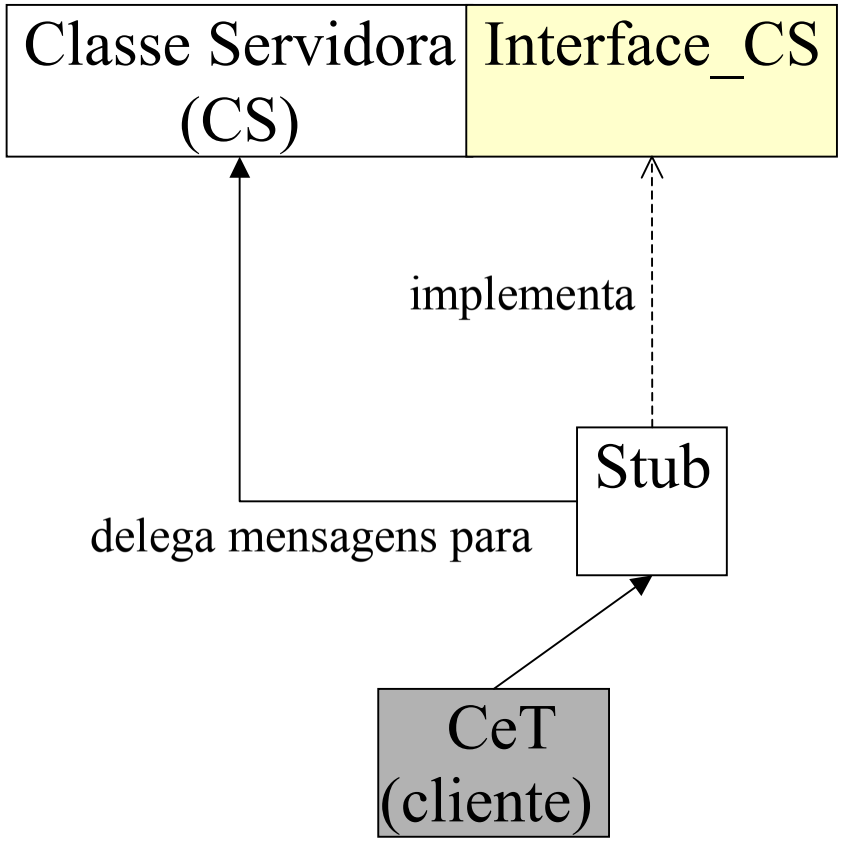
- Substituem objetos com os quais os objetos da classe em teste (CeT) interagem.
- Não contêm lógica: só métodos que podem ser usados pelos casos de teste para controlar o comportamento do objeto substituído.
- Quando usar [Massol e Husted 2003] :
  - Quando o comportamento do objeto substituído não é determinístico
  - Quando é difícil preparar o objeto sendo substituído.
  - Quando é difícil forçar um determinado comportamento (em especial, exceções) no objeto substituído.
  - Quando o objeto substituído tem uma interface gráfica.
  - Quando o objeto substituído é lento.
  - O caso de teste precisa obter informações do tipo: o método X foi chamado no objeto substituído?
  - O objeto substituído ainda não foi implementado.

Stubs (1)



Retorna sem executar nada;
Retorna valor constante;
Retorna contagem do nro de chamadas;
Retorna valor que varia conforme o nro de chamadas ao stub;
Retorna valores mínimo e máximo dos parâmetros enviados ao stub;
Retorna valores lidos de arquivo, banco de dados ou outra estrutura de dados;
Registra execução em arquivo de log;
Simula exceções;
...

Stubs (2)



Funciona como um intermediário (proxy) entre a CeT e a CS.
Delega mensagens da CeT para a CS.
Serve para testar tanto o cliente (simula ou altera respostas do servidor para o cliente) quanto o servidor (simula ou altera mensagens do cliente para o servidor).

Executando testes com JUnit

- Classe fundamental para criação de teste de unidade:
  - ***TestCase*** → parte da biblioteca junit.jar (necessária no classpath).
  - Convenção de nomes:
    - Para a classe de teste: **<nome da CeT>Test.java**
    - Para os métodos da classe de teste: **test<nome do método em teste>**
- TestSuite:
  - Coleção de TestCases
  - Nome default (Eclipse): **AllTests.java**
- *Fixture*:
  - Contexto necessário para os testes (ex.: stubs)

Executando testes com JUnit

- Passos:
  1. Crie os casos de teste para a classe
     - Adequado para testar cada método isoladamente
  2. Crie uma subclasse da classe ***TestCase***.
  3. Adicione variáveis de instância (atributos) para serem utilizados nos testes (correspondem à CeT).
  4. Crie um construtor que aceita uma cadeia de caracteres como parâmetro e passe-o para a ***superclasse***.
  5. Reescreva o método *setUp()*.
  6. Reescreva o método *tearDown()*.

```
import junit.framework.TestCase;
import Calculadora;

public class CalculadoraTest extends TestCase {

  private Calculadora calc_div;
  private Calculadora calc_mult;

  public TestCalc(String arg0) {
    super(arg0);

protected void setup(){
      calc_div = new Calculadora();
      calc_mult = new Calculadora ();
  }

   protected void teardown(){
      calc_div = null;
      calc_mult = null;
  }
...
```

Cria subclasse de TestCase

Declara objetos da CeT como atributos

Cria construtor especial

Prepara configuração de testes (*fixture*): cria objetos em teste

Fecha configuração de testes

Criação de casos de teste

- Crie um método para cada caso de teste
- Incluir o método *TestRunner.run( )* no *main* para invocar os métodos de teste
- Utilize assertivas (*Assert*) para comparar os resultados obtidos com os esperados (oráculo)

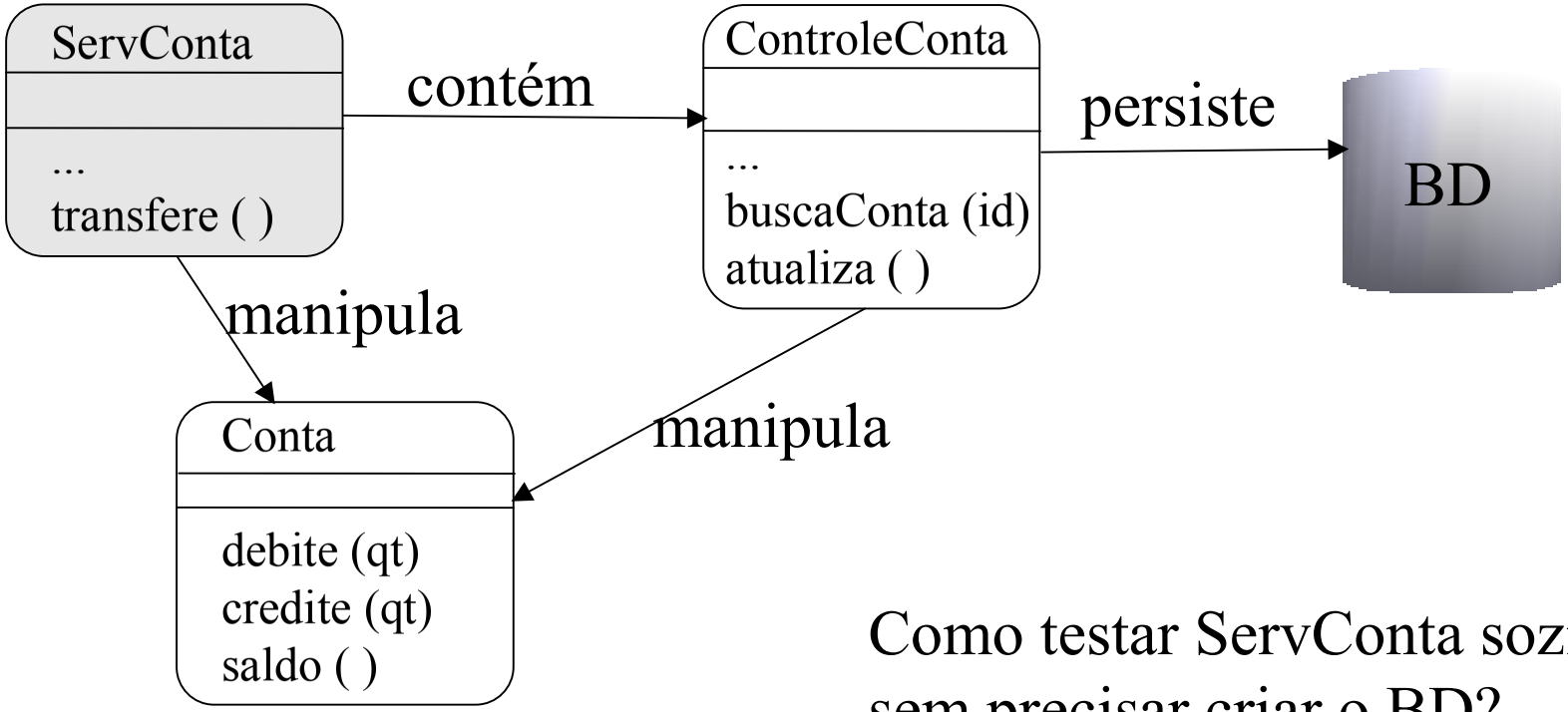
Cria casos de teste

```
...

public static void main(String[] args) {
junit.textui.TestRunner.run(CalculadoraTest.class);

public void testDivisao() {
        float resultado;

        resultado = calc_div.divisao(1.0/5.0);
        assertEquals(resultado, 0.2);

        }

        ...
}
```

chama run( ) para executar os testes

Compara com resultado esperado

Mais um exemplo



Como testar ServConta sozinha, sem precisar criar o BD?
==> uso de mock

[Massol e Husted 2003]

Classe em teste

```
public class ServConta
{
    private ControleConta cConta;
    public void setControleConta (ControleConta
     controlador)
      {  this.cConta = controlador; }
   public void transfere(String de, String para, long
     qtia)
     { Conta deConta = this.cConta.buscaConta (de);
     Conta paraConta = this.cConta.buscaConta
     (para);
     deConta.debite(qtia);
     paraConta.credite(qtia);
    this.cConta.atualiza (deConta);
    this.cConta.atualiza(paraConta);  }
}
```

Mock que implementa a interface de ControleConta

```
import  java.util.Hashtable;
public class MockControleConta
     implements ControleConta
{
    // estrutura que "simula" o BD de Contas
   private Hashtable contas = new Hastable ( );

   //define o que buscaConta deve retornar
   public void criaConta (String cliente, Conta cc)
   {     this.contas.put (cliente, cc);          }

   public Conta buscaConta (String cliente)
   {   return (Conta)     this.contas.get (cliente);  }

    // não retorna nada, como se término normal
   public void atualiza (Conta cc)
   {        // não faz nada        }
}
```

```
import junit.framewrok.TestCase;
public class TestServConta extends TestCase   ⇦ Caso de teste
{
   public void testTransfereOK ( )
   {
     MockControleConta mockCConta = new MockControleConta (
    );
     Conta deCliente = new Conta (“MarcusValério”, 300000000);
     Conta paraCliente = new Conta (“Eliane”, -200);
      mockCConta.criaConta (“1”, deCliente);
      mockCConta.criaConta (“2”, paraCliente);
      ServConta servTransfer = new ServConta ( );
      servTransfer.setControleConta (mockCConta);
     servTransfer.transfere (“1”, “2”, 500);
     assertEquals (300000000-500, deCliente.saldo ( ));
     assertEquals (-200+500, paraCliente.saldo ( ));
   }
}
```

Prepara

Cria instâncias de Conta

Inicializa tabela hash do mock

Instancia classe em teste e estabelece objeto mock como servidor

Aplica o teste

Verifica o resultado

Principais pontos aprendidos