CES-28 Fundamentos de Engenharia de Software	
Primeira Prova do Primeiro Bimestre	
Nome: Gabarito	**Data: 26/08/2014**
Sem Consulta -- 3h	

Obs.: 1. Qualquer dúvida de codificação Java só pode ser sanada com textos oficiais da Oracle ou JUnit.
2. Questões com itens diversos, favor identificar claramente pela letra que representa o item, para que eu saiba precisamente a que item corresponde a resposta dada!
3. Só pode implementar usando Eclipse ou outro ambiente Java as questões ou itens indicados com o rótulo **[IMPLEMENTAÇÃO]**!
4. Incluindo todas as obrigatórias, escolher questões que totalizem 14 pontos no total.
5. Gerar .pdf deste documento Word e postar na atividade P1B1!

Parte 1 – 9 Questões Objetivas /0.5 cada/

1. Marque a alternativa correta sobre TDD:
   a. TDD é uma técnica de testes que serve apenas para criação de testes automatizados.
   b. O isolamento das classes para o teste de unidade no TDD não faz com que diminua o acoplamento na aplicação.
   c. Para que o TDD funcione nunca o código de produção pode ser feito antes dos testes.
   d. Os testes gerados no TDD podem servir como documentação do código fonte.
   e. O primeiro passo no TDD quando se encontra um bug é escrever um teste que detecte aquele bug.

   Resp.: d.

2. [Multiple Choice (multiple correct)] Marque as alternativas corretas sobre TDD. Em que situações se deve introduzir um mock object nos testes quando se utiliza o TDD como técnica de design? /0.5/
   a. A classe precisa acessar funcionalidade de um dispositivo de hardware.
   b. A classe utiliza uma outra classe como parte de sua solução interna.
   c. A classe precisa acessar um sistema remoto.
   d. Você desejar expor a API de uma dependência da classe para qual ela vai delegar funcionalidade.

   Resp.: a., c. e d.
3. Uma aplicação precisa ter diversos modelos de tela representados por classes, entre eles um que possui apenas um formulário simples, um que possui um formulário mestre-detalhe (exemplo: cadastro de uma pessoa [mestre] com sua lista de telefones [detalhe]) e um que possui um formulário com uma tabela. Os

dois últimos são classes que herdam da primeira. Uma classe que precisasse ter um formulário mestre-detalhe mais uma tabela, deveria estender qual classe? /0.5/

a. A classe com o formulário simples
b. A classe com o formulário com a tabela
c. A classe com o formulário mestre-detalhe
d. O uso de herança não é a melhor solução, pois causará duplicação de código

Resp.: d. Seria melhor usar composição!

4. Marque a alternativa ERRADA a respeito de herança e composição: /0.5/
   a. Uma das desvantagens da herança é que em várias linguagens ela só pode ser utilizada uma vez.
   b. Uma das desvantagens da composição é que depois que o objeto que compõe a classe é passado no construtor, o comportamento não pode ser mais alterado.
   c. Uma das vantagens da herança é que os métodos protegidos e públicos podem ser sobrepostos nas subclasses sem que se definam exatamente os pontos de extensão.
   d. Uma das vantagens da composição é que uma classe pode ser composta por diversas instâncias de tipos diferentes, possibilitando a combinação de comportamentos.

   Resp.: b.

5. O princípio de _____ permite que objetos que pertencem a diferentes classes respondam de forma distinta a mensagens idênticas. /0.5/

   Resp.: Polimorfismo

6. Marque Verdadeiro (V) ou Falso (F) para a seguinte afirmativa (Caso a afirmativa seja falsa, torne-a verdadeira):

   (F) Método de acesso, definido em uma dada classe, é um método que possibilita o acesso a um objeto da classe. /0.5/
   Método de acesso, definido em uma dada classe, é um método que possibilita o acesso a uma variável de instância de um objeto da classe.

   (V) Uma herança por especialização ocorre quando a subclasse é uma forma especializada da superclasse, satisfazendo as especificações da superclasse em todos os aspectos relevantes, em especial o relacionamento "é-um". /0.5/

   (V) Um acoplamento abstrato em Java é aquele no qual pelo menos a classe B de duas classes A e B associadas (acopladas) tem que ser classe abstrata ou interface Java. /0.5/

   (F) **[OBRIGATÓRIA]** Supondo que o código em pontinhos compile corretamente, o código abaixo compila corretamente. /0.5/
   Obs. 1: Neste caso, qualquer que seja sua resposta, explique o por quê!

Obs. 2: Não se pode experimentar essa questão no Eclipse ou outro ambiente Java!

```
public class Exemplo {
	public int avaliaEstado(int mensagem, String estado){
		...
	}
	protected String avaliaEstado(int estado, String mensagem){
		...
	}
}
```

Parte 2 – 5 Questões Abertas – /8.0/

7. Diferencie um objeto do tipo Mock de um do tipo Stub. /1.0/
   Resp.: Stub serve para simular o comportamento da classe real de modo que o teste possa ser realizado, sendo apropriado para simular um comportamento desejado; por mock, entende-se um objeto que irá verificar se as ações esperadas foram executadas, sendo apropriado para verificar um comportamento esperado.
8. Dado que Tubarao é subclasse de Peixe, explique por que não é permitido realizar a seguinte atribuição: /1.0/

   ```
   ...
   Tubarao tubarao = new Peixe ( );
   ...
   ```

   Resp.: O princípio da substituição expresso pelo relacionamento "é-um" não é satisfeito, uma vez que um Peixe não-é-um Tubarao; além disso, como o tipo estático do objeto tubarao é Tubarao e a classe Tubarao é subclasse da classe Peixe, podem existir métodos que não tenham sido implementados na classe do tipo dinâmico de tubarao, a saber, Peixe. Na hipótese da atribuição da questão estar correta, seria perfeitamente legal enviar uma mensagem ao objeto tubarao referente a método especificado apenas na classe Tubarao – sem erro de compilação, pois o método é de classe estática Tubarao –, mas não na classe Peixe. Isso causaria um erro de tempo de execução.
9. [a] O que é uma variável imutável ou de atribuição única? Exemplifique uma variável imutável usando métodos de acesso e usando código Java, explicando porque a variável exemplo deve ser imutável e quais as garantias de que no dado contexto ela é imutável mesmo! Não vale repetir exemplo discutido em sala de aula! /1.0/ [b] Qual a diferença de uma variável imutável de uma constante? /0.5/ /Total 1.5/
   Resp.: [a] Uma variável imutável é uma variável de instância que é modificada apenas uma vez em tempo de execução, geralmente na construção do objeto.

   Por exemplo, numa classe carta da baralho para um jogo, uma vez que a carta seja criada, não pode ter seu naipe e valor modificados. Ou seja, naipe e valor devem ser implementados em Java como variáveis imutáveis. Basta então criar

métodos do tipo set privados para naipe e valor, enquanto os métodos do tipo get podem ser protegidos ou públicos e fazer com que o construtor use os métodos do tipo set na criação de objetos cartas. Assim, na criação do objeto a variável imutável tem seu valor mudado em tempo de execução, mas apenas uma vez, pois o construtor só realiza essa tarefa uma única vez e o set é privado, de modo que não dá mais para mudar o valor estabelecido para a variável na construção! Além disso, para garantir a imutabilidade da variável em Java, basta defini-las como final também!

```
public class Carta{
      public Carta(int naipe, int valor){
            setNaipe(naipe);
            setValor (valor);
      }

      private void setNaipe(int naipe){_naipe = naipe;}
      private void setValor (int valor){_valor = valor;}
      public int getNaipe(){return _naipe;}
      public int getValor (){return _valor;}

      private final int _naipe;
      private final int _valor;
}
```

[b] Podemos caracterizar essa diferença em três níveis. No primeiro, a variável imutável tem seu valor atribuído uma única vez em tempo de execução, enquanto uma constante tem seu valor atribuído uma única vez em tempo de compilação. No segundo, normalmente a imutável é uma variável de instância e a constante é uma variável estática (ou de classe ou de interface). E no terceiro, variável imutável vai possuir um valor constante possivelmente diferente para cada instância da classe da qual ela faz parte, enquanto que uma constante possuirá um valor constante para todos os objetos de classes com visibilidade da constante.

10. **[OBRIGATÓRIA]** No trecho de código Java abaixo, claramente a classe SubClass estende a classe SuperClass. /Total 2.0/
    a. Quais são as classes estática e dinâmica das variáveis supC, subC e supC nas linhas 01, 03 e 05, respectivamente? /0.5/

Linha	Variável	Classe Estática	Classe Dinâmica
01	supC	SuperClass	SuperClass
03	subC	SubClass	SubClass
05	supC	SuperClass	SubClass

b. [b(1)] Nas linhas 02, 04 e 06, que valor a variável local x recebe ao final de cada atribuição? /0.5/ [b(2)] Explique, para cada uma dessas linhas, usando os conceitos de "classe estática", "classe dinâmica" e outros pertinentes, por que é assim e por que a atribuição compila corretamente? /0.5/

Linha	Variável Local	Valor	[b(1)]	[b(2)]
02	x	12	Como a variável supC tem classe estática e classe dinâmica SuperClass, o método amarrado dinamicamente é o add(int a, int b)da classe SuperClass!	A variável supC tem classe estática e classe dinâmica SuperClass, que possui método add(int a, int b)!
04	x	2	Como a variável subC tem classe estática e classe dinâmica SubClass, o método amarrado dinamicamente é o add(int a, int b)da classe SubClass!	A variável subC tem classe estática e classe dinâmica SubClass, que possui método add(int a, int b)!
06	x	2	Como a variável supC tem classe estática SuperClass e classe dinâmica SubClass, o método amarrado dinamicamente é o add(int a, int b)da classe SubClass!	A variável supC tem classe estática SuperClass e classe dinâmica SubClass; o método a ser amarrado dinamicamente é o método add(int a, int b) da classe SubClass; compila corretamente porque a classe estática de supC, SuperClass, superclasse de SubClass, tem um método add(int a, int b) com mesma assinatura!

Obs.: Neste caso, era obrigatório responder 'usando os conceitos de "classe estática", "classe dinâmica" e outros pertinentes', como está na questão. Outros pertinentes incluem herança, overriding e polimorfismo. Mas tinha que usar na argumentação os conceitos de "classe estática" e "classe dinâmica"; sem eles a maior parte dos pontos deve ser tirada: perda parcial, se falou bem apenas com herança, overriding e polimorfismo, e perda total, caso contrário!

c. Examinando os métodos das duas classes, pode-se dizer que o princípio da substituição se preserva? Justifique sua resposta! /0.5/

Resp.: De acordo com o princípio da substituição, um tipo A é dito ser um subtipo de um tipo B se uma instância do tipo A pode tomar o lugar do tipo B sem nenhum efeito observável. No caso, o princípio da substituição não se preserva, porque a subclasse SubClass, com base nos métodos add(~) e subtract(~), não "é uma classe SuperClass, ou seja, SubClass não é um subtipo de SuperClass. O método add(~) de SubClass corresponde ao método subtract(~) de SuperClass e o método subtract(~) de SubClass corresponde ao método add(~) de SuperClass. Na linha 06, o efeito observável é exatamente o contrário do add(~) da SuperClass, classe estática da variável supC, pois houve overriding por substituição neste caso, em que a propriedade intrínseca de soma é trocada pela propriedade intrínseca de subtração.

```
public class SuperClass{
   public int add(int a, int b) { return a + b; }
   public int subtract(int a, int b) { return a - b; }
...
}

public class SubClass extends SuperClass{
   public int add(int a, int b) { return a - b; }
   public int subtract(int a, int b) { return a + b; }
...
}
Cliente de SuperClass e SubClass
...
SuperClass supC = new SuperClass( );   // linha 01
int x = supC.add(7, 5);                      // linha 02
...
SubClass subC = new SubClass( );       // linha 03
x = subC.add(7, 5);                    // linha 04
...
supC = subC;                                 // linha 05
x = supC.add(7, 5);                    // linha 06
...
```

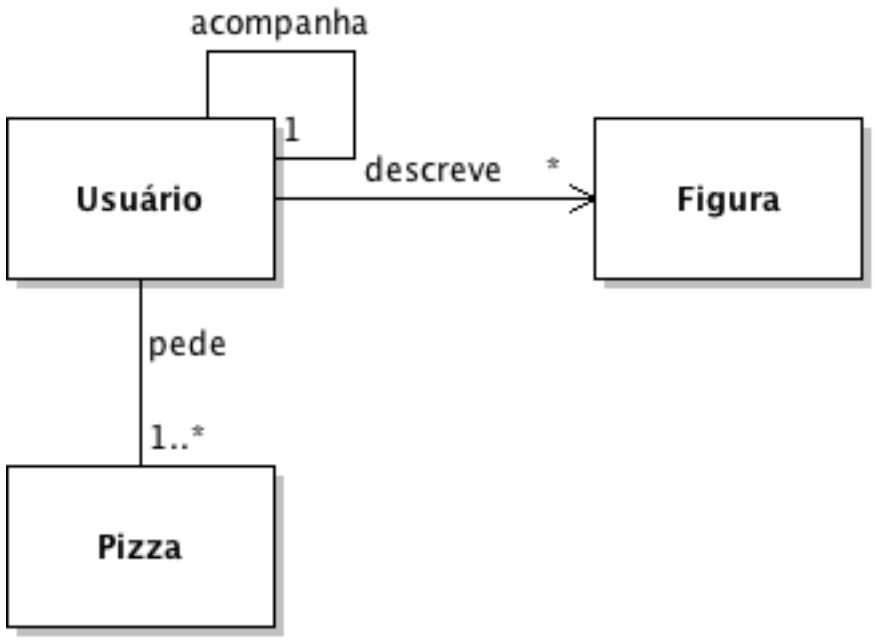
11. **[OBRIGATÓRIA]** Dado o diagrama de classes abaixo, faça o seguinte: /Total 2.5/
    a. Apresente o código arquitetural ou implícito da classe Usuário. /2.0/

Resp.:

```
Public class Usuario {
      public Usuario (Usuario usuarioAcompanhado, Pizza pizza) {
            associaUsuarioAcompanhado(usuarioAcompanhado);
            addPizza(pizza);
      }
      private void associaUsuarioAcompanhado(Usuario usuario){
            _usuarioAcompanhado = usuario;
      }
      public Usuario getUsuarioAcompanhado{
            return _usuarioAcompanhado;
      }
      public void addPizza(Pizza p) {
            _pizzas.add(p);
      }
      public void removePizza(Pizza p) {
            // Tem que garantir que sempre fica uma pizza na lista!
            // O certo seria lançar uma exceção: simplifiquei aqui!
            // Neste caso, o cliente do objeto não sabe se a pizza p
            // foi removida ou não!
            if (_pizzas.size() > 1)
                  _pizzas.remove(p);
      }
      public void getPizzas(){
            return Collections.unmodifiableList(_pizzas);
      }
```

```
        public void addFigura(Figura figura) {
        // Poderia ser descreveFigura(Figura figura)
              _figuras.add(figura);
        }
        public void removeFigura(Figura figura) {
              _figuras.remove(figura);
        }
        public void getFiguras(){
              return Collections.unmodifiableList(_figuras);
        }
        private Usuario         _usuarioAcompanhado;
        private List<Figura>    _figuras =  new LinkedList<Figura>();
        private List<Pizza>     _pizzas  =  new LinkedList<Pizza>();
}
```

b. Apresente o código arquitetural ou implícito da classe Figura. /0.5/
Resp.: Como a classe Figura não tem visibilidade da classe Usuario, não podemos assumir nenhum código implícito para ela.



Parte 3 – 2 Questões Envolvendo Implementação – /6.5/

12. **[OBRIGATÓRIA] [IMPLEMENTAÇÃO]** Criar usando TDD uma calculadora que efetue as operações de adição, subtração, multiplicação e divisão de reais, sabendo-se que um número não pode ser dividido por zero. Será verificado se o aluno utilizou o TDD corretamente, considerando corretamente todas as possibilidades e as incluindo nos testes. Para isso, cole aqui o código de produção e o de teste à medida que for desenvolvendo a calculadora; identifique por números crescentes começando por 1 e coloque os códigos de produção e de testes na ordem em que forem criados. Mas vou testar o código final com minha bateria de testes própria. Buscar apenas na Oracle ou JUnit informações sobre como testar exceção em Java a ser empregada nesta questão. A propósito, qualquer dúvida de codificação Java só pode ser sanada

com textos oficiais da Oracle ou JUnit. /3.0/

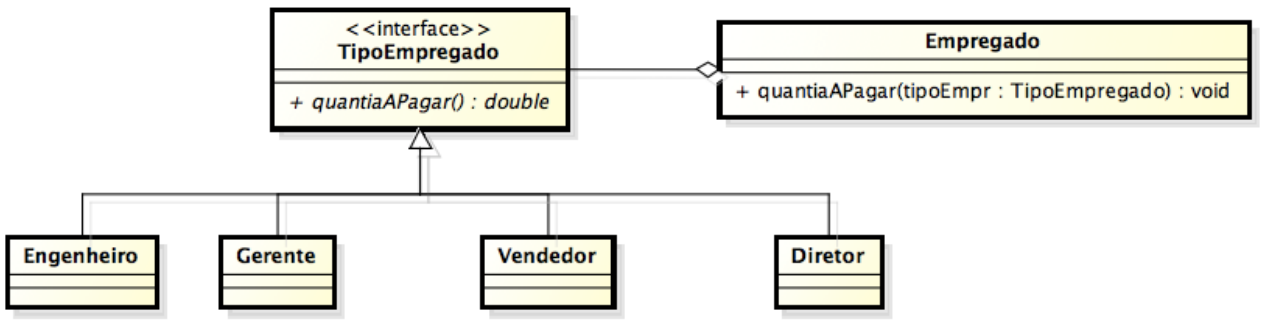
Resp.:

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalcTest {

	@Before
	public void setUp() throws Exception {
		calculator = new Calculator();
	}
	@Test
	public void quandoSomoZeroComZeroThenRetornaZero() {
		assertEquals(0+0,calculator.sum_1(0,0),0.0002);
	}
	public void quandoSomoZeroComOutroNumThenRetornaOutroNum() {
		assertEquals(0+1,calculator.sum_2(0,1),0.0002);
	}
	public void quandoSomoNumComZeroThenRetornaNum() {
		assertEquals(1+0,calculator.sum_2(1,0),0.0002);
	}
	public void quandoSomoNumComEleMesmoThenRetornaDobro() {
		assertEquals(1+1,calculator.sum_2(1,1),0.0002);
	}
		// Por simplicidade, junto todos os casos num teste só!
		// Cada caso de teste trata de uma propriedade da operação
	public void quandoSomoNumComOutroNumThenRetornaSomaDeles() {
		assertEquals(1+(-1),calculator.sum_2(1,-1),0.0002);
		assertEquals(-1+1,calculator.sum_2(-1,1),0.0002);
		assertEquals(-1+(-1),calculator.sum_2(-1,-1),0.0002);
		assertEquals(2+(-1),calculator.sum_2(2,-1),0.0002);
		assertEquals(-1+2,calculator.sum_2(-1,2),0.0002);
		assertEquals(1+(-2),calculator.sum_2(1,-2),0.0002);
		assertEquals(-2+1,calculator.sum_2(-2,1),0.0002);
	}
	@Test
	public void quandoSubtraioZeroDeZeroThenRetornaZero() {
		assertEquals(0,calculator.subtraction_3(0,0),0.0002);
	}
		// Idem para subtração: junto todos os casos num teste só!
		// Cada caso de teste trata de uma propriedade da operação
	@Test
	public void quandoSubtraioNumComOutroNumThenRetornaSubtracaoDeles() {
		assertEquals(0-(-1),calculator.subtraction_4(0,-1),0.0002);
		assertEquals(-1-0,calculator.subtraction_4(-1,0),0.0002);
		assertEquals(1-1,calculator.subtraction_4(1,1),0.0002);
		assertEquals(1-(-1),calculator.subtraction_4(1,-1),0.0002);
		assertEquals(-1-1,calculator.subtraction_4(-1,1),0.0002);
		assertEquals(-1-(-1),calculator.subtraction_4(-1,-1),0.0002);
		assertEquals(2-(-1),calculator.subtraction_4(2,-1),0.0002);
		assertEquals(-1-2,calculator.subtraction_4(-1,2),0.0002);
		assertEquals(1-(-2),calculator.subtraction_4(1,-2),0.0002);
```

```
            assertEquals(-2-1,calculator.subtraction_4(-2,1),0.0002);
      }
      @Test
      public void quandoMultiplicoZeroComZeroThenRetornaZero() {
            assertEquals(0*0,calculator.multiply_5(0,0),0.0002);
      }
            // Idem para multiplicação: junto todos os casos num teste só!
            // Cada caso de teste trata de uma propriedade da operação
      @Test
      public void
      quandoMultiplicoNumComOUtroNumThenRetornaMultiplicacaoDeles() {
            assertEquals(1*0,calculator.multiply_6(1,0),0.0002);
            assertEquals(0*1,calculator.multiply_6(0,1),0.0002);
            assertEquals(1*1,calculator.multiply_6(1,1),0.0002);
            assertEquals(2*2,calculator.multiply_6(2,2),0.0002);
            assertEquals(1*(-1),calculator.multiply_6(1,-1),0.0002);
            assertEquals(-1*1,calculator.multiply_6(-1,1),0.0002);
            assertEquals(-1*-1,calculator.multiply_6(-1,-1),0.0002);
      }
      @Test
      public void quandoDividoHumComHumThenRetornaHum() {
            assertEquals(1/1,calculator.divide_7(1,1),0.0002);
      }
            // Idem para divisão: junto todos os casos num teste só!
            // Cada caso de teste trata de uma propriedade da operação
      @Test
      public void quandoDividoNumComOutroNumThenRetornaDivisaoDeles() {
            assertEquals(0/1,calculator.divide_8(0,1),0.0002);
            assertEquals(1/-1,calculator.divide_8(1,-1),0.0002);
            assertEquals(-1/1,calculator.divide_8(-1,1),0.0002);
            assertEquals(-1/-1,calculator.divide_8(-1,-1),0.0002);
                  // 0/0 = NaN (Not-a-Number)
            assertEquals(Float.NaN,calculator.divide_8(0,0),0.0002);
                  // 1/0 = +Infinity

      assertEquals(Float.POSITIVE_INFINITY,calculator.divide_8(1,0),0.0002);
                  // -1/0 = -Infinity
            assertEquals(Float.NEGATIVE_INFINITY,calculator.divide_8(
                                                      -1,0),0.0002);
            // Estes três casos poderiam ter sido tratados com exceção divisão por zero,
            // mas assim fica mais simples com if/switch, sem precisar de try/catch
      }
      Calculator calculator;
   }
   -- ---  ----
public class Calculator {

      public float sum_1(float i, float j) {
            return 0;
      }
      public float sum_2(float i, float j) {
                  // Não estou tratando valores de contorno; por exemplo:
                  // Float.MAX_VALUE + Float.MAX_VALUE = Float.Positive_Infinity
            return i + j;
```

```
    }
    public float subtraction_3(float i, float j) {
        return 0;
    }
    public float subtraction_4(float i, float j) {
        // Não estou tratando valores de contorno; por exemplo:
        // -Float.MAX_VALUE - Float.MAX_VALUE = -Float.Positive_Infinity
        return i - j;
    }
    public float multiply_5(float i, float j) {
        return 0;
    }
    public float multiply_6(float i, float j) {
        // Não estou tratando valores de contorno; por exemplo:
        // Float.MAX_VALUE * 2 = Float.Positive_Infinity
        return i * j;
    }
    public float divide_7(float i, float j) {
        return 1;
    }
    public float divide_8(float i, float j) {
        // Não estou tratando valores de contorno; por exemplo:
        // -Float.MAX_VALUE * 2 = Float.Negative_Infinity
        // Não estou tratando divisão por zero com exceção:
        // deixo para quem chamou tratar resultado como normal, NaN ou +/-
        // Infinity!
        return i / j;
    }

}
```

13. **[OBRIGATÓRIA]** Considerando os códigos abaixo das classes Empregado, Engenheiro, Gerente, Vendedor e Diretor, e a interface TipoEmpregado, realize as tarefas descritas abaixo: /3.5/
    a. Crie o diagrama de classes UML correspondente que represente as associações entre essas classes e a interface. /1.0/
    Resp.:



    b. Qual o nome do conceito de orientação objeto usado no código em amarelo da classe Empregado? /0.5/
    Resp.: Delegação
    c. Considerando o código abaixo da classe Diretor, informe o que será

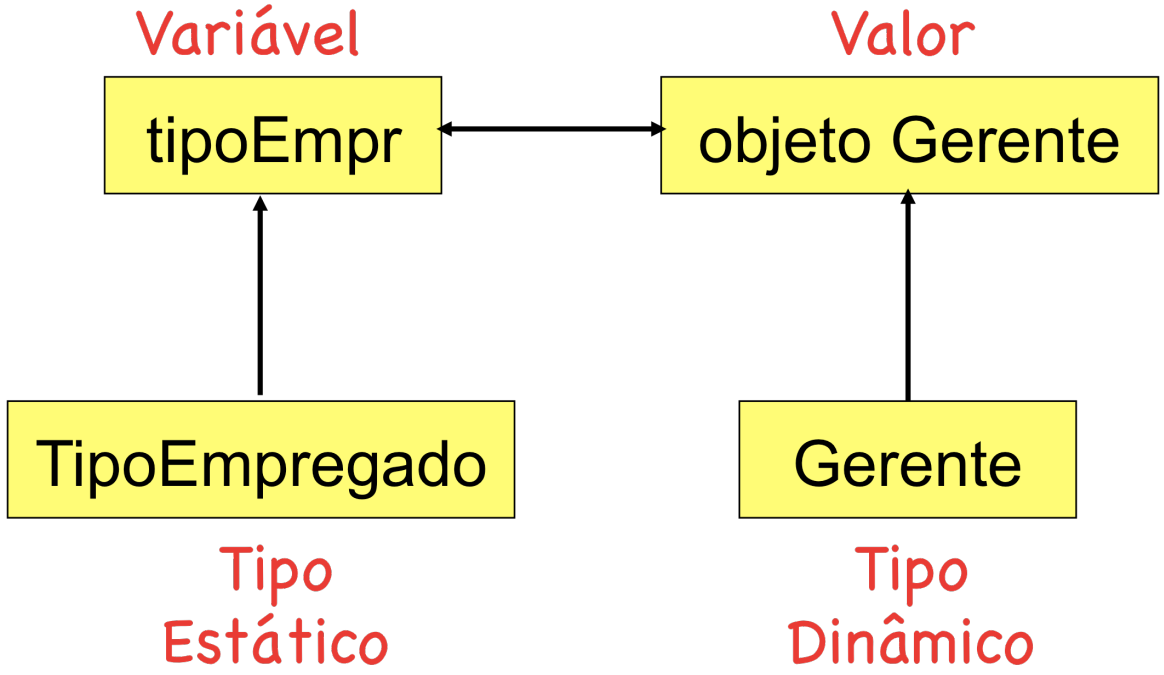
impresso no main() da classe TesteFuncionario apresentada abaixo também. Para oferecer a resposta a esta questão, a classe TesteFuncionario não deve ser implementada no Eclipse ou outro ambiente Java e a resposta deve ser dada antes do item e) ser respondido. /0.5/

Resp.:

```
200.0
500.0
```

d. Desenhe o diagrama que mostre o tipo estático e o tipo dinâmico da variável tipoEmp da classe TesteFuncionario. /0.5/

Resp.:



e. **[IMPLEMENTAÇÃO]** Implemente as classes e interface apresentadas nesta questão no Eclipse ou outro ambiente Java, exceto a classe TesteEmpregado. Desenvolva, teste e apresente (cole) aqui uma bateria de testes JUnit para testar o funcionamento do método quantiaAPagar() das classes Engenheiro, Gerente, Vendedor e Diretor. Evite repetição de código no Junit. /1.0/

Resp.:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TestFuncionario {

	@Before
	public void setUp() throws Exception {
		empregado = new Empregado();
		salario = 100;
```

```
        vendas = 100;
    }
    @Test
    public void testQuantiaAPagarDiretor() {
        tipoEmpr= new Diretor(salario, vendas);
        double expected = salario+vendas*0.4;
        assertEquals(expected, empregado.quantiaAPagar(tipoEmpr),0.00001);
    }
    @Test
    public void testQuantiaAPagarEngenheiro() {
        tipoEmpr= new Engenheiro(salario);
        double expected = salario;
        assertEquals(expected, empregado.quantiaAPagar(tipoEmpr),0.00001);
    }
    @Test
    public void testQuantiaAPagarGerente() {
        tipoEmpr= new Gerente(salario, vendas);
        double expected = salario+vendas*0.1;
        assertEquals(expected, empregado.quantiaAPagar(tipoEmpr),0.00001);
    }

    @Test
    public void testQuantiaAPagarVendedor() {
        tipoEmpr= new Vendedor(vendas);
        double expected = vendas+vendas*0.2;;
        assertEquals(expected, empregado.quantiaAPagar(tipoEmpr),0.00001);
    }

    Empregado         empregado=null;
    TipoEmpregado     tipoEmpr=null;
    double                  salario=0;
    double                  vendas=0;
}
```

```
public class Empregado {
   public double quantiaAPagar(TipoEmpregado tipoEmpr) {
                return tipoEmpr.quantiaAPagar();
        }
}
```

```
public  interface TipoEmpregado {
                public abstract double  quantiaAPagar();

}
```

```
public class Engenheiro implements TipoEmpregado {
        private double salario=0;

        Engenheiro (double salario){
                this.salario=salario;
        }
```

```
        public double quantiaAPagar() {
                return salario;
        }
}
```

```
public class Gerente implements TipoEmpregado{
        private double salario=0;
        private double vendas=0;

        Gerente(double salario, double vendas){
                this.salario=salario;
                this.vendas=vendas;
        }

        public double quantiaAPagar() {
                return salario+vendas*0.1;
        }
}
```

```
public class Vendedor implements TipoEmpregado{
        private double vendas=0;

        Vendedor (double vendas){
                this.vendas=vendas;
        }

        public double quantiaAPagar() {
                return vendas+vendas*0.2;
        }
}
```

```
public class Diretor implements TipoEmpregado{
        private double salario=0;
        private double vendas=0;

        Diretor(double salario, double vendas){
                this.salario=salario;
                this.vendas=vendas;
        }

        public double quantiaAPagar() {
                return salario+vendas*0.4;
        }
}
```

```
public class TesteFuncionario {
        // Esta classe não deve ser implementada no ambiente Java
   public static void main (String args[]){
        TipoEmpregado tipoEmpr = new Gerente(100.0, 1000.0);
        Empregado emp = new Empregado();

        System.out.println(emp.quantiaAPagar(tipoEmpr));
        System.out.println(emp.quantiaAPagar(new Diretor(100.0, 1000.0)));
  }
}
```