UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

# Lista de Exercícios: Teste de Unidade

*Prof. André Takeshi Endo*

## Mockito

**(Exercício 1)** Considere as classes abaixo.

**Pessoa.java**

```
public class Pessoa {
   int codigo, idade;
   String nome;

   //getters and setters
   public int getCodigo() {
      return codigo;
   }
   …
}
```

**RHService.java**

```
public interface RHService {
   public ArrayList<Pessoa> getAllPessoas();
}
```

**PessoaDAO.java**

```
public class PessoaDAO {

   RHService rhservice;

   public PessoaDAO(RHService rhservice) {
      this.rhservice = rhservice;
   }

   public boolean existePessoa(String nome) {
      ArrayList<Pessoa> pessoas = rhservice.getAllPessoas();
      for(Pessoa p : pessoas) {
         if(p.getNome().equalsIgnoreCase(nome))
            return true;
      }
      return false;
   }
}
```

Implemente dois casos de teste em JUnit para o método “existePessoa(..)”:

- (i) um CT deve testar a situação no qual a pessoa existe, e
- (ii) no qual a pessoa não existe.

Use o Mockito para simular o retorno do método “getAllPessoas()”.

**(Exercício 2)** Considere as classes abaixo.

**MathOps.java**

```
public interface MathOps {
   public int fatorial(int n);
}
```

**Somatoria.java**

```
public class Somatoria {
   MathOps mathOps;

   public Somatoria(MathOps mathOps) {
      this.mathOps = mathOps;
   }

   /**
    * @param numeros
    * @return a somatoria do fatorial de cada inteiro no array numeros
    */
   public int somaDeFatoriais(int numeros[]) {
      //TODO

      return 0;
   }
}
```

Implemente o método “somaDeFatoriais()” segundo o que está especificado no comentário e dois casos de teste em JUnit:

- com o vetor = {3, 4}
- com o vetor = {0, 1, 2, 3, 4}

Use o Mockito para simular o retorno do método “fatorial()”. Verifique também quantas vezes o método “fatorial()” foi chamado em cada CT.

**(Exercício 3)** Estude o funcionamento da classe HashMap do Java. Com base nesta classe, elabore vários casos de teste que ilustrem a utilização do Mockito. Dê maior ênfase as funcionalidades mencionadas nos slides do Mockito mas que não foram ilustradas nos exemplos.

**(Exercício 4)** [Adaptado de Acharya2015] Simulação de cotação de ações. O software observa tendências do mercado e:

- compra novas ações
- vende ações existentes

Considere que o sistema possui as seguintes classes:

- MarketWatcher
- Portfolio
- StockBroker
- Stock com os atributos symbol, companyName e price.

Os testes serão realizados sob o método perform() da classe StockBroker, apresentada a seguir. O método perform() funciona da seguinte forma:

- aceita um portfolio e uma ação (stock)
- recupera o preço atual de mercado
- compara o preço atual com a média das ações compradas
- Se o preço atual subiu 10%, ele vende 10 ações
  - Caso contrário, ele compra ações.

```
public class StockBroker {
  private final static BigDecimal LIMIT
            = new BigDecimal("0.10");

  private final MarketWatcher market;

  public StockBroker(MarketWatcher market) {
    this.market = market;
  }

  public void perform(Portfolio portfolio,Stock stock) {
    Stock liveStock = market.getQuote(stock.getSymbol());
    BigDecimal avgPrice = portfolio.getAvgPrice(stock);
    BigDecimal priceGained =
        liveStock.getPrice().subtract(avgPrice);
    BigDecimal percentGain = priceGained.divide(avgPrice);
    if(percentGain.compareTo(LIMIT) > 0) {
      portfolio.sell(stock, 10);
    }else if(percentGain.compareTo(LIMIT) < 0){
      portfolio.buy(stock);
    }
  }
}
```

A classe Portfolio lê informações de um banco de dados e a classe MarketWatcher conecta na Internet para recuperar as cotações atuais. Nesse caso, para testar o método perform() essas funcionalidades não estão disponíveis.

Implemente casos de teste em JUnit e Mockito para testar o método perform() nas seguintes situações:

- ações são vendidas.
- ações são compradas.
- o objeto da classe MarketWatcher não consegue conexão com a Internet e lança um exceção.
- o objeto da classe Portfolio não consegue conexão com o BD e lança exceções em diferentes pontos.

**(Exercício 5)** Considere as três classes abaixo:

**MathOps.java**

```
public interface MathOps {
   public int fatorial(int n);
}
```

**Primo.java**
```
public interface Primo {
   public boolean ehPrimo(int n);
}
```

**Somatoria.java**
```
public class Somatoria {
   MathOps mathOps;
   public Somatoria(MathOps mathOps) {
       this.mathOps = mathOps;
   }

   /**
   * @param numeros
   * @return a somatoria do fatorial de cada inteiro no array numeros
   * que nao eh primo
   */
   public int somaDeFatoriais(int numeros[], Primo p) {
      int soma = 0;
      for (int i = 0; i < numeros.length; i++) {
         int numero = numeros[i];
         if(! p.ehPrimo(numero)) {
            soma += numero;
         }
      }
      return 0;
   }
}
```

Implemente casos de teste em JUnit para o método “somaDeFatoriais(..)” da classe “Somatoria”. Use o Mockito para simular e verificar interações com as classes “MathOps” e “Primo”. Implemente os casos de teste, considerando os vetores abaixo como entrada:

- {5, 10};
- {3, 4, 4, 5}

**(Exercício 6)** Considere as duas classes e duas interfaces abaixo:

**Funcionario.java**
```
public class Funcionario {
   private int id;
   private String nome, cpf;

   //getters e setters
}
```

**FuncionarioDAO.java**
```
public interface FuncionarioDAO {
   public ArrayList<Funcionario> getFuncionariosBy(String categoria);
```

```
}
```

**ReceitaFederal.java**

```
public interface ReceitaFederal {
    public boolean isCPFBloqueado(String cpf);
}
```

**RelatorioDeFuncionarios.java**

```
public class RelatorioDeFuncionarios {

    FuncionarioDAO funcDao;
    ReceitaFederal rf;

    public RelatorioDeFuncionarios(FuncionarioDAO funcDao) {
        this.funcDao = funcDao;
    }

    public void setRf(ReceitaFederal rf) {
        this.rf = rf;
    }

    //retorna a qtde de funcionarios da categoria fornecida com o cpf bloqueado
    public int getFuncComCPFBloqueado(String categoria) {
        int numeroDeFuncionarios = 0;

        ArrayList<Funcionario> funcCategoria = funcDao.getFuncionariosBy(categoria);

        for(Funcionario f : funcCategoria) {
            if( rf.isCPFBloqueado(f.getCpf()) )
                numeroDeFuncionarios++;
        }

        return numeroDeFuncionarios;
    }
}
```

Implemente casos de teste em JUnit para o método "getFuncComCPFBloqueado(..)" da classe "RelatorioDeFuncionarios". Use o Mockito para simular e verificar interações. Implemente três casos de teste para os seguintes cenários:

- Existem 2 funcionários na categoria "tecnico" que não estão com o CPF bloqueado.
- Existe 1 funcionário na categoria "analista" que está com o CPF bloqueado.
- Existem 4 funcionários na categoria "gerente" com os CPFs: (123456789-00, 111222333-44, 654321987-23, 098876654-99), sendo que os CPFs 111222333-44 e 098876654-99 estão bloqueados.

**(Exercício 7)** Considere as classes e interfaces a seguir; as mesmas não podem ser alteradas.

**TurmaController.java**

```
public class TurmaController {
   TurmaDAO turmaDao;
   VerificadorDeCodigos verificador;

   public TurmaController(TurmaDAO pTurmaDao) {
      turmaDao = pTurmaDao;
   }

   public void setVerificador(VerificadorDeCodigos verificador) {
      this.verificador = verificador;
   }

   public String cadastrarTurma(Turma t) {
                                                          if(!
verificador.verificarCodigoDisciplina(t.getCodDisciplina()))
         return "codigo disciplina invalido";

      if(! verificador.verificarCodigoTurma(t.getCodTurma()))
         return "codigo turma invalido";

      if(turmaDao.existe(t))
         return "turma ja existe";

      if(turmaDao.salvar(t))
         return "turma salva com sucesso";
      else
         return "turma nao salva. Erro no BD";
   }
}
```

**Turma.java**

```
public class Turma {
   String codDisciplina, codTurma;
   int maximoAlunos;

   //adicionar os getters e setters
}
```

**TurmaDAO.java**

```
public interface TurmaDAO {
   public boolean existe(Turma turma);
   public boolean salvar(Turma turma);
}
```

**VerificadorDeCodigos.java**

```
public interface VerificadorDeCodigos {
   public boolean
            verificarCodigoDisciplina(String codigo);

   public boolean
            verificarCodigoTurma(String codigo);
}
```

Implemente casos de teste em JUnit para o método “cadastrarTurma(..)” da classe “TurmaController”. Use o Mockito para simular as interfaces “TurmaDAO” e “VerificadorDeCodigos”. Implemente cinco casos de teste para cada um dos cinco possíveis retornos do método “cadastrarTurma(..)”.

**(Exercício 8)** Considere as 2 classes e 2 interfaces a seguir; as mesmas não podem ser alteradas.

**Auditor.java**

```
public class Auditor {
   Validador v;
   EmpregadoDAO dao;

   public Auditor(EmpregadoDAO dao) {
      this.dao = dao;
   }

   public void setValidador(Validador v) {
      this.v = v;
   }

   public String getSuperSalarios(String categoria, long salario) {
      if(! v.ehCategoriaValida(categoria))
                  throw new IllegalArgumentException("categoria
invalida");

      if(! v.ehSalarioValido(salario))
         throw new IllegalArgumentException("salario invalido");

      List<Empregado> emps = dao.getAll();
      int naCategoria = 0, acima = 0;
```

**Empregado.java**

```
public class Empregado {
   long salario;
   String categoria;

    public Empregado(long salario, String categoria)
{
      this.salario = salario;
      this.categoria = categoria;
   }

   public String getCategoria() {
      return categoria;
   }

   public long getSalario() {
      return salario;
   }
}
```

**Validador.java**

```
public interface Validador {
         public   boolean   ehCategoriaValida(String
```

```
        for (Empregado e : emps) {
            if(categoria.equals( e.getCategoria())) {
                naCategoria++;

                if(e.getSalario() > salario)
                    acima++;
            }
        }

        if(naCategoria == 0)
            return "ninguem na categoria";

        if(acima == 0)
            return "ninguem na categoria com salario alto";

        return "pessoas com salario alto: " + acima;
    }
}
```

```
categoria);

    public boolean ehSalarioValido(long salario);
}
```

**EmpregadoDAO.java**

```
public interface EmpregadoDAO {
    public List<Empregado> getAll();
}
```

Implemente casos de teste em JUnit para o método “getSuperSalarios(..)” da classe “Auditor”. Use o Mockito para simular as interfaces “Validador” e “EmpregadoDAO”. Implemente cinco casos de teste, sendo 2 para as duas possíveis exceções (categoria e salario) e 3 casos de teste para cada um dos possíveis retornos do método “getSuperSalarios(..)”.

# Referências


- Sujoy Acharya. “Mockito for Spring”, 2015. 178 pages, Packt Publishing Limited.