

# Teste de Unidade: Mocking

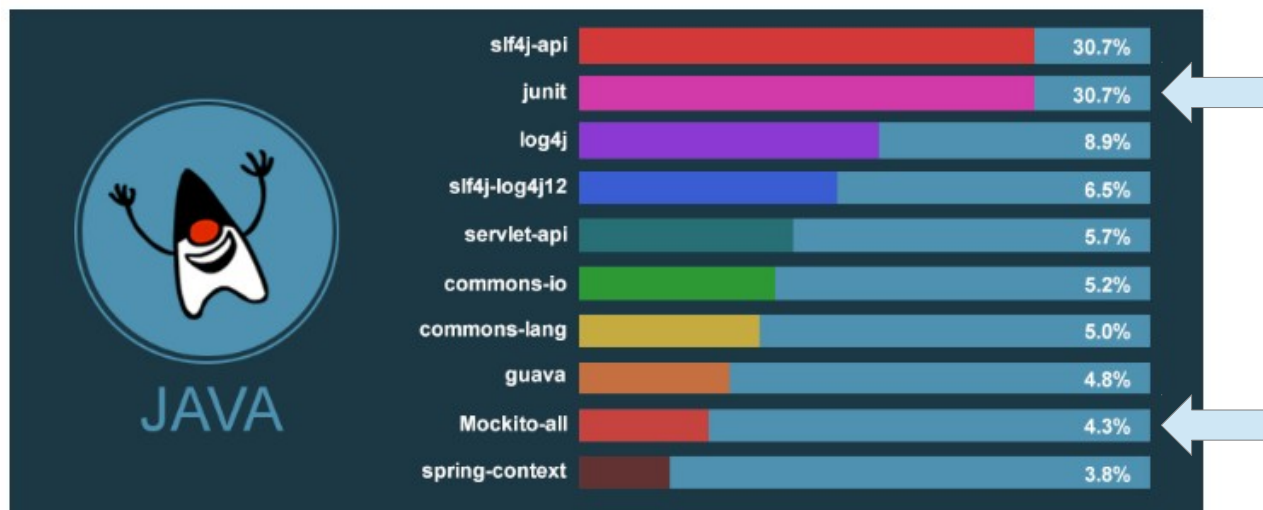
Prof. André Takeshi Endo

# Introdução

- Com *frameworks* xUnit (e.g. Junit), estamos testando apenas unidades?
- Se métodos de outras classes estão sendo chamados, já é integração
- Quero que meu teste continue sendo de unidade
- ***Solução: criar objetos que simulam o comportamento complexo de objetos reais de maneira controlada***
- ***Mock objects***

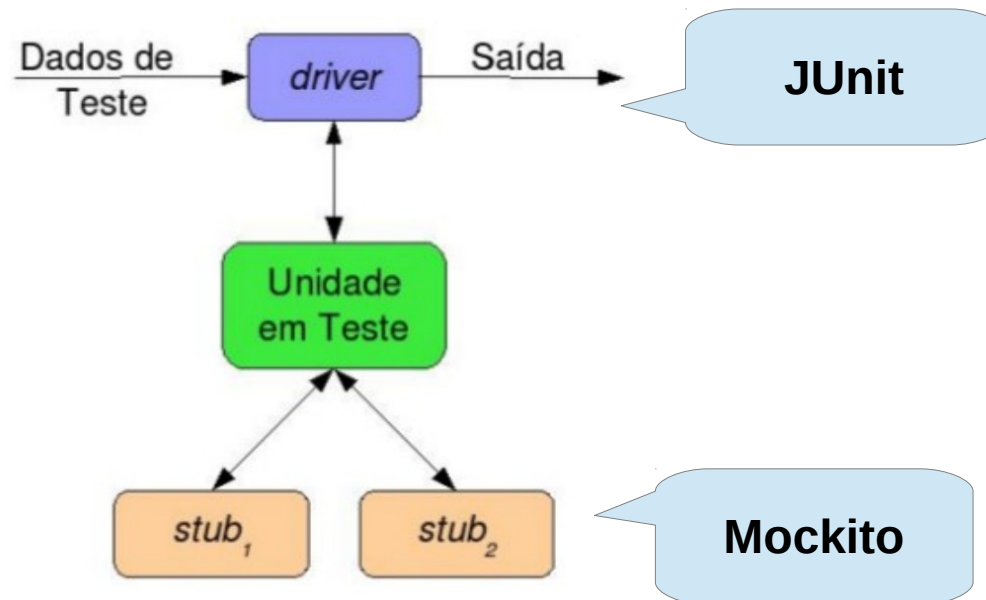
# Mockito

- Mockito
  - *Framework de mocking para teste de unidade em Java*
- Por quê?



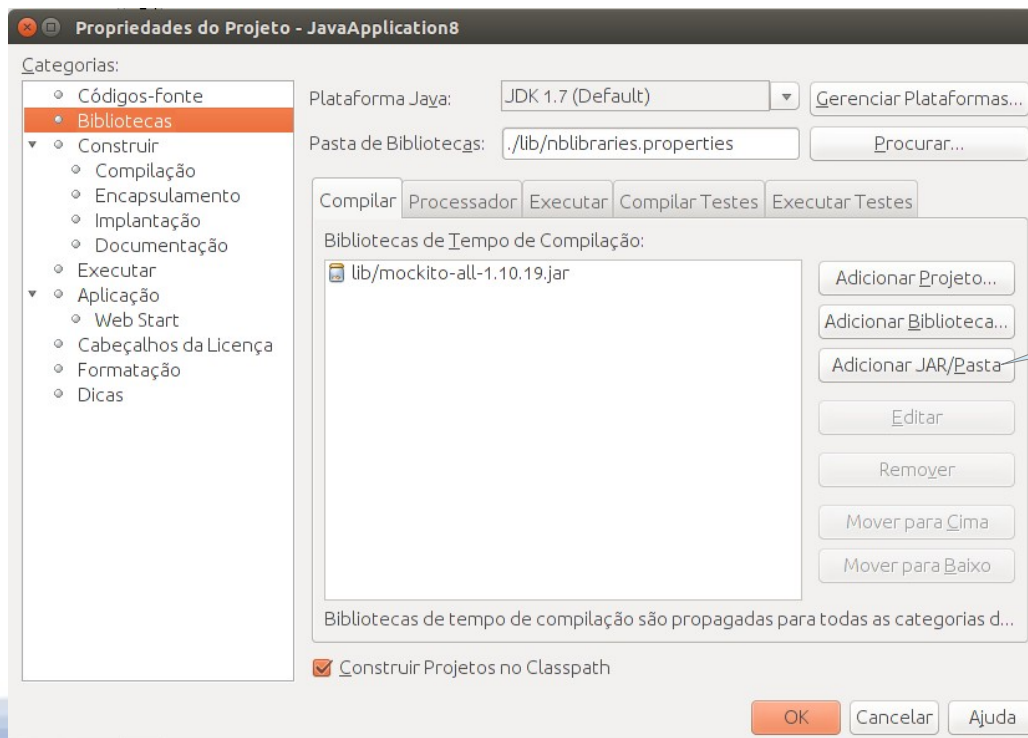
# Mockito

- Usar o Mockito para:
  - Verificar se interações com um objeto aconteceram
  - Fazer o *stub* de métodos chamados



# Mockito

- Fazer o *download* do jar do Mockito (Moodle)
- Incluir o jar no projeto
- Netbeans → botão direito no projeto → propriedades → aba biblioteca



- Clique aqui!  
- Localize o diretório onde  
está o jar

# Mockito

- Importar os métodos do Mockito

```
import static org.mockito.Mockito.*;

public class JUnitTest01 {
    ...
    @Test
    public void verificarInteracoes() {

        List mockedList = mock(List.class);

    }
    ...
}
```

# Mockito

- Fazer o mock de classes

```
import static org.mockito.Mockito.*;

public class JUnitTest01 {
    ...
    @Test
    public void verificarInteracoes() {
        List mockedList = mock(List.class);

    }
    ...
}
```

- Criação do objeto Mock  
Você pode “*mockar*” qualquer classe ou interface.

Forma genérica:

<Tipo> objetoMockado = mock(<Tipo>.class);

# Mockito

- Verificar interações com um objeto
  - Determinado método foi chamado?
  - Os métodos foram chamados nesta ordem?
- Simular a resposta de um método
  - Retorna um valor (objeto)
  - Retorna exceção



# Mockito

- **Determinado método foi chamado?**

```
@Test
public void verificarInteracoes() {
    List mockedList = mock(List.class);

    ...

    mockedList.add("um");
    mockedList.add("dois");
    mockedList.clear();

    ...

    verify(mockedList).add("dois");
    verify(mockedList).add("um");
    verify(mockedList).clear();
}
}
```

# Mockito

- **Determinado método foi chamado?**

```
@Test  
public void verificarInteracoes() {  
    List mockedList = mock(List.class);
```

...

```
mockedList.add("um");  
mockedList.add("dois");  
mockedList.clear();
```

Acontece a interação com o objeto mockado!

...

```
verify(mockedList).add("um");  
verify(mockedList).add("dois");  
verify(mockedList).clear();
```

Verifica se métodos do mock foram invocados.

```
}
```

```
}
```

# Mockito

- **Determinado método foi chamado?**

```
@Test  
public void verificarInteracoes() {  
    List mockedList = mock(List.class);
```

```
    ...
```

```
    mockedList.add("um");  
    mockedList.add("dois");  
    mockedList.clear();
```

```
    ...
```

```
    verify(mockedList).add("um");  
    verify(mockedList).add("dois");  
    verify(mockedList).clear();
```

```
    }
```

```
}
```

**Adicione uma verificação  
para um método que não  
foi invocado anteriormente.  
O que acontece?**

# Mockito

- **Determinado método foi chamado?**
  - E se foi chamado x vezes?

```
@Test
public void verificarInteracoes() {
    List mockedList = mock(List.class);

    mockedList.add("um");
    mockedList.add("dois");
    mockedList.add("tres");

    verify(mockedList, times(1)).add("um");

    verify(mockedList, times(3)).add( anyString() );
}
```

# Mockito

- **Determinado método foi chamado?**
  - E se foi chamado x vezes?

```
@Test
public void verificarInteracoes() {
    List mockedList = mock(List.class);

    mockedList.add("um");
    mockedList.add("dois");
    mockedList.add("tres");

    verify(mockedList, times(1)).add("um");

    verify(mockedList, times(3)).add( anyString() );
}
```

Verifica se o método *add("um")* foi chamado 1 vez.

Verifica se o método *add()* com qualquer parâmetro String foi chamado 3 vezes.

# Mockito

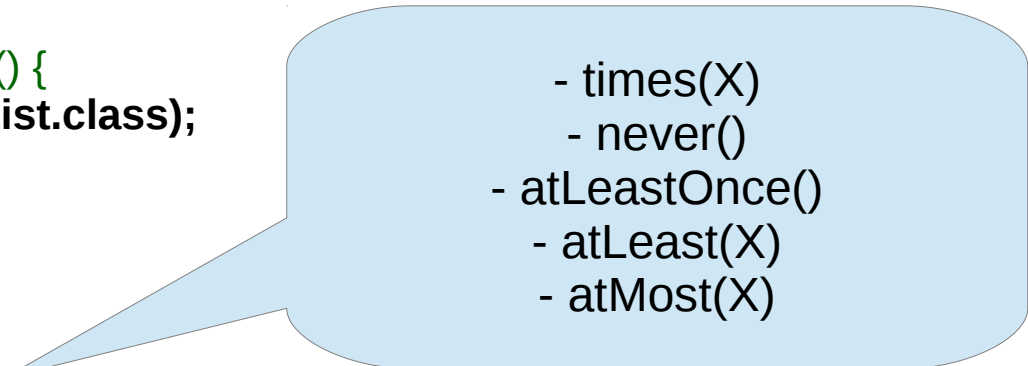
- **Determinado método foi chamado?**
  - E se foi chamado x vezes?

```
@Test
public void verificarInteracoes() {
    List mockedList = mock(List.class);

    mockedList.add("um");
    mockedList.add("dois");
    mockedList.add("tres");

    verify(mockedList, times(1)).add("um");

    verify(mockedList, times(3)).add( anyString() );
}
```



- times(X)
- never()
- atLeastOnce()
- atLeast(X)
- atMost(X)

# Mockito

- Os métodos foram chamados nesta ordem?

```
@Test
public void verificarInteracoesNaOrdem() {
    List mockedList = mock(List.class);

    mockedList.add("um");
    mockedList.add("dois");
    mockedList.add("tres");

    InOrder inOrder = inOrder(mockedList);
    inOrder.verify(mockedList).add("um");
    inOrder.verify(mockedList).add("dois");
    inOrder.verify(mockedList).add("tres");
}
```

Usando a classe InOrder, você consegue garantir que a verificação dos métodos invocados aconteceu na ordem estabelecida.

Tente usar uma ordem diferente!

# Mockito

- **Retorna um valor (objeto)**

```
@Test
public void stubMethods02() {
    List mockedList = mock(ArrayList.class);

    when( mockedList.get(0) ).thenReturn("zero");
    when( mockedList.contains(any()) ).thenReturn(Boolean.TRUE);

    System.out.println(mockedList.get(0));
    System.out.println(mockedList.get(10));
    System.out.println(mockedList.contains("joao"));
}
```



# Mockito

- Retorna um valor (objeto)

Define o comportamento para a interação com determinados métodos.

```
@Test
public void stubMethods() {
    List mockedList = mock(ArrayList.class);

    when( mockedList.get(0) ).thenReturn("zero");
    when( mockedList.contains(any()) ).thenReturn(Boolean.TRUE);

    System.out.println(mockedList.get(0));
    System.out.println(mockedList.get(10));
    System.out.println(mockedList.contains("joao"));
}
```

Aqui os métodos são chamados!  
O que é impresso na tela?

# Mockito

- **Retorna exceção**

```
@Test(expected = IndexOutOfBoundsException.class)
public void stubMethodException() {
    List mockedList = mock(ArrayList.class);

    when( mockedList.get(30) ).thenThrow( new IndexOutOfBoundsException() );

    mockedList.get(30);
}
```

# Mockito

- **Retorna exceção**

Define o comportamento de exceção para a interação com determinados métodos.

`when(<método do mock chamado>).thenThrow(<objeto herda Throwable>);`

```
@Test(expected = IndexOutOfBoundsException)
public void stubir...
```

```
    List mockedList = mock(ArrayList.class);
```

```
    when( mockedList.get(30) ).thenThrow( new IndexOutOfBoundsException() );
```

```
    mockedList.get(30);
```

```
}
```

# Mockito

- **Argument matchers**

`mockedObject.callMethod(<____>)....`

- `any()` → qualquer valor de argumento
- `any<String | Float | Int ...> ()` → qualquer argumento deste tipo
- `anyCollection()`  
`anyList()`

# Mockito

- **Exemplo:** abrir portão com cancela
- Cada automóvel possui uma tag (string)
- Todos os automóveis (tag, placa e dono) conhecidos são armazenados em um BD
  - Caso conhecido, a cancela levanta, a mensagem “seja bem-vindo” é mostrada e o sistema registra o horário de entrada do automóvel
  - Caso contrário, uma mensagem é dada ao usuário
- Assuma que o método abrir() da classe PortaoController é chamado toda vez que um automóvel é detectado pelo sensor;
  - A tag do automóvel é passada como parâmetro ao método abrir
  - O valor “ERRO” é enviado caso o automóvel não tenha tag
- Crie classes que encapsulem a interação com o hardware da cancela e de um display (mensagens ao usuário)
- Implementar e testar o método abrir() usando JUnit, e Mockito para simular os objetos associados.

# Mockito

- **Exemplo:** abrir portão com cancela
- Classes
  - Automovel
  - AutomovelDAO
  - Cancela
  - Display
  - PortaoController
  - RegistroEntradaDAO
- Método da classe PortaoController → “*public boolean abrir(String tag)*”

# Bibliografia

- <http://mockito.org/>
- <http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html>
- <http://junit.org/>
- <http://www.tutorialspoint.com/junit/>
- <http://www.vogella.com/articles/JUnit/article.html>
- <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>