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 ómacacosô) 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 ñsource folderò **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 legivel!!!!**

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() // ñesticaòa elipse na direcao 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)**

Pois, apesar de todo círculo ser uma elipse, existem métodos de uma elipse que não existem em um círculo, como por exemplo, "void Ellipse.stretchMaior()". Como todo método de uma classe pai é herdada na classe filha, uma opção seria sobrescrever o método para sempre gerar uma exceção, mas que fere o princípio do desenvolvimento em POO, pois, independente do círculo, o método retornaria uma exceção que deveria ser tratada. Portanto, para manter cada classe em sua

responsabilidade, não se deve fazer uma herança simples pai/filho entre um círculo e uma elipse.

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)**

```
public abstract class ElipseType {
	protected double _eixoMaior;
	protected double _eixoMenor;

	public ElipseType(double eixoMaior, double eixoMenor)
	{
		_eixoMaior = eixoMaior;
		_eixoMenor = eixoMenor;
	}

	public double getEixoMaior() {
		return _eixoMaior;
	}

	public double getEixoMenor() {
		return _eixoMenor;
	}

	public double getArea()
	{
		return Math.PI*_eixoMaior*_eixoMenor;
	}
}

public class Circle extends ElipseType{
	public Circle(double radius)
	{
		super(radius, radius);
	}

	public double getRadius()
	{
		return _eixoMaior;
	}

	public void setRadius(double radius)
	{
		_eixoMaior = radius;
		_eixoMenor = radius;
	}
}


public class Elipse extends ElipseType{

	public Elipse(double eixoMaior, double eixoMenor)
	{
		super(eixoMaior, eixoMenor);
	}

	public void setEixoMaior(double eixo)
	{
		_eixoMaior = eixo;
	}


	public void setEixoMenor(double eixo)
	{
		_eixoMenor = eixo;
	}
}
```

A solução consiste em criar uma classe abstrata ElipseType que seja pai das classes Elipse e Circle. Assim, em ElipseType ficam métodos e variáveis comuns a Elipse e Circle, promovendo reuso de código, e nas classes Elipse e Circle os métodos e variáveis específicos de cada um deles. A principal desvantagem desse modelo é que, apesar de todo círculo ser uma elipse, não se pode fazer a conversão:

```
Circle teste = new Circle(2);
Elipse teste1 = (Elipse) teste;
```

Em que uma possível solução seria:

```
Circle teste = new Circle(2);
ElipseType teste1 = (ElipseType) teste;
Elipse teste2 = (Elipse) teste1;
```

Sendo ainda sim uma solução ruim. Além disso, funções que esperam elipses e círculos como argumentos teriam que esperar um objeto ElipseType.

QUESTÃO 2 - TDD

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

[F] 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. È 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] ï 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!

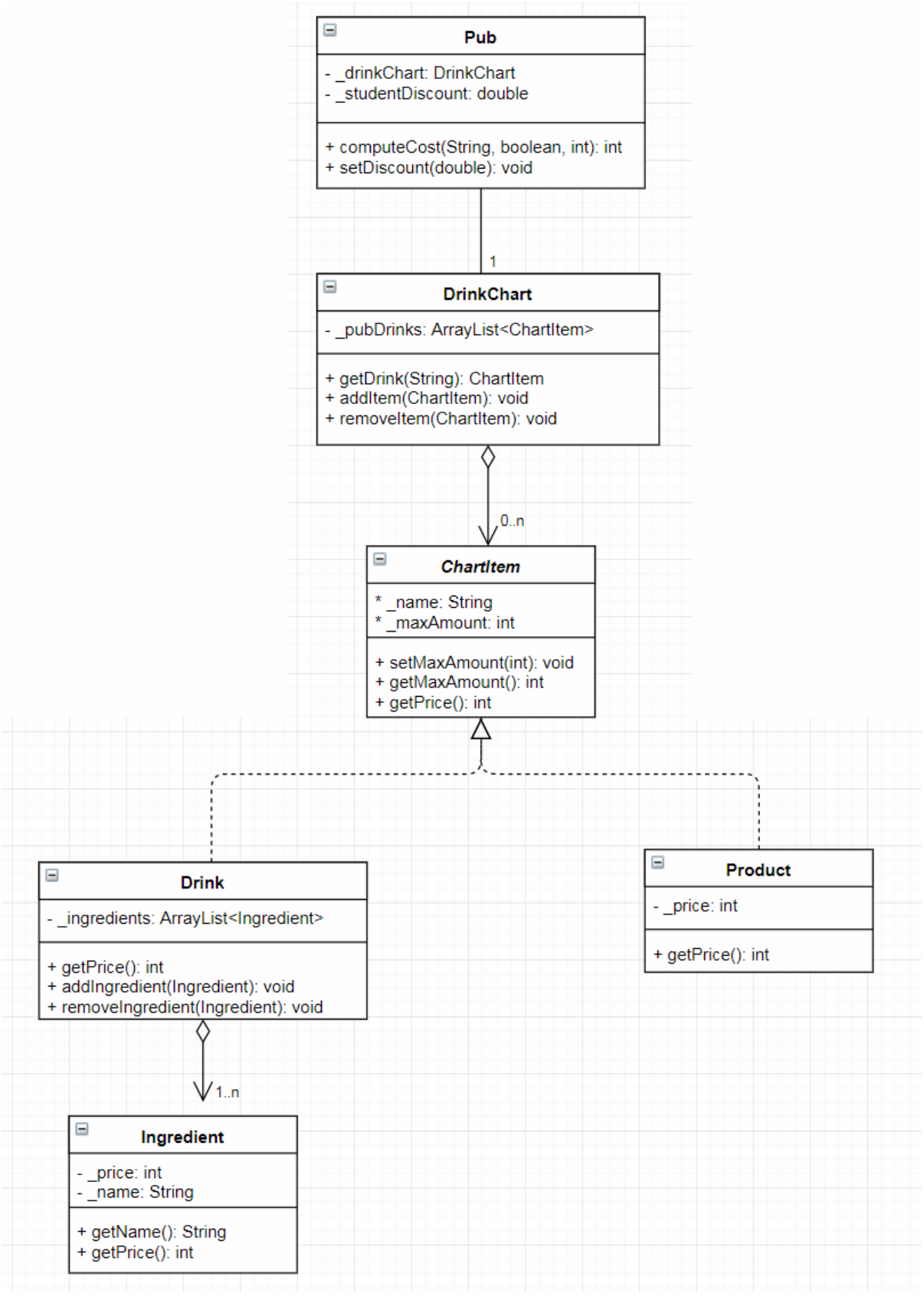
a) Refatore o código, criando novas classes de forma a dividir melhor as responsabilidades. **(3.0 PT)**
   i) Uma tarefa comum de manutenção deste código seria *incluir e remover ingredientes e drinks no modelo, ou modificar as regras em relação aos já existentes.*

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)**

A minha solução utiliza dos conceitos de POO como a Single Responsability para que o código seja reusado e corrigido facilmente. Para alterar os ingredientes do drink, existem os métodos addIngredient e removeIngredient, e, como Ingredient é uma classe, vários drinks podem usar o mesmo objeto instanciado, promovendo o reuso de código. Também pode-se facilmente remover itens do DrinkChart com os métodos addItem e remove Item.

Além disso, a solução diferencia Drinks de Products, pois a classe Product não tem ingredientes, além de ser a única em que é oferecida desconto a estudantes. Portanto, ao se adicionar mais Drinks ou Produtos ao DrinkChart, só precisa se verificar se o ChartItem retornado por getDrink é instance de Product para aplicar o desconto aos estudantes.

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)**

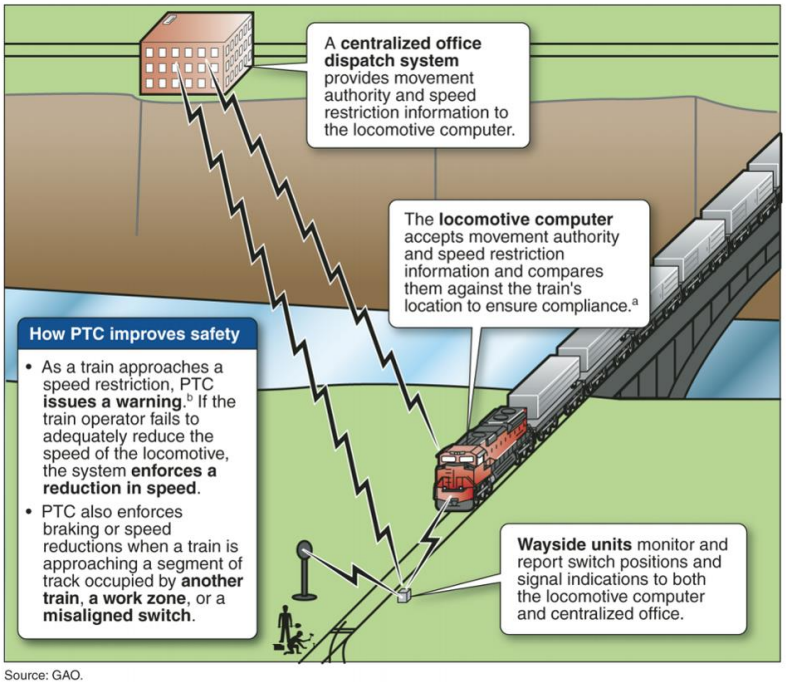

Pub
- _drinkChart: DrinkChart
- _studentDiscount: double
+ computeCost(String, boolean, int): int
+ setDiscount(double): void
1
DrinkChart
- _pubDrinks: ArrayList<ChartItem>
+ getDrink(String): ChartItem
+ addItem(ChartItem): void
+ removeItem(ChartItem): void
0..n
ChartItem
* _name: String
* _maxAmount: int
+ setMaxAmount(int): void
+ getMaxAmount(): int
+ getPrice(): int
Drink
- _ingredients: ArrayList<Ingredient>
+ getPrice(): int
+ addIngredient(Ingredient): void
+ removeIngredient(Ingredient): void
1..n
Ingredient
- _price: int
- _name: String
+ getName(): String
+ getPrice(): int
Product
- _price: int
+ getPrice(): int

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

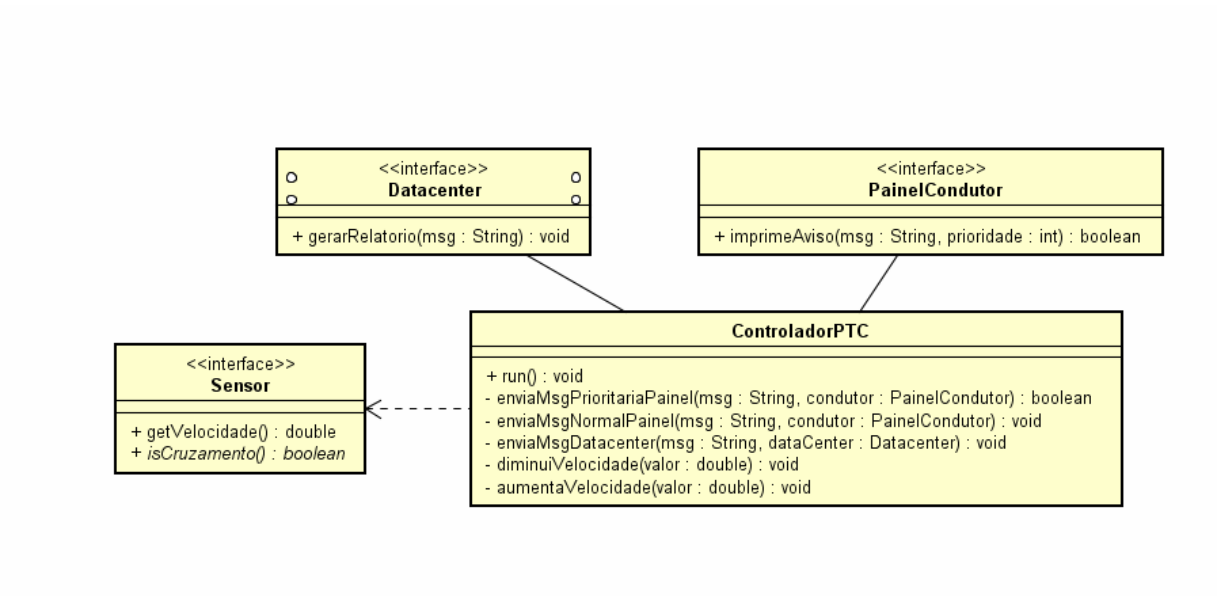
Considere um sistema de Controle positivo de trens (*Positive Train Control - PTC*). Um PTC requer a coleta e a ação em 2 tipos de informações:

a) Dados urgentes que devem ser acionados imediatamente; e
b) 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();

		// checa 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);
			}
		}

		// checa 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 Mokito, resolva as questões abaixo:

a) Teste a inicialização do objeto **ControladorPTC**. **(1.0 PT).**
b) 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).**
c) 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).**
d) 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 nãp 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