

CURSO DE TESTES AUTOMATIZADOS MOCKS

Wagner Costa

wcaquino@gmail.com

NEM TUDO É UMA MARAVILHA

- No módulo anterior, vimos que o JUnit nos ajuda a seguir com segurança, a ter mais confiança em nosso código;
- Contudo, existe a parte do processo que depende do ser humano...
 - Quem pode garantir que João testou bem o seu módulo?
 - E se o meu módulo usa o módulo de João?
- Meu cenário de testes ficou mais complexo;
 - Estes módulos podem precisar de uma infra-estrutura pesada:
 - Acessar um servidor web, um container, um banco de dados...
 - E o componente que eu quero testar nem usa esta infra-estrutura.
- Mas... e se alguns dos módulos que eu preciso instanciar ainda não foram implementados?

RESUMINDO

- Eu gostaria muito de testar meu módulo, mas:
 - Não gostaria que bugs no código que ele acessa atrapalhassem meus testes;
 - Não gostaria de correr o risco de perder o controle sobre o cenário de testes devido à dependência de muitos outros módulos;
 - Não gostaria de iniciar serviços que não são essenciais para testar uma determinada lógica de negócio;
 - Não gostaria de depender de componentes que ainda não estão prontos;
- Seria ótimo se eu pudesse isolar apenas o que eu quero testar:
 - Verificar apenas o comportamento do meu módulo;

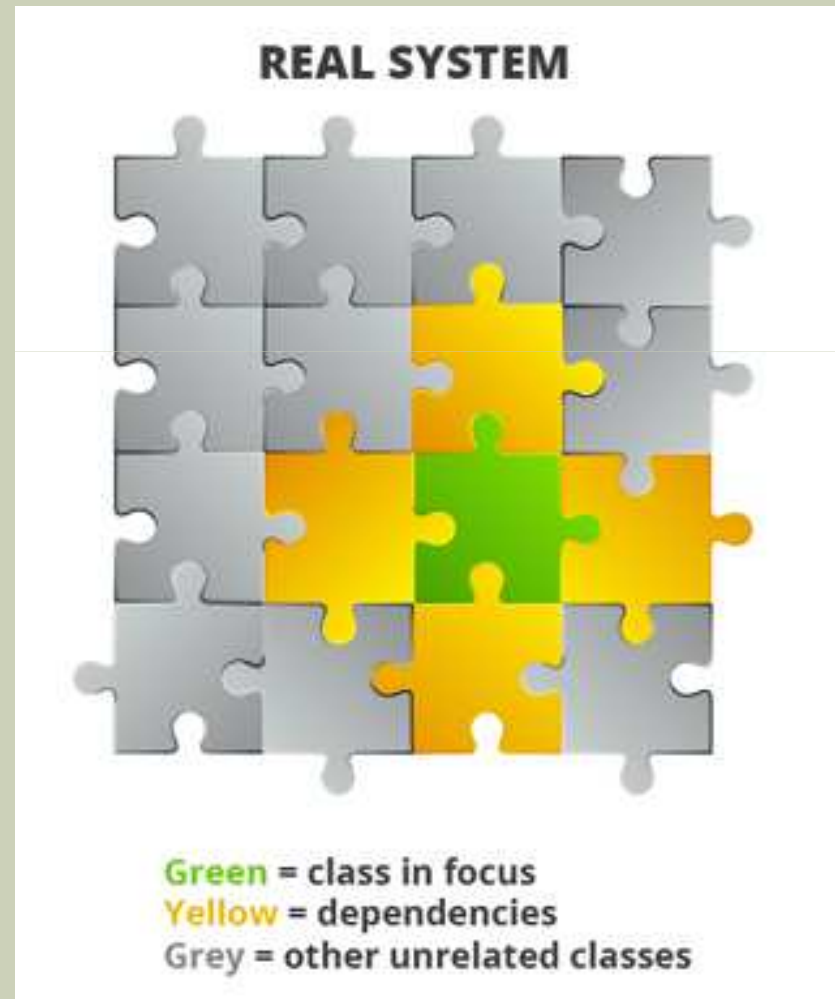
FORMALIZANDO O PROBLEMA

- Existem situações em que codificar o teste de unidade se torna uma atividade complexa;
- Às vezes, é preciso:
 - Instanciar um ambiente de difícil configuração;
 - Usar um objeto cujo comportamento é difícil de reproduzir;
 - Usar um objeto cujo desempenho é ruim;
 - Interagir com uma interface com o usuário (UI – User Interface);
 - Consultar um objeto, mas os dados ainda não estão disponíveis no mesmo;
 - Usar um objeto que ainda não existe;

MOCK OBJECTS

- *Mock Objects* (ou apenas *mocks*) substituem objetos com os quais o objeto testado colabora:
 - Da mesma forma, permitem que o código a ser testado seja isolado;
- *Mocks* não implementam lógica alguma;
 - Na verdade, eles oferecem métodos que permitem configurar e verificar o comportamento esperado:
- Sendo assim, os próprios testes definem o comportamento do objeto com o qual colaboram;

CLASSE EM PRODUÇÃO



CLASSE COM MOCKS

CLASS IN UNIT TEST



Green = class in focus
Yellow = mocks for the unit test

MOCK OBJECTS

- Resumindo...

- Um *mock* é um objeto que substitui um objeto com o qual seu teste colabora. O código do teste pode chamar métodos no *mock*, de forma que este último vai retornar resultados de acordo com o comportamento configurado no próprio teste.

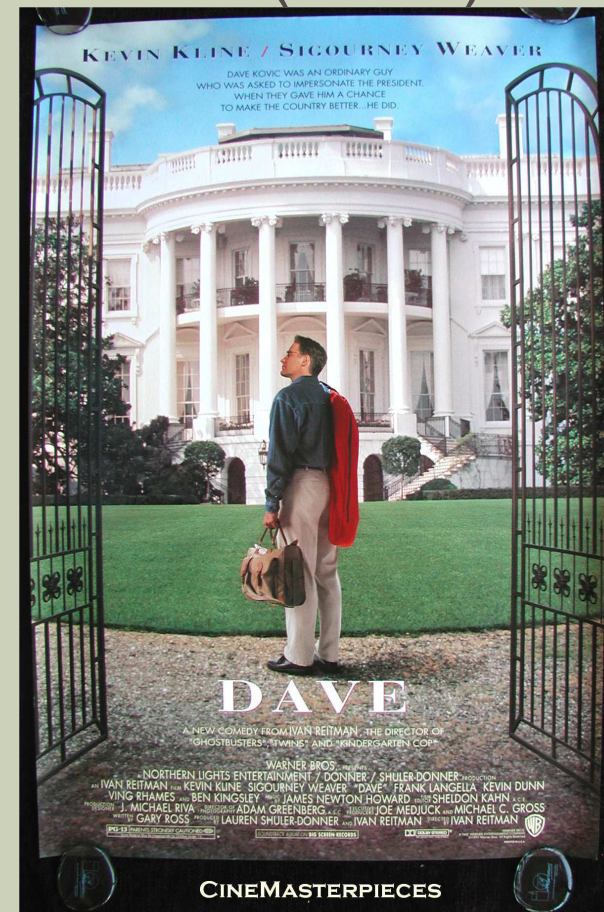
- Os principais frameworks para criar mocks em Java são: *EasyMock* e *Mockito*;

FILMES QUE UTILIZAM MOCKS

Truman Show (1998)



Dave (1993)



EASYMOCK

■ O que é?

- É uma biblioteca que fornece uma maneira fácil de usar mocks para uma determinada **interface ou classe**;
- Versão atual: 3.2; (trabalharemos com a 3.0)

■ Onde baixar?

- <http://www.easymock.org/Downloads.html>

COMO USAR EM SUA APLICAÇÃO?

- Adicione o *easymock.jar* no *classpath* de sua aplicação;
- A versão 3.0 possui a dependência para a lib *cglib-nodep-2.1_3-osgi.jar*

MÉTODO A SER TESTADO

```
public Entidade salvar(Entidade entidade) throws Exception{  
    validarCamposObrigatorios(entidade);  
    validarRegras(entidade);  
  
    //Salvando...  
    entidade = persistencia.salvar(entidade);  
  
    return entidade;  
}
```

OPERAÇÕES COM MOCKS

- Criação
- Injeção
- Gravação de Expectativas
- Execução do Teste
- Verificações

CRIANDO UM MOCK

```
EntidadeDAOInterface persistencia =  
    EasyMock.createMock(EntidadeDAOInterface.class);
```

CONFIGURANDO O COMPORTAMENTO ESPERADO

- Suponha que após criar o mock, executamos o seguinte código em nosso teste:

```
Entidade entidade = getEntidadeValida();  
persistencia.salvar(entidade);
```

- O que estamos fazendo na segunda linha de código?
 - Estamos gravando o comportamento esperado;
- Assim que é criado, um *mock* está no estado “de gravação”:
 - Neste momento, toda chamada realizada no *mock* funciona como a gravação de uma expectativa;

CONFIGURANDO O COMPORTAMENTO ESPERADO

- Mas ainda não definimos o valor de retorno que a `DesafioNegocio` vai receber quando chamá-lo;

- Podemos fazer da seguinte maneira:

```
Entidade entidadeRetornada = getEntidadeValida();  
entidadeRetornada.setId(1L);  
EasyMock.expectLastCall().andReturn(entidadeRetornada);
```

- E agora, o que acontece com o nosso código?
 - **Funciona!!!**

REPLAY & VERIFY

```
@Before
public void setUp() {
    desafioNegocio = new DesafioNegocioFinal();
    persistencia = EasyMock.createMock(DesafioPersistenciaInterface.class);
    desafioNegocio.setPersistencia(persistencia);
}

@Test
public void testSalvar() throws Exception {
    DesafioEntidade entidade = getEntidadeValida();
    DesafioEntidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);

    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.replay(persistencia); //a seguir!

    DesafioEntidade entidadePersistida = desafioNegocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());

    EasyMock.verify(persistencia); //a seguir!
}
```

REPLAY

- Como visto anteriormente, ao criar um mock, ele está no estado de gravação:
 - Neste estado, configuramos as expectativas;
- Quando executamos o método *replay(mock)*, estamos mudando o estado do *mock* do estado de gravação para o estado de replay.
- A partir deste momento, o objeto passa a se comportar como um *Mock Object*, conferindo se as chamadas estão realmente acontecendo conforme esperado.

VERIFY

- Mas... se o comportamento do mock já consiste em verificar se as chamadas estão sendo feitas da maneira correta, por que ainda precisamos de um método `verify()`?
 - Na verdade, o mock só verifica se cada chamada realizada condiz com as expectativas definidas;
 - O mock não verifica se todo o comportamento esperado está sendo executado;

TUDO COMENTADO

```
@Before
public void setUp() {
    desafioNegocio = new DesafioNegocioFinal();
    persistencia = EasyMock.createMock(DesafioPersistenciaInterface.class);
    desafioNegocio.setPersistencia(persistencia);
}

@Test
public void testSalvar() throws Exception {
    DesafioEntidade entidade = getEntidadeValida();
    DesafioEntidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);
    //Informando as chamadas que devem ser realizadas
    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.replay(persistencia);
    //Efetuando as chamadas
    DesafioEntidade entidadePersistida = desafioNegocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());
    //Checar se todas as chamadas previstas foram realizadas
    EasyMock.verify(persistencia);
}
```

QUESTÃO 1

■ Este código funciona?

```
@Test
public void testSalvar() throws Exception{
    DesafioEntidade entidade = getEntidadeValida();
    DesafioEntidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);

    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.replay(persistencia);

    DesafioEntidade entidadePersistida = desafioNegocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());

    EasyMock.verify(persistencia);
}
```

RESPOSTA

- A expectativa era de duas chamadas, mas apenas uma foi realizada.

es: 1

Failure Trace

```
java.lang.AssertionError:  
  Expectation failure on verify:  
    salvar(br.dtp.treinamento.DesafioEntidade@209f4e): expected: 2, actual: 1  
at org.easymock.internal.MocksControl.verify(MocksControl.java:111)  
at org.easymock.EasyMock.verify(EasyMock.java:1608)  
at br.dtp.treinamento.DesafioMockTest.testSalvar(DesafioMockTest.java:48)
```

QUESTÃO 2

- Este código funciona?

```
@Test
public void testSalvar() throws Exception{
    DesafioEntidade entidade = getEntidadeValida();
    DesafioEntidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);

    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
    EasyMock.replay(persistencia);

    DesafioEntidade entidadePersistida = desafioNegocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());

    //EasyMock.verify(persistencia);
}
```

QUESTÃO 3

■ E esse?

```
@Test
public void testSalvar() throws Exception{
    Entidade entidade = getEntidadeValida();
    Entidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);

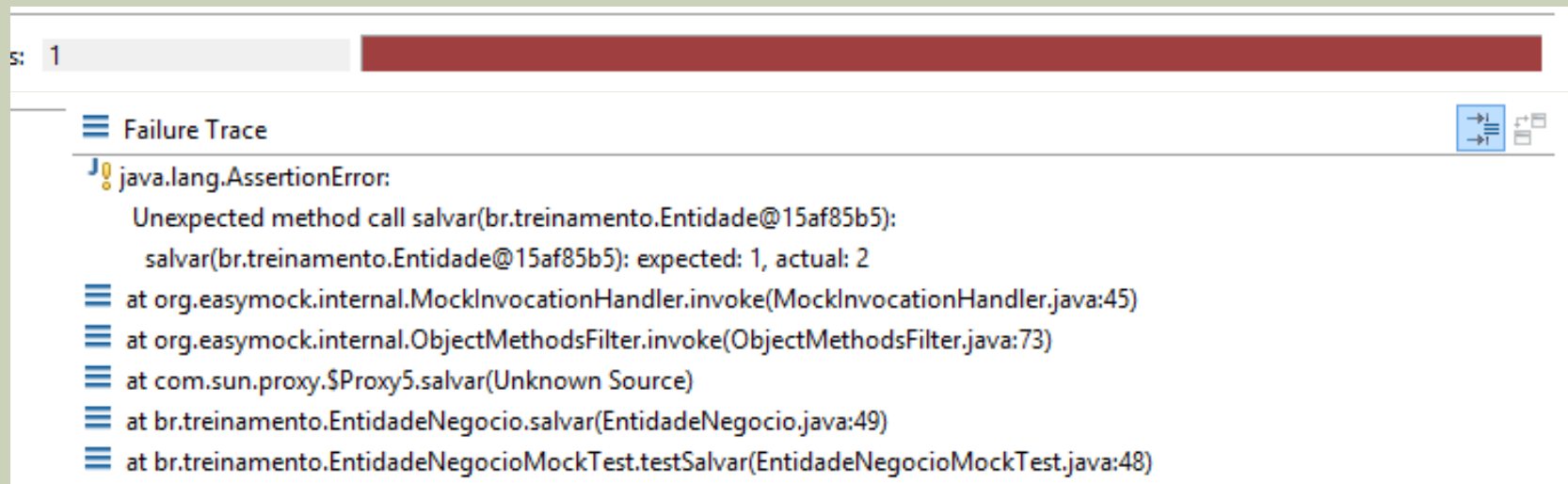
    persistencia.salvar(entidade);
    EasyMock.expectLastCall().andReturn(entidadeRetornada);

    EasyMock.replay(persistencia);

    Entidade entidadePersistida = negocio.salvar(entidade);
    entidadePersistida = negocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());
}
```


RESPOSTA

- O Mock sabia o que responder na primeira vez, mas ninguém avisou para ele que teria uma segunda chamada...



The screenshot shows a failure trace in an IDE. At the top, there is a tab labeled 's: 1' and a red progress bar. Below this, the 'Failure Trace' section is expanded, showing a Java exception: `java.lang.AssertionError: Unexpected method call salvar(br.treinamento.Entidade@15af85b5): salvar(br.treinamento.Entidade@15af85b5): expected: 1, actual: 2`. The stack trace includes the following frames:

- `at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:45)`
- `at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:73)`
- `at com.sun.proxy.$Proxy5.salvar(Unknown Source)`
- `at br.treinamento.EntidadeNegocio.salvar(EntidadeNegocio.java:49)`
- `at br.treinamento.EntidadeNegocioMockTest.testSalvar(EntidadeNegocioMockTest.java:48)`

ESPERANDO UM NÚMERO DE CHAMADAS

- Como fazemos para configurar o comportamento do mock para esperar exatamente um número n de chamadas a um método?
- Suponha que queremos salvar a mesma entidade duas vezes:
 - Como consequência, o mock será chamado duas vezes;
 - Podemos então verificar se a chamada acontece exatamente duas vezes:

```
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);  
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);
```

=

```
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);  
EasyMock.expectLastCall().times(2);
```

ESPERANDO UM NÚMERO DE CHAMADAS

- E se quisermos chamar várias vezes, mas não nos importamos em relação à quantidade?

```
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);  
EasyMock.expectLastCall().anyTimes();
```

- E se não nos importamos com a quantidade, desde que seja chamado pelo menos uma vez?

```
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);  
EasyMock.expectLastCall().atLeastOnce();
```

- E se quisermos “relaxar” um pouco o rigor na quantidade de vezes que um método é chamado?

- Podemos usar uma faixa de valores

```
EasyMock.expect(persistencia.salvar(entidade)).andReturn(entidadeRetornada);  
EasyMock.expectLastCall().times(2, 5);
```

FAZENDO DO JEITO “CERTO”

- Uma vez que estamos usando o método *expectLastCall()*, não precisamos chamar o método *expect()* na chamada.
 - Também podemos usar o *expectLastCall()* para determinar o valor de retorno:

```
persistencia.salvar(entidade);  
EasyMock.expectLastCall().andReturn(entidadeRetornada).atLeastOnce();
```

- Se não estamos usando *expectLastCall()*, temos que usar o *expect()* pra configurar o retorno;
- Métodos sem retorno (*void*) só podem ser acessados via *expectLastCall()*

DEFININDO COMPORTAMENTO DE ERRO

- Podemos definir um comportamento de erro através de exceções:
 - Para isso, basta definir uma exceção como valor de retorno

```
persistencia.salvar(entidade);  
EasyMock.expectLastCall().  
    andThrow(new Exception("Já existe uma entidade com este texto"))  
    .atLeastOnce();
```

- Checked exceptions só podem ser lançadas se estiverem declaradas no método:
 - O erro não pode ser visto em tempo de compilação;
- Unchecked exceptions podem ser lançadas por qualquer método;

DEFININDO COMPORTAMENTO VARIÁVEL

- Podemos configurar uma mesma chamada para retornar valores diferentes em sucessivas invocações:

```
persistencia.salvar(entidade);  
EasyMock.expectLastCall()  
    .andReturn(null).times(2)  
    .andReturn(entidadeRetornada).times(3)  
    .andThrow(new Exception("Já existe uma entidade com este texto")).once();
```

- Neste exemplo:
 - O método retorna null, duas vezes;
 - Retorna uma entidade, três vezes;
 - Lança uma exceção, uma vez;

EXPECTATIVAS FLEXÍVEIS COM MATCHERS

- Se quisermos definir o comportamento para a adição de produtos, não importando a quantidade, podemos usar um matcher:

```
persistencia.getById(EasyMock.anyLong());
```

- Se um método tiver mais que um parâmetro e precisar de Matcher em algum deles, terá que usar em todos.

OUTROS TIPOS DE MATCHERS

- eq(X value);
- anyBoolean();
- anyByte();
- anyChar();
- anyDouble();
- anyFloat();
- anyInt();
- anyLong();
- anyObject();
- anyShort();
- eq(X value, X delta);
- aryEq(X value);
- isNull();
- notNull();
- isA(Class clazz);
- It(X Number);
- leq(X Number);
- geq(X Number);
- gt(X Number);
- startsWith(String prefix);
- contains(String substring);
- endsWith(String suffix);
- matches(String regex);
- find(String regex);
- and(X first, X second);
- or(X first, X second);
- not(X value).

STRICT MOCKS

- Um mock normal não verifica a ordem em que as chamadas são realizadas:
 - Ele se contenta apenas em verificar se as expectativas estão corretas; e
 - Se todas as expectativas acontecem (quando usamos `verify()`);
- No entanto, podemos criar um mock que verifica a ordem exata em que as chamadas acontecem:
 - Se houver violação de ordem, o teste quebra!

```
persistencia =  
    EasyMock.createStrictMock(DesafioPersistenciaInterface.class);
```

NICE MOCKS

- Ao contrário dos StrictMocks, os NiceMocks são muito mais “legais”
 - Por default, os nice não se importam com a ordem da chamadas
 - Os nicemocks também não reclamam de chamadas inesperadas (não gravadas), quando estas ocorrem, o mock retorna um valor default, dependendo do tipo de retorno
 - 0 (zero) para numéricos primitivos
 - False para booleanos
 - Null para Objetos

```
persistencia =  
    EasyMock.createNiceMock(DesafioPersistenciaInterface.class);
```

DEFININDO ORDEM DINAMICAMENTE

- Nem sempre queremos usar um mock estrito:
 - Às vezes queremos checar a ordem apenas durante algum momento;
- Para isso, podemos ligar ou desligar o comportamento de checar a ordem das mensagens:

```
EasyMock.checkOrder(persistencia, true);  
//...  
EasyMock.checkOrder(persistencia, false);
```

RESET

- Sempre que quisermos reiniciar um mock, ou seja, jogar fora as expectativas que já foram geradas para o mesmo, podemos chamar o método reset():

```
EasyMock.reset(persistencia);
```

- Quando chamamos reset(), o mock volta para o estado de gravação;
- É útil para reusar um mock;

TIPOS DE RESET

- **Reset**
- **resetToDefault**
- **resetToStrict**
- **resetToNice**

REFACTORING

```
@BeforeClass
public static void setUp() {
    desafioNegocio = new DesafioNegocioFinal();
    persistencia = EasyMock.createMock(DesafioPersistenciaInterface.class);
    desafioNegocio.setPersistencia(persistencia);
}

@Before
public void methodSetUp() {
    EasyMock.reset(persistencia);
}

@Test
public void testSalvar() throws Exception {
    DesafioEntidade entidade = getEntidadeValida();
    DesafioEntidade entidadeRetornada = getEntidadeValida();
    entidadeRetornada.setId(1L);

    persistencia.salvar(entidade);
    EasyMock.expectLastCall()
        .andReturn(entidadeRetornada)
        .once();
    EasyMock.replay(persistencia);

    DesafioEntidade entidadePersistida = desafioNegocio.salvar(entidade);
    assertNotNull(entidadePersistida.getId());

    EasyMock.verify(persistencia);
}
```

MAIS CONTEÚDO...

- Documentação oficial
- http://easymock.org/EasyMock3_2_Documentation.html

MÃOS À OBRA

- Colocar o *easymock* no *classpath* da aplicação
- Criar os testes para a classe “EntidadeNegocio”
 - Barra Verde
 - 100% das linhas cobertas
- Criar um ÚNICO método que:
 - Verifique a quantidade de registros;
 - Insere um registro;
 - Verifica que a quantidade foi incrementada
 - Tenta inserir o mesmo registro mas recebe uma exceção
 - Excluir o registro
 - Verifica que a quantidade foi decrementada

REQUISITOS

- Atributos:
 - **nome:**
 - O tamanho deve ser entre 5 e 30 caracteres.
 - Campo obrigatório
 - **dataInicial:**
 - A data inicial não pode ser inferior à data atual
 - **dataFinal:**
 - não pode ser inferior à data inicial
 - Campo obrigatório apenas se a data inicial for informada
 - **numeroDocumento:**
 - Deve ser um numero maior que zero
 - Campo obrigatório
 - **tipoDocumento:**
 - Deve ser 1 ou 2
 - Campo obrigatório
 - **email:**
 - deve conter '@' e '.'
- Regras:
 - Entidades do tipo CPF não pode ser removidas
 - Não podem existir entidades com nomes iguais
 - Não é possível alterar o nome da entidade
 - A data de gravação é atribuída no momento da gravação no banco