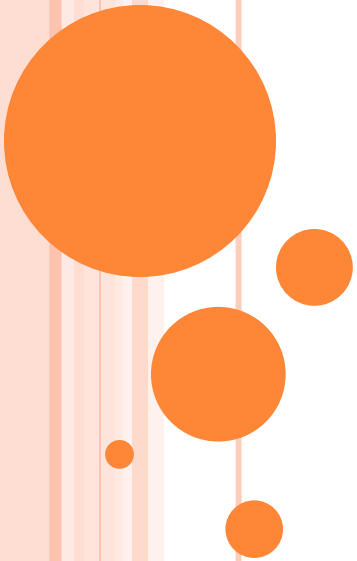


Stubs e Mocks

Paulyne Jucá (paulyne@ufc.br)

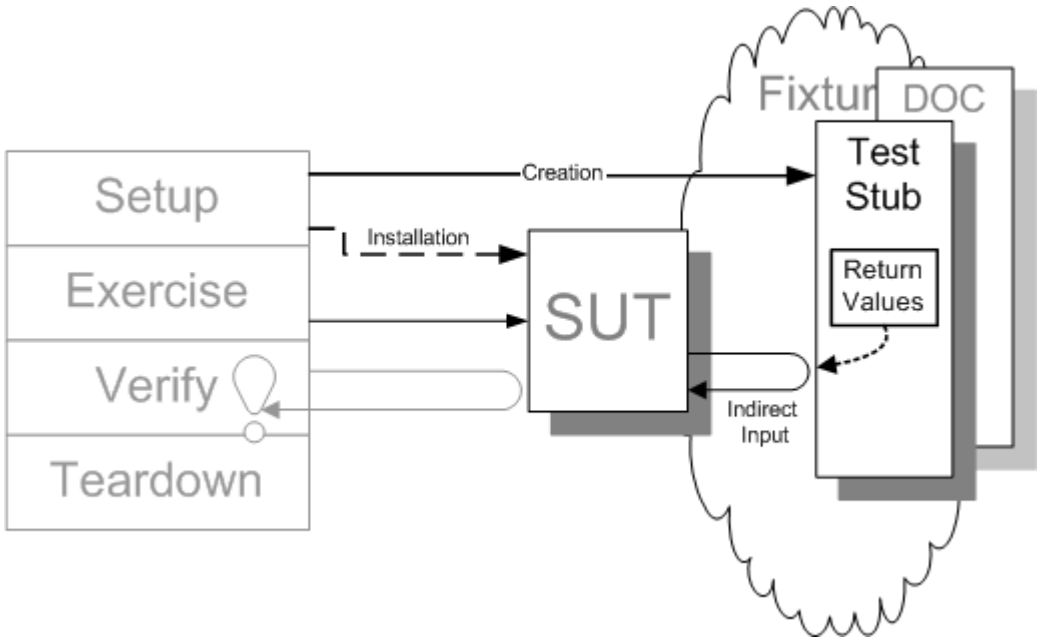
Para que stubs e mocks?

- Para isolar dependências
- Permitir trabalho paralelo
- Permitir que o trabalho evolua mesmo quando partes do software ainda não foram feitas

- Ex:
  - BD inacessível
  - BD lento, atrasando o teste de outra parte do sistema

STUBs

- Um objeto criado especificamente para o teste que substitui o objeto real e alimenta o sistema sendo testado (system under test – SUT) indiretamente com os valores desejados



STUBs – outra definição

- Implementação temporária mínima de um componente usado pelo SUT com o objetivo de permitir o controle e a observação do SUT durante os testes.

- Pode ser usado em OO ou sistemas estruturados

STUBs

- Como funciona?
  - Primeiro definimos uma implementação (específica para o teste) de uma interface da qual o SUT depende. Essa implementação é configurada para responder com valores (ou exceções) que vão exercitar uma parte do SUT. O SUT vai usar a implementação feita para o teste, em vez da implementação real.
- Quando usar?
  - Necessidade de testar o funcionamento do SUT com diferentes dados de entrada indiretos
  - Injetar valores onde o SUT faz chamada a software não disponível no ambiente de teste
- Importante
  - Permite testar o **estado** da interação entre o SUT e o software que irá ser desenvolvido (stub)

Exemplo STUB - estruturado

- int verificaAmaiorqueB(int a, int b) {
  - // não faz teste nenhum aqui, só retorna um dos
  - // resultados possíveis ou um valor inválido
  - return 0;
- }

Exemplo STUB - OO

- String getUserFromTable(int ID) {
  - User user = new User (ID, “Paulyne”);
  - return user;
- }
- Ou
- String getUserFromTable(int ID) {
  - return null;
- }
- Ou
- String getUserFromTable(int ID) {
  - throw new Exception(“Usuário não encontrado”);
- }

Usando um STUB para teste unitário

```
class OrderStateTester {
    public void testOrderSendMailIfUnfilled() {
        Order order = new Order(TALISKER, 51);
        order.fill(new Customer("eu@aqui.com"));
        asserEquals("eu@aqui.com", order.getEmail());
    }
}
```

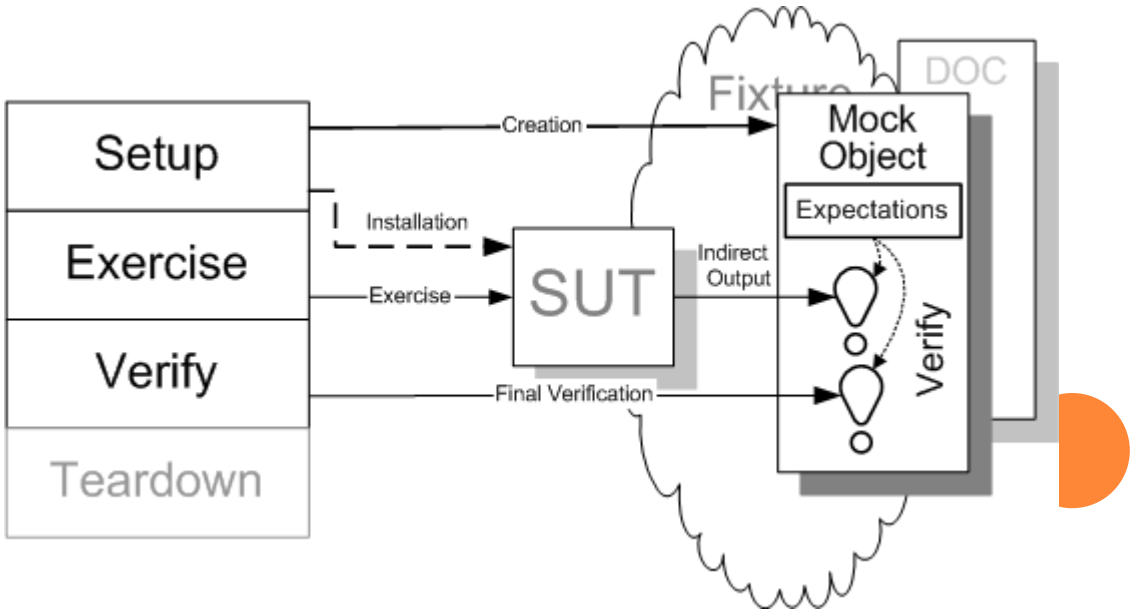
- Tanto Order, quanto Customer só tem as informações mínimas para realizar o teste.
- O resultado final é o teste para saber se o email foi enviado. Apenas o estado final interessa.

Dá para fazer usando framework? Stub com JMock

```
public void testInvoice_addLineItem_ECS() {
    final int QUANTITY = 1;
    Product product = new Product(getUniqueNumberAsString(),
        getUniqueNumber());
    Mock customerStub = mock(ICustomer.class);
    customerStub.stubs().method("getZone").will(returnValue(ZONE
        _3));
    Invoice inv = new Invoice((ICustomer)customerStub.proxy());
    // Exercise
    inv.addItemQuantity(product, QUANTITY);
    // Verify
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    assertLineItemsEqual("", expItem, actual);
}
```

MOCKs

- Como funciona?
  - Substitui um objeto que o SUT depende por um objeto específico de teste que verifica o comportamento desse objeto de teste na interação com o SUT
  - servem para emular ou instrumentar o contexto (serviços requeridos) de objetos do SUT
  - Devem ser simples de usar e não devem replicar a implementação do código real



MOCKS

- Quando usar?
  - Mock serve como ponto de observação quando precisamos verificar o comportamento de forma a evitar que problemas causado pela impossibilidade de observar efeitos colaterais de métodos invocados no SUT.
  - Para usar o mock, precisamos prever o valor de todos ou da maioria dos argumentos do método chamado antes de exercitar o SUT. Não devemos usar o mock se uma assertiva falsa não puder ser reportada para o executor do teste (ex. quando o SUT está rodando em um container que captura e come todas as exceções levantadas). Nesse caso, é mais indicado usar um Spy.

MOCKs

- Quando usar?
  - Mock (especialmente aqueles usados com o apoio de ferramentas dinâmicas) normalmente usa o método equals dos vários objetos sendo comparados. Se seu entendimento de igualdade é diferente do entendimento do SUT, não podemos usar mock ou teremos que adicionar um método equals onde não é necessário. Essa manobra é chamada de “equality pollution”. Algumas implementações de mock evitam esse problema permitindo a especificação de um comparator a ser usado nas assertivas.
  - Mocks podem ser “strict” ou ‘lenient’ (algumas vezes chamados de “nice”). Um mock strict falha se as chamadas são recebidas em uma ordem diferente da especificada quando o mock foi programado. Um mock lenient tolera chamadas fora de ordem.

- Importante
  - O teste contém verificações embutidas
  - Emula o comportamento

MOCKs – PORQUE USAR?

- 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 o 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 – FASES

- Construção
  - Construir o objeto mock para substituir a dependência. Dependendo da ferramenta usada, podemos também construir uma classe para esse objeto mock, usar um gerador de código para ela ou usar objetos gerados dinamicamente.
- Configurar com valores esperados
  - Configurar os métodos (e seus parâmetros), bem como o valor de retorno. Essa fase acontece antes da instalação do mock
- Instalação
  - O mock é colocado junto do SUT para permitir seu uso. Um método comum de tratar dependências é a injeção de dependência.

MOCKs – FASES

- Uso
  - Quando o SUT chama um método do mock, ele compara a chamada do método (e argumentos) com sua definição. Se o método não foi definido ou os argumentos estão errados, a assertiva falha o teste imediatamente. Se a chamada existe, mas está fora de sequência, um mock strict falha e um lenient prossegue. Chamadas faltando são detectadas na verificação final
  - Se o método chamado tem algum parâmetro, o mock precisa dar retorno ou atualizar alguma coisa para permitir ao SUT continuar executando o cenário de teste. Valores válidos são geralmente retornados nesse caso

- Verificação final
  - A maioria da verificação de resultados acontece dentro do mock no momento que ele é chamado pelo SUT. O mock vai falhar o teste se os métodos foram chamados com argumentos errados ou se os métodos não existirem. Mas o que acontece os métodos esperados nunca forem chamados no mock? O objeto mock pode ter dificuldade em perceber que o teste terminou. Mesmo assim é preciso garantir que a verificação final aconteça. Alguns frameworks de mock encontraram um jeito de isso acontecer através da inclusão da chamada de tearDown, mas muitos frameworks ainda dependem do testador para chamar a verificação final

MOCKS

```
import  org.jmock.Mock ;
import  org.jm ock.MockObjectTestCase  ;
. . .

public  class ExampleTest extends MockObjectTestC as e
{
public void testMethod  ( )
{
ExampleTest  mockExample  = new ExampleTest  ( )  ;

/  /  Step 1 :  create mock
Mock mockOb j  = mock  ( AnyClass .class ) ;
/  /  Step 2 :  set  expectations  ( as part  of  t he test  )  .
 mockExample. expects  (once( )).method  ( " testMethod "  )
 .with( eq( "f oo" ) )
 .will (returnValue ("bar"))  ;
/  /  Step 3 :  call method  to  be tested  and  pass mock  as  argumen t
 Client client  = new Client ( )  ;
 client. method (mockOb j )  ;
/  /  Step  4 :  verify  expectations
 mockExample. verif y  ( )  ;
}
}
```

STUBs x MOCKs

- Imagine que você precisa testar um controlador de emails, mas não tem o repositório de emails pronto ainda
- Você tem nesse caso duas classe:
  - Repositorio (acesso a base de dados – não feita ainda)
  - Controlador (responsável pela solicitação de adição e remoção de novos email na base de dados – lógica de negócio está aqui. Ex.: permissão de acesso)

STUBs x MOCKs

- No primeiro teste, você deseja verificar se o funcionamento do método cadastrar está retornando a mensagem “email já cadastrado” quando você tenta adicionar um email já existente
  - Nesse teste, sua verificação é no controlador.
  - Mas você precisa que o Repositorio retorne “true” ou “false” no método existe(String email).
    - Um stub resolve esse problema!
  - Você pode criar uma classe Repositório com um único método existe nela. Não importa se o resto dos métodos esperados para a classe não estejam prontos. Você não precisa deles para esse teste

STUBs x MOCKs

- Mas você pode usar um framework para isso.

```
[Test]
public void Quando_email_existe_retorna_mensagem_email_existente(){
var repositorioEmails = new Mock<ilistadeemails>();
    repositorioEmails        .Setup(r => r.Existe(email_existente))        .Returns(true);
var controller = new EmailController(repositorioEmails.Object);
var retorno = controller.Cadastrar(email_existente);
    Assert.AreEqual("E-mail já cadastrado.", retorno.JsonDataAsString());
    }
</ilistadeemails>
```

```
[Test]
public void Quando_ocorre_erro_db_no_metodo_existe_retorna_mensagem_tente_novamente(){
var repositorioEmails = new Mock<ilistadeemails>();
    repositorioEmails        .Setup(r => r.Existe(email_novo))        .Throws(new Exception());
var controller = new EmailController(repositorioEmails.Object);
var retorno = controller.Cadastrar(email_novo);
    Assert.AreEqual("Por favor tente novamente.", retorno.JsonDataAsString());
    }
</ilistadeemails>
```

STUBs x MOCKs

- Mas imagine que você quer testar a interação entre Repositório e Controlador
  - Você deseja verificar se o controlador está chamando o repositório com valores válidos e se o repositório se comporta corretamente nessa interação
    - Mas o repositório ainda não está pronto!?
  - Nesse caso, você vai usar mock para simular o comportamento do repositório
    - Você deve configurar o mock definindo como ele deve agir para cada situação no teste e depois verificar se o comportamento foi o esperado

STUBs x MOCKs

- O teste é feito no mock

```
[Test]
public void Quando_email_nao_existe_adiciona_na_lista_de_emails(){
var repositorioEmails = new Mock<ilistadeemails>(MockBehavior.Strict);
    repositorioEmails        .Setup(r => r.Existe(email_novo))        .Returns(false);
var controller = new EmailController(repositorioEmails.Object);
    controller.Cadastrar(email_novo);
    repositorioEmails.Verify(r => r.Inserir(email_novo), Times.Once());
    }
</ilistadeemails>
```

- Aqui, o teste é para verificar se o email foi inserido no repositório uma única vez. (inserir(email_novo))

STUBs x MOCKs

- O uso de mocks também permite testes mais complexos como a adição de outra emails diferentes e a verificação de se os dois foram inseridos e se o segundo foi mesmo o último

STUBs x MOCKs

```java
@Test
public void testInvideJaneAndJohn() {
    Party party = mock(Party.class);
    Person peter = new Person("Peter");
    Person jane = new Person("Jane");
    Person john = new Person("John");

    // "listen" to invocations of Party.addGuest() method
    final Holder<Person> lastInvited = new Holder<Person>();
    doNothing().when(party).addGuest(argThat(new BaseMatcher<Person>() {

        @Override
        public boolean matches(Object arg0) {
            lastInvited.set((Person) arg0);
            return true;
        }

        @Override
        public void describeTo(Description arg0) {
        }
    }));

    // Peter invites his friends
    peter.invideToParty(jane, party);
    peter.invideToParty(john, party);

    // verify 2 guests are invited and John is the last one
    verify(party, times(2)).addGuest(any(Person.class));
    assertEquals(john, lastInvited.get());
}
```

Frameworks disponíveis

- Java: Mockito, Jmock, EasyMock
- Ruby: Mocha
- .NET: NMock, Rhino Mocks
- Perl: Test::MockObjects
- Etc...

Referências

- http://xunitpatterns.com
- http://www.agileandart.com/2011/09/20/injecao-de-dependencia-e-testes-com-dubles/
- http://media.pragprog.com/articles/may_02_mock.pdf
- http://www.betgenius.com/mockobjects.pdf
- http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0410831_06_cap_02.pdf
- www.ic.unicamp.br/~eliane/Cursos/Inf307
- http://viniciusquaiato.com/blog/diferenca-entre-mocks-e-stubs/
- http://blog.camilolopes.com.br/mocks/
- http://blog.camilolopes.com.br/criando-mocks-com-mockito/
- http://marcschwieterman.com/blog/simple-stub-creation-with-mockito-partial-mocks/