UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

Lista de Exercícios: Teste caixa-preta
particionamento em classes de equivalência (PCE)
análise de valor limite (AVL)

Prof. André Takeshi Endo

Para cada um dos exercícios a seguir, aplique os critérios de teste caixa-preta (funcional): ***particionamento em classes de equivalência e análise de valor limite.*** *Caso uma descrição de classes não seja fornecida, elabora as classes e implemente os casos de teste com o auxílio do* ***JUnit e Mockito****.*

Os exercícios de listas anteriores podem ser refeitas com a diferença que para projetar os casos de teste os critérios de teste sejam aplicados.

(Exercício 1) Cálculo da hipoteca (Mousavi). Considere os seguintes requisitos:

- R1- O sistema deve receber três valores como entrada: gênero (true → feminino e false → masculino), idade ([18, 55]) e salário ([0-10000]). Como saída, o sistema deve calcular o valor máximo da hipoteca para essa pessoa.
- R2- O valor máximo da hipoteca é calculado pela multiplicação do valor do salário com um fator (tabela no R4).
- R3- Mensagens de erro específicas devem ser geradas para valores inválidos de idade e salário.
- R4- O fator para calcular a hipoteca (R2) é definido pela tabela a seguir:

Categoria	Homem	Fator	Mulher	Fator
Jovem	18-35 anos	75	18-30 anos	70
Médio	36-45 anos	55	31-40 anos	50
Idoso	46-55 anos	30	41-50 anos	35

(Exercício 2) Considere um método para adição segura. A assinatura do método é apresentada a seguir:

int safe_add(int a, int b) throws OverflowException, UnderflowException;

Tal método faz a adição segura, tratando possíveis casos de overflow e underflow por meio do lançamento de exceções.

Dica: Em Java, as constantes `Integer.MIN_VALUE` e `Integer.MAX_VALUE` são usadas para referenciar o limite inferior e superior de variáveis do tipo int.

(Exercício 3) Um funcionário recebe um salário mensal e pode ganhar várias bonificações esporádicas ao longo do ano. Com essas informações, o software deve calcular o ganho mensal do funcionário e retornar qual a sua aliquota de imposto de renda (ver a tabela a seguir).

Base de cálculo mensal em R$	Alíquota %
Até 1.903,98	–
De 1.903,99 até 2.826,65	7,5
De 2.826,66 até 3.751,05	15,0
De 3.751,06 até 4.664,68	22,5
Acima de 4.664,68	27,5

(Exercício 4) Considere uma classe que receba como entrada o peso em quilos e a altura em metros e, com base no cálculo do IMC e na tabela a seguir, fornece a situação atual. Caso as entradas não atendam as seguintes restrições, exceções específicas precisam ser geradas:

- peso de 40 a 200 quilos.
- altura de 1,20 m a 2,5 m.

Resultado	Situação
Abaixo de 17	Muito abaixo do *peso*
Entre 17 e 18,49	Abaixo do *peso*
Entre 18,5 e 24,99	*Peso* normal
Entre 25 e 29,99	Acima do *peso*
Entre 30 e 34,99	*Obesidade* I
Entre 35 e 39,99	*Obesidade* II (severa)
Acima de 40	*Obesidade* III (mórbida)

(Exercício 5) 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 casos de teste em JUnit para o método “existePessoa(..)”.

(Exercício 6) 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 CTs em JUnit.

(Exercício 7) [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.

A classe Portfolio lê

```
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);
    }
  }
}
```

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().

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

(Exercício 8) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "validarNovaSenha()" especificada a seguir. Implemente os casos de teste com o auxílio do JUnit e Mockito. Considere as classes abaixo:

Dicionario.java

```
public interface Dicionario {
   public ArrayList<String> getListaDeSenhasInvalidas();
}
```

VerificadorDeSenhas.java

```
public class VerificadorDeSenhas {

   Dicionario dic;

   public VerificadorDeSenhas(Dicionario dic) {
      this.dic = dic;
   }

   public boolean validarNovaSenha(String senhaFornecida) {
       //TODO
       return true;
   }
}
```

O método "validarNovaSenha()" recebe como entrada uma senha e retorna se é válida (true) segundo as regras abaixo:

- A senha deve ter de 5 a 10 caracteres
- O primeiro caracter deve ser alfabético, numérico ou "_" (*underscore*)
- Os outros caracteres podem ser quaisquer combinação de caracteres alfabético, numérico ou "_" (*underscore*).
- A senha não pode existir em um dicionário pré-definido. No exemplo, a lista de senhas que não podem ser usadas é retornada do método "getListaDeSenhasInvalidas()" da interface Dicionario.

Se a senha for inválida, retorna false. Envie junto uma planilha que ilustra a criação das classes válidas e inválidas. Implemente apenas os testes.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

(Exercício 10) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "cadastrarTurma(..)" especificada a seguir. Implemente os casos de teste com o auxílio do JUnit e Mockito. Considere as classes abaixo:

VerificadorDeCodigos.java

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

BancoDeDados.java

```
public class BancoDeDados {
   private VerificadorDeCodigos verificador;

   public BancoDeDados(VerificadorDeCodigos verificador) {
      this.verificador = verificador;
   }

   public String cadastrarTurma(String codDisciplina, String codTurma, int numeroAlunos) {
      //todo
      return "";
   }
}
```

O método "cadastrarTurma(..)" recebe como entrada o código da disciplina, o código da turma e número de alunos e retorna:

- "Codigo de disciplina invalido" se não atender ao padrão: string com examente 5 caracteres sendo os dois primeiros letras, os dois seguintes números e o último caracter uma letra.
- "Codigo de turma invalido" se não atender ao padrão: string com examente 4 caracteres sendo os dois primeiros letras e os dois seguintes números.
- "Numero de alunos invalido" se a turma não tiver ao menos 3 alunos e no máximo 44 alunos.
- "Sucesso" se todas as informações estiverem corretas.

O método "cadastrarTurma(..)" utiliza métodos da interface "VerificadorDeCodigos" para validar os códigos de disciplina e turma.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

(Exercício 12) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "*calcularFrete(..)*" especificada a seguir. Implemente os casos de teste com o auxílio do JUnit e Mockito. Considere as classes a seguir:

CustoDAO.java

```
public interface CustoDAO {
   public int getCustoPorGrama(String regiao);
}
```

Calculadora.java

```
public class Calculadora {
   private CustoDAO custoDao;

   public void Calculadora(CustoDAO custoDao) {
      this.custoDao = custoDao;
   }

   public int calcularFrete(String regiao, int peso)  throws Exception {
      //nao precisa implementar
      return 0;
   }
}
```

O método "*calcularFrete(..)*" recebe como entrada a região (string) e o peso (em gramas) e segundo as regras abaixo calcula o custo do frete:

- A região deve ser uma das seguintes strings: "norte", "nordeste", "centro", "sudeste" e "sul". Uma exceção deve ser lançada, caso contrário.
- O peso deve ser maior que 0 (>0) e menor ou igual a 2.000.000. Uma exceção deve ser lançada, caso contrário.
- O método "getCustoPorGrama(..)" da interface CustoDAO retorna o custo por grama para a região informada. Se a região não existir, o método retorna -10.
- O frete é calculado pela multiplicação do peso com o valor retornado pela interface CustoDAO para a região informada.

Elabore uma planilha que ilustra a criação das classes válidas e inválidas. Implemente apenas os testes.

(Exercício 14) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "cadastrarTurma(..)" especificada a seguir. Implemente os casos de teste com o auxílio do JUnit e Mockito. Considere as classes abaixo:

VerificadorDeCodigos.java

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

BancoDeDados.java

```
public class BancoDeDados {
   private VerificadorDeCodigos verificador;

   public BancoDeDados(VerificadorDeCodigos verificador) {
      this.verificador = verificador;
   }

   public String cadastrarTurma(String codDisciplina, String codTurma, int numeroAlunos) {
      //todo
      return "";
   }
}
```

O método "cadastrarTurma(..)" recebe como entrada o código da disciplina, o código da turma e número de alunos e retorna:

- "Codigo de disciplina nao é valido" se não atender ao padrão: string com examente 6 caracteres sendo os dois primeiros números, os dois seguintes letras em maiúsculo e o último caracter um número.
- "Codigo de turma fora do padrao" se não atender ao padrão: string com examente 3 caracteres sendo os dois primeiros letras e o seguinte número.
- "Numero de alunos invalido" se a turma não tiver ao menos 5 alunos e no máximo 30 alunos.
- "Turma cadastrada com sucesso" se todas as informações estiverem corretas.

O método "cadastrarTurma(..)" utiliza métodos da interface "VerificadorDeCodigos" para validar os códigos de disciplina e turma.

(Exercício 16) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "*autorizar(..)*" especificada a seguir. Considere as classes a seguir:

ClienteDao.java

```
public interface ClienteDao {
    public boolean ehCliente(String nome);
}
```

MontanhaRussaControlador.java

```
public class MontanhaRussaControlador {
    ClienteDao clienteDao;

    public MontanhaRussaControlador(ClienteDao pClienteDao) {
        clienteDao = pClienteDao;
    }

    public String autorizar(String nome, int idade) throws Exception {
        //todo
        return "";
    }
}
```

O método "*autorizar(..)*" recebe como entrada o nome e a idade de uma pessoa e segundo as regras a seguir decide qual o procedimento para andar na montanha russa:

- O nome deve ser composto somente de letras e espaço e conter ao menos duas palavras. A idade deve ser ao menos 1 e no máximo 120 ([1,120]). Caso o nome ou idade sejam inválidos, uma exceção deve ser lançada.
- A interface ClienteDao é usada para verificar se a pessoa é cliente usando o nome; caso não seja uma exceção deve ser lançada.
- Ao passar pelas verificações anteriores, o método retorna:
  - *"autorizado"* se idade >=18 e <=90;
  - *"acompanhado dos pais"* se idade < 18; ou
  - *"acompanhado do responsavel legal"* se idade > 90.

Elabore uma planilha que ilustra a criação das classes válidas e inválidas. Implemente os casos de teste com o auxílio do JUnit e Mockito.

(Exercício 18) Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método "*efetuarPedidoDeOrcamento(..)*" especificada a seguir. Considere as classes abaixo:

ProdutoDAO.java

```
public interface ProdutoDAO {
    public int getQuantidadeDisponivel(String codigoDeBarras);
}
```

SemEstoqueException.java

```
public class SemEstoqueException extends Exception { }
```

Atendente.java

```
public class Atendente {
    private ProdutoDAO produtoDAO;

    public Atendente(ProdutoDAO produtoDAO) {
        this.produtoDAO = produtoDAO;
    }

    public String efetuarPedidoDeOrcamento(String codigoDeBarras, int quantidade)
        throws IllegalArgumentException, SemEstoqueException {

        return "";
    }
}
```

O método "*efetuarPedidoDeOrcamento(..)*" recebe como entrada o código de barras de um produto e a quantidade solicitada e segundo as regras a seguir decide qual tipo de orçamento:

- O código de barras deve ser composto somente de números e ter o tamanho exato de 13 caracteres. A quantidade é no mínimo 1 e no máximo 1000 produtos. Caso o código ou a quantidade sejam inválidos, uma `IllegalArgumentException` deve ser lançada.
- Se as entradas forem válidas, a quantidade em estoque é recuperada da classe `ProdutoDAO` e comparada com a quantidade solicitada. Se a quantidade solicitada for maior que a em estoque, `SemEstoqueException` deve ser lançada.
- Ao passar pelas verificações anteriores, o método retorna:
  - *"Orcamento normal"* se a quantia solicitada for menor que metade do estoque, ou
  - *"Orcamento variavel"* se a quantia solicitada for maior ou igual a metade do estoque.

Envie junto uma planilha que ilustra a criação das classes válidas e inválidas. Implemente os casos de teste com o auxílio do JUnit e Mockito.

(*Exercício 20)* Aplique os critérios de teste funcional ***particionamento em classes de equivalência*** e ***análise de valor limite*** para o método *"consultarSituacao(..)"* especificada a seguir. Implemente os casos de teste com o auxílio do JUnit e Mockito. Considere as classes abaixo:

CartorioEleitoral.java

```
public interface CartorioEleitoral {
   /** Esse método pode retornar:
    * - "nao existe": se o cpf não possui titulo associado
    * - "pendencia": o titulo possui alguma pendencia
    * - "OK": situacao regularizada para o título  */

   public String verificar(String cpf);
}
```

VerificadorEleitoral.java

```
public class VerificadorEleitoral {
   private CartorioEleitoral cartorioEleitoral;

   public VerificadorEleitoral(CartorioEleitoral cartorioEleitoral) {
      this.cartorioEleitoral = cartorioEleitoral;
   }

   public String consultarSituacao(int idade, String cpf) throws Exception {
      //todo
      return "";
   }
}
```

O método *"consultarSituacao(..)"* recebe como entrada a idade e o CPF de uma pessoa, e:

- lança uma exceção se a idade ou CPF forem inválidos. A idade válida deve ser maior que 0 e menor que 200 (0 < idade < 200) e o CPF deve conter 11 caracteres sendo todos números.
- Caso contrário, pode retornar as seguintes strings:
- *"nao pode votar"* se a idade for menor que 16 (<16).
- *"faca um titulo"* se o método verificar() da interface CartorioEleitoral retornar "nao existe"
- *"regularize seu titulo"* se o método verificar() da interface CartorioEleitoral retornar "pendencia"
- Se o método verificar() da interface CartorioEleitoral retornar "OK" então
  - retorna *"voto facultativo"* se a for idade >=16 e <=17 ou idade > 70;
  - *"voto obrigatorio"*, caso contrário.

O método *"consultarSituacao(..)"* utiliza um método da interface "CartorioEleitoral" para verificar a situação do título pelo CPF.

Referências

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