

CES-28 Prova 1 - 2017

Sem consulta - individual - com computador - 3h

PARTE I - ORIENTAÇÕES

1. Qualquer dúvida de codificação Java só pode ser sanada com textos/sites oficiais da **Oracle** ou **JUnit**.
 - a. Exceção são idiomas (ou 'macacos') da linguagem como sintaxe do método `.equals()`, ou sintaxe de set para percorrer collections, não relacionados ao exercício sendo resolvido. Nesse caso, podem procurar exemplos da sintaxe na web.
2. Sobre o uso do **Mockito**, com a finalidade de procurar exemplos da sintaxe para os testes, podem ser usados sites de ajuda online, o próprio material da aula com (pdf's, exemplos de código e labs), assim como o seu próprio código, **mas sem usar código de outros alunos**.
 - a. Lembre-se de configurar seu build com os jar(s) existentes em **bibliotecas.zip**.
3. Questões com itens diversos, favor identificar claramente pela letra que representa o item, para que seja possível saber precisamente a que item corresponde a resposta dada!
 - a. **NO CASO DE NÃO IDENTIFICAÇÃO, A QUESTÃO SERÁ ZERADA,**
4. Se necessário realizar implementação, somente serão aceitos códigos implementados no Eclipse. Essas questões ou itens serão indicados com o rótulo **[IMPLEMENTAÇÃO]**! Para as outras questões, pode ser usado o Eclipse, caso for mais confortável, digitando os exemplos, mas não é necessário um código completo, executando. Basta incluir trechos do código no texto da resposta.
5. Deve ser submetido tanto no TIDIA, como no GITLAB os seguintes entregáveis:
 - a. **QUESTÕES DE IMPLEMENTAÇÃO:** Código completo e funcional da questão, bem como todas as bibliotecas devidamente configuradas nos seus respectivos diretórios. O projeto deve SEMPRE ter um "source folder" **src** (onde estarão os códigos fontes) e outro **test**, onde estarão as classes de testes, caso seja o caso. Cada questão de implementação deve ser um projeto à parte.
 - b. **DEMAIS QUESTÕES:** Deve haver ser submetido um arquivo PDF com as devidas respostas. Use os números das questões para identificá-las.
6. No caso de diagramas, pode ser usado qualquer editor de diagrama UML, assim como desenhar no papel, tirar a foto, e **incluí-la no pdf dentro da resposta, NÃO como anexo separado. Atenção: use linhas grossas, garanta que a foto é legível!!!!**

PARTE II - CONCEITUAL

QUESTÃO 1 - CIRCLE X ELLIPSE

Dado que a classe ELLIPSE é pai da classe CIRCLE (*faz sentido, porque círculos são elipses*), a classe CIRCLE pode reusar todo o conteúdo da classe ELLIPSE, bastando para isso apenas sobrescrever os métodos, visando garantir que os eixos maior e menor permaneçam iguais. No entanto, o método

void Ellipse.stretchMaior() // “estica” a elipse na direção do eixo maior

não funciona com CIRCLE, pois o resultado deixa de ser um círculo.

Não é possível fazer CIRCLE pai de ELLIPSE, pois seria conceitualmente errado, já que nem toda ELLIPSE é um CIRCLE. Analise, o que aconteceria se uma função espera um círculo e recebe uma elipse?

Além disso, o método *double Circle.getRadius()* não faz sentido com uma elipse.

- a) Explique este dilema conceitualmente, usando para isso apenas os conceitos e vocabulários constantes de POO, especialmente àqueles relacionados a responsabilidade e herança. **(1.0 PT)**

R: Como ELLIPSE é pai da classe CIRCLE, ocorre o fenômeno da **herança**, isto é, a classe CIRCLE herda o conteúdo de ELLIPSE, bastando para ser coerente **sobrescrever** os métodos convenientes para igualar o eixo maior ao eixo menor. Por conta da existência do método *void Ellipse.stretchMaior() // “estica” a elipse na direção do eixo maior*, tal método geraria problemas se fosse herdado tal como é por CIRCLE, uma vez que não faz sentido o círculo ser “esticado” na direção do eixo maior, passando a ser uma elipse. Além disso, se invertêssemos a relação

pai-filho, isto é, CIRCLE passar a ser classe pai de ELLIPSE, incorreremos em uma relação conceitualmente errada, e além disso, manteríamos o problema de um método da classe pai não fazer sentido quando usado pela classe filha, como é o caso de *double Circle.getRadius()*. Sendo assim, gera-se um problema envolvendo os conceitos de **responsabilidades e herança**, afinal nenhuma classe, nem ELLIPSE nem CIRCLE, está inteiramente “contida” na outra. Uma classe filha deveria herdar todas as responsabilidades da classe pai, o que torna esse problema um dilema, pois, em ambos os casos explicitados acima, ao menos uma responsabilidade seria “perdida” pela classe filha em relação à classe pai.

- b) Forneça uma solução que ainda promova o reuso de código. A sua solução pode ter uma desvantagem, no ponto de vista do programador que usa as suas classes. Explique-a conceitualmente a solução e a desvantagem, usando o vocabulário de POO, do ponto de vista do programador que usa as suas classes. **(1.0 PT)**

R: A classe CIRCLE deve continuar sendo **subclasse** de ELLIPSE. Além da **sobrescrita** dos métodos a fim de igualar para CIRCLE os valores do eixo maior e do eixo menor, deve-se sobrescrever também o método *void Ellipse.stretchMaior()*, (**@Overriding**), podendo funcionar como opção um método que aumente “simultaneamente” os valores do eixo maior e do eixo menor, de modo a figura permanecer um círculo, e não se transformar em uma elipse. A desvantagem presente é que deixamos de obedecer aos princípios SOLID, quando sobrescrevemos o método *Ellipse.stretchMaior()*, adicionando uma outra funcionalidade de também esticar o eixo menor.

OBS: NÃO SÃO NECESSÁRIAS IMPLEMENTAÇÕES COMPLETAS, APENAS DEVE-SE USAR TRECHOS DE CÓDIGO NA RESPOSTA QUANDO RELEVANTE.

QUESTÃO 2 - TDD

Dado as sentenças abaixo, marque V para àquelas que são verdadeiras, ou F para as falsas. **(1.0 PT)**

[**V**] Podemos dizer que o exemplo a seguir é um bom exemplo de TDD?

Recebemos um código legado bastante grande de um projeto anterior, desenvolvido sem nenhum teste, e refatoramos o mesmo, criando testes. É iniciado pelo desenvolvimento de testes triviais, passando por testes simples, testes de unidade, até chegar em testes maiores, com o objetivo de nos certificarmos de que o código funciona e posteriormente permitir a evolução e manutenção desse código.

[**V**] TDD supõe uma serie de ferramentas de desenvolvimento. A comparação do TDD versus um desenvolvimento não-TDD seria muito menos favorável se não existissem ferramentas e IDEs "bonitinhas" para automatizar testes, inclusive facilitar a leitura dos resultados dos mesmos, verificar rapidamente o que passou e não passou, facilitar inclusive varias refatoracoes comuns, e ferramentas de diff e controle de versão para reverter eventuais erros e/ou encontrar as últimas mudanças com data e responsável. Inclusive podemos considerar isso como uma das razoes porque o TDD demorou algumas décadas para aparecer, e não apareceu nos primórdios da computação.

[**F**] Refatorações no TDD são relativamente infrequentes, acontecem apenas quando é detectado algum erro que deve ser corrigido. Uma refatoração é sempre retrabalho e o resultado de algum erro humano.

[**V**] Há alguns casos limite tão comuns que praticamente sempre devemos testar pelo menos vários deles, especialmente quando se usam estruturas de dados. Caso vazio, cheio, apenas um elemento, ultimo e primeiro, usar o índice zero versus índice 1, etc. Para algumas estruturas de dados, pode também ser importante testar os casos de número de elementos par e ímpar, ou entrada ordenada e desordenada. Quando se implementa uma pilha, por exemplo, testar pelo menos algumas dessas condições deve ser um reflexo automático para o programador TDD.

PARTE III - IMPLEMENTAÇÃO

[IMPLEMENTAÇÃO] – QUESTÃO 3 UM BAR COM MAU CHEIRO.

Abra o projeto Pub.java, e execute os testes. Nesse projeto existe uma série de mal cheiros e problemas de responsabilidades.

IMPORTANTE - NUNCA MUDE OS TESTES! ELES DEVEM CONTINUAR FUNCIONANDO!

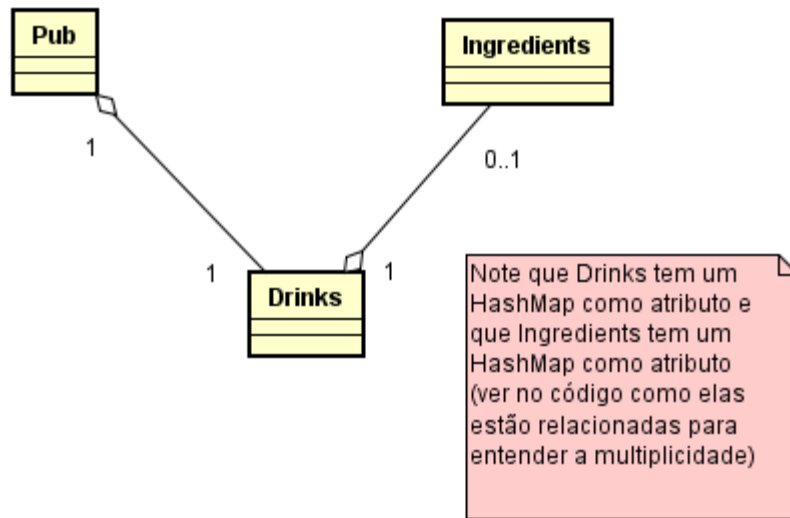
- a) Refatore o código, criando novas classes de forma a dividir melhor as responsabilidades. **(3.0 PT)**
 - i) Uma tarefa comum de manutenção deste código seria *incluir e remover ingredientes e drinks no modelo, ou modificar as regras em relação aos já existentes.*

R: IMPLEMENTADO EM CÓDIGO.

- b) A sua solução deve facilitar a tarefa de manutenção descrita em *a-i* e ainda continuar provendo o reuso de código. Considerando o requisito apresentado no item *a-i*, explique como a manutenibilidade e reuso são promovidos pela sua solução. **(0.5 PT)**

R: Como o código com mau cheiro estava cheio de particularidades, a manutenibilidade e o reuso do código estavam comprometidas, isto é, seria necessária uma grande reestruturação a cada funcionalidade que desejasse ser implementada. Após a refatoração ficou bem simples: Com o uso de HashMaps, tanto para os ingredientes quanto para os drinks, é bem trivial adicionar ou remover drinks e ingredientes, por exemplo, pelos próprios métodos de HashMap. Além disso, o código se tornou de fáceis manutenção e reaproveitamento dado que foram criadas classes mais “especialistas” nas suas responsabilidades. Agora é possível incrementar funcionalidades no código por meio da manutenibilidade e reuso promovidos, além da maior clareza e maior proteção geradas na refatoração.

- c) Escreva um diagrama UML representando a sua solução, considerando as classes, associações e tipos de associações (agregação/associação/composição), bem como multiplicidades. **(0.5 PT)**



R:

PS: descontos são arredondados para o próximo inteiro maior - *Math.ceil()* resolve o arredondamento.

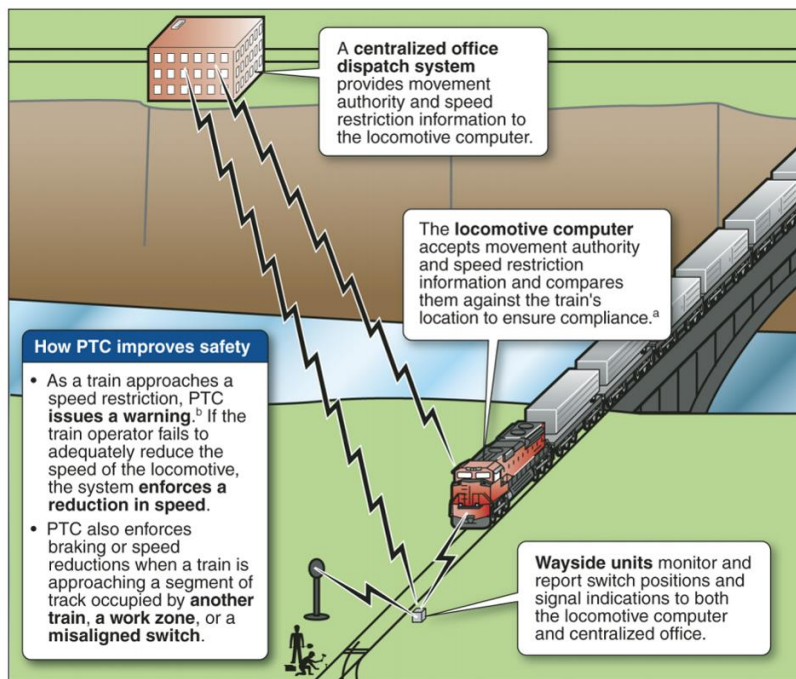
[IMPLEMENTAÇÃO] – QUESTÃO 4 –CONTROLE POSITIVO DE TRENS.

Considere um sistema de Controle positivo de trens (*Positive Train Control - PTC*). Um PTC requer a coleta e a ação em 2 tipos de informações:

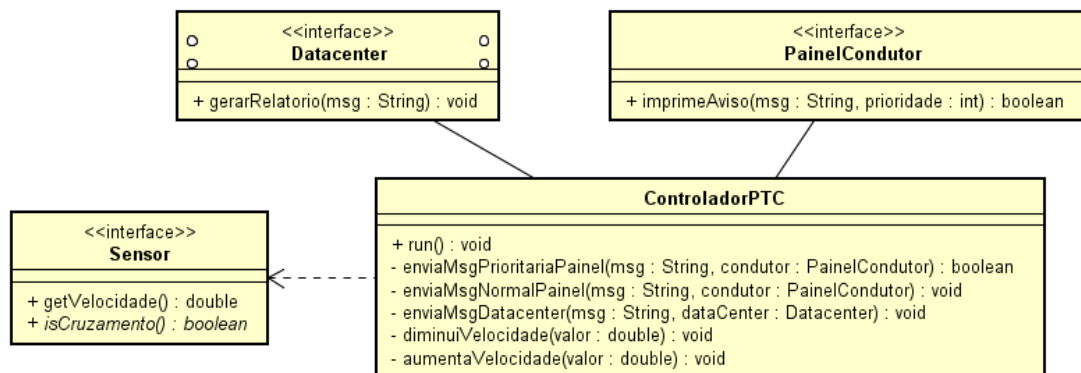
- Dados urgentes que devem ser acionados imediatamente; e
- Dados enviados para o datacenter para serem mensurados e utilizados posteriormente.

Para esse desenvolvimento, **sensores de trilhos** coletam e registram dados sobre a rota, a velocidade e as características de carga dos trens. Todos os dados passam pela **camada de controle**, onde o software de mensagens e regras de negócios determina o que fazer com os dados. Conforme o trem se aproxima de um cruzamento, as mensagens para alterar a velocidade são transmitidas para o **painel do condutor com alta prioridade**. Informações com menos urgência, sobre velocidade, eficiência de combustível, peso e outras são armazenadas no **datacenter**

para serem analisadas. **Caso essas direções urgentes sejam ignoradas, ações automáticas entram em ação** no **sistema de bordo do trem** para parar, diminuir ou acelerar a velocidade do mesmo. Colisões são evitadas e os dados são armazenados de maneira segura.



Source: GAO.



O diagrama de classe que implementa o supracitado sistema é apresentado na figura acima. Abaixo é apresentado o código que implementa o ControladorPTC.

```

package Q4.ptc;

import java.util.concurrent.TimeUnit;

public class ControladorPTC {
    private Sensor sensor;
    private Datacenter dataCenter;
    private PainelCondutor painelCond;

    public ControladorPTC(Sensor sensor, Datacenter dataCenter, PainelCondutor painelCond) {
        super();
        this.sensor = sensor;
        this.dataCenter = dataCenter;
        this.painelCond = painelCond;
    }

    public void run() {

        double velocidade = sensor.getVelocidade();
        boolean isCruzamento = sensor.isCruzamento();

        // cheça se o trem esta com velocidade acima do permitido no cruzamento
        if (isCruzamento && (velocidade > 100)) {
            boolean result = enviaMsgPrioritariaPainel("Velocidade alta", painelCond);
            if (result == false) {
                diminuiVelocidade(20);
            }
        }

        // cheça se o trem esta lento demais no cruzamento
        if (isCruzamento && (velocidade < 20)) {
            boolean result = enviaMsgPrioritariaPainel("Velocidade Baixa", painelCond);
            if (result == false) {
                aumentaVelocidade(20);
            }
        }

        else {
            enviaMsgDatacenter(new Double(velocidade), dataCenter);
            enviaMsgNormalPainel(new Double(velocidade), painelCond);
        }
    }
}
  
```



```

    }

    }

    public boolean enviaMsgPrioritariaPainel(String msg, PainelCondutor condutor) {
        boolean result = condutor.imprimirAviso(msg, 1);
        if (result == false) {
            try {
                TimeUnit.SECONDS.sleep(10);
                result = condutor.imprimirAviso(msg, 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return result;
    }

    public void enviaMsgNormalPainel(Object msg, PainelCondutor condutor) {
        condutor.imprimirAviso(msg.toString(), 1);
    }

    public void enviaMsgDatacenter(Object msg, Datacenter datacenter) {
        datacenter.gerarRelatorio(msg.toString());
    };

    public void diminuiVelocidade(double valor) {
        this.painelCond.diminuiVelocidadeTrem(valor);
    };

    public void aumentaVelocidade(double valor) {
        this.painelCond.aceleraVelocidadeTrem(valor);
    };
}

```

Através do uso de Test Double e do uso do Framework Mockito, resolva as questões abaixo:

- Teste a inicialização do objeto **ControladorPTC**. **(1.0 PT)**.
- Construa um caso de teste, quando o trem não se encontra em um cruzamento, ou seja, o método ***isCruzamento()*** de **Sensor** retorna falso. Verifique o comportamento se deu certo. **(1.0 PT)**.
- Construa um caso de teste, quando o trem se encontra em um cruzamento e a velocidade é superior 100Km/h, ou seja, o método ***isCruzamento()*** de **Sensor** retorna verdadeiro. Além disso, o usuário localizado no Painel do Condutor deve informar que leu a mensagem, ou seja, o retorno do método ***enviaMsgPrioritariaPainel()*** deve ser verdadeiro. Verifique o comportamento se deu certo. **(1.0 PT)**.
- Construa um caso de teste, quando o trem se encontra em um cruzamento e a velocidade é inferior a 20Km/h, ou seja, o método ***isCruzamento()*** de **Sensor** retorna verdadeiro. Além disso, o usuário localizado no Painel do Condutor não deve confirmar a leitura da mensagem, ou seja, o retorno do método ***enviaMsgPrioritariaPainel()*** deve ser falso. Verifique o comportamento se deu certo. **(2.0 PT)**.

- a. **Observação A:** Deve ser usar Test Double nas classes não relacionadas ao comportamento do Controlador.
- b. **Observação B:** Na correção será considerado que aderência e pertinência do Test Double selecionado.
- c. **Observação C:** Será avaliado a pertinência e cobertura dos casos de testes realizados, ou seja, devem ser construídos casos de testes para cada uma das funcionalidades do CDS apresentadas no cenário acima apresentado.
- d. **Observação D:** Um melhor detalhamento do cenário pode ser encontrado em: <https://www.forbes.com/sites/hilarybrueck/2015/05/20/how-positive-train-control-works-how-it-could-make-rail-travel-safer/#722452ac7e9d>