CES-28 Prova 2 – 2017

Nome: Dylan Nakandakari Sugimoto / COMP-19

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 - Demeter não é só procurar por vários '.' na mesma linha.

[livro The Pragmatic Programmer cap 5]

```
1 void processTransaction(BankAccount acct, int value) {
        2 Person *who;
        3 Money amt;
        4 amt.setValue(value);
        5 acct.setBalance(amt);
        6 who = acct.getOwner();
        7 // saves log of all transactions
        8 logService(who->name(), SET_BALANCE);
}
```

a) O código acima contém uma violação da Lei de Demeter. Encontre-a, e explique porque é uma violação [0.5].

Resposta:

Na linha 6 e na linha 8 (que na verdade é consequência da linha 6), em que está destacado em amarelo, ocorre o acesso de um objeto que não é parâmetro da classe ou do método, ou criado dentro do método nem é global, ou seja, ocorre um acesso a um objeto que não devia ocorrer ou pelo menos pode-se dizer desnecessário e perigoso por possibilitar alterações indevidas por um objeto que não ter permissão para realizar tal ação. O melhor seria se "who" recebesse direto de "acct" o nome do dono da conta já o método só necessita disso.

b) Corrija o código acima (não precisa do código completo) [0.5].

   a. Obs.: Não precisa mudar outras classes explicitamente, apenas indique o que precisaria ser mudado. Por exemplo: "a classe Money precisa de um novo método *getMoneys()*", sem precisar implementar *getMoneys()*.

      Resposta:

      Como citado anteriormente, seria melhor se "acct" retornasse o nome do dono da conta. Assim, para isso é necessário escrever um método do tipo

"getOwnerName" na classe BankAccount. E talvez remover esse método "getOwner" se ele só for útil nesse método do exemplo nessa classe.

c) Depois da correção, indique qual dependência entre quais classes foi eliminada pela sua correção e explique porque melhorou a manutenibilidade, fornecendo um exemplo de mudança em uma das entidades do projeto que não implicaria mais em mudança no código acima [0.5].

Reposta:

Considerando que nenhum outro método tinha acesso ao dono da conta, então, a classe em questão não tem mais associação com a classe Person que é o dono da conta. E o BankAccount trata de pegar o nome do dono da conta o que significa que uma mudanças no Person afeta apenas BankAccount que pode tratar as mudanças de forma a não afetar a classe em questão. Isso é benéfico para manutenibilidade do código. Por exemplo, se fosse criado uma outra classe que também pudesse ter conta no banco, por exemplo, uma empresa. Seria necessário apenas realizar esse tratamento de natureza de classe no BankAccount, e usar algum método de acesso ao nome dessa nova classe para pegar o nome do dono da conta e retorna esse nome como string se for o caso da classe em questão receber em string.

d) Explique a melhoria realizada acima em relação a pelo menos um dos princípios SOLID ou GRASP [0.5].

No SOLID, associa-se diretamente com o conceito de Single Responsability, pois não é da responsabilidade da classe em questão saber quem a pessoa que é dono da conta, ela apenas pergunta para a classe BankAccount e BankAccount que é responsável de saber responder o nome para a classe em questão.
High Coesion do GRASP é muito semelhante ao Single Responsability do SOLID, mas também podemos citar o Low Coupling já que diminuímos o acoplamento entre as classes ao eliminar a associação entre Person e a classe em questão.

QUESTÃO 2 - TDD

Dada as questões abaixo, marque a opção correta:

a) Ha um conjunto de requisitos e casos de teste a serem implementados em um projeto TDD. Durante o ciclo do TDD, percebemos que há um caso limite ou exceção simples de

implementar, que não foi previsto, e é importante e necessário para que tudo funcione. O que fazemos? (0.5)

I. Não podemos adicionar funcionalidades, mesmo que pequenas. Precisamos recombinar as funcionalidades e casos previstos com o chefe ou o resto do grupo.

   Reposta: Errado. Podemos adicionar funcionalidades pequenas que não necessitam de reestruturação na ordem da implementação e de testes das funcionalidade por causa dessa adição.

II. Adicionamos mais um caso de teste na lista "to do", e o TDD continua normalmente. Esse novo teste será implementado como qualquer outro durante a fase RED do TDD.

   Resposta:
   Certo. Na fase RED fazemos os testes para testar essa exceção ou limite.

III. Necessariamente precisamos incluir esse caso nos testes já existentes, através de asserts extra.
   Resposta:
   Errado. Fazemos novos testes, pois isso se refere a um novo caso de teste.

IV. Implementamos esse novo caso na fase BLUE do TDD, ou seja, considerando-o como uma refatoração, pois não existe teste para ele.
   Resposta:
   Errado, os testes são feitos na fase RED.

b) O TDD indica que não misturemos as fases RED, GREEN, e BLUE, e que hajam varias iterações do mesmo ciclo. Um dos propósitos disso eh fazer com que a qualquer momento no ciclo TDD, apenas um (o novo teste) ou pelo menos, poucos testes estejam falhando ou com erro. Ou seja, fazer com que o código de produção permaneça relativamente próximo de "compilando, executando e 100% *green tests*", mesmo durante o desenvolvimento. A não ser em casos raros em que um bug ou refatoração muito fundamentais quebrem momentaneamente muitos testes, gostaríamos que isso fosse sempre verdade. Consideramos que isso ajuda a manter a qualidade do software no TDD. (0.5)

   I. V (x) . Embora seja melhor que sempre que possível fazer o código implementado passar em todos testes, uma vez que uma boa técnica para o TDD é o baby steps. Assim, se houver muitos testes que não passaram após implementação para passar em alguns, é bom revisar com a equipe para verificar se os passos não estão grandes de demais o que evita os tais raros casos de bug ou refatorações que quebram momentaneamente os muitos testes.

   II. F ( )

c) Antes de implementar um modulo de software em TDD, listamos uma lista de funcionalidades e/ou casos de teste. Começamos a implementação destas funcionalidades, e no meio da lista, percebemos que a implementação seria mais fácil e incremental se trocássemos a ordem das funcionalidades da lista. Mas o TDD não permite mudar a ordem preestabelecida a não ser que se renegociem as prioridades desses requisitos com o resto do grupo ou o chefe. (0.5)

I. V (x). Afinal, pode haver uma equipe para testes e outra para implementação e se a equipe de implementação não avisar a equipe de teste da mudança, a equipe de teste vai continuar fazendo os testes que testam a implementação de funcionalidade que não estão sendo implementadas enquanto que a implementação não está fazendo o código passar em nenhum dos testes. Havendo, portanto, uma quebra do ciclo do TDD.

II. F ( )

d) Assinale os itens que são verdadeiros: (0.5)

I. Testes funcionam como documentação, porque o teste é um exemplo de uso da classe.

Correto. Inclusive esse é um dos benefícios do TDD.

II. O TDD exige que a única documentação seja na forma de testes, desde o começo até o produto final. Não vale nenhum documento escrito ou on-line, nem nenhuma ferramenta de organização.

Falso. O TDD não possui nenhum conteúdo rígido assim sobre a documentação. Pode haver outros documentos que complementam os testes.

III. Um teste pode funcionar como documentação, mas para isso, precisa ser bem organizado. Separar testes correlatos em classes separadas, usar nomes descritivos para os métodos @Test, separar as fixtures em @Before ou @BeforeClass, etc, Essas práticas auxiliam os testes a serem legíveis e se comportarem como boas documentações.

Correto. Testes com nomes ruins e desorganizados são tão ruins quanto qualquer manual desorganizado e com sentenças ambíguas.

IV. Qualquer documentação pode incluir também exemplos de uso bem organizados. Uma vantagem de usar testes ao invés de documentação texto, é que se os testes são mantidos "green", se garante que os testes são atualizados quando o código muda, enquanto um exemplo em texto pode não funcionar mais.

Correto. A maioria do programador sabe disso por experiência quando vai procurar ajuda para resolver um bug pela internet. Há textos desatualizados na internet que não resolvem o bug ou geram outros bugs.

V. O TDD exige que todos o comportamento da classe, aos mínimos detalhes, incluindo exatamente como se comportar em todos os casos limite e possíveis exceções, estejam bem especificados antes de se começar a programar, mesmo que estejam escritos no papel de forma mais informal, como lista de testes.

Correto. Afinal de contas esses casos limites ou exceções devem ser testados.

VI. Durante a fase de refatoração, é obrigatório focar apenas na nova funcionalidade implementada durante as últimas fases RED-GREEN. Não podemos refatorar nada que não seja diretamente relacionado a nova funcionalidade.

Falso. Não nenhum problema de refatorar algum algoritmo anteriormente testado desde que ele continue passando nos testes após refatoração.

PARTE III - IMPLEMENTAÇÃO

QUESTÃO 3 - Joãozinho programa um Alarme!

Joãozinho precisa testar a classe Alarme, e para isso ele implementou a classe fake Sensor (que no sistema real será substituida pela classe sensor real, que realmente lê dados de um sensor real). Joãozinho reclama que só conseguiu implementar um teste com JUnit, e ele não sabe como testar.

a) Joãozinho tem dificuldade para testar porque o código não segue SOLID. Qual o principal princípio SOLID violado, e como a solução de Joãozinho dificulta o teste e a posterior integração da classe real no sistema? A solução pode melhorar em relação há vários princípios, mas ha um mais diretamente violado. (1.0)

O princípio de Single Responsability foi violado, pois a classe Alarme cria a classe Sensor, mas a classe Sensor já existe independente da existência da Classe Alarme, pois Sensor é um sistema real. Portanto, não é responsabilidade da classe Alarme criar a classe Sensor. Alarme deve receber o Sensor. Com a solução de Joãozinho um objeto Alarme contém um Sensor para vida toda, não podendo ser trocado, caso quebre ou caso haja uma atualização de melhoramento de componentes.

b) **[IMPLEMENTAÇÃO]** modifique o código seguindo SOLID para permitir testar com facilidade, e facilitar tambem a posterior integração da classe Sensor real. Implemente os testes (dica: use mockito). (3.0)

c) Realize e mostre (copie aqui pequenos trechos de código com versões “antes” e “depois”) **DUAS** (provavelmente pequenas) refatorações adicionais realizadas para melhorar a legibilidade do código e/ou seguir boas práticas de POO, explicando sucintamente a razão concreta da refatoração. (1.0)

Resposta:

Antes:

```
        private final double LowPressureThreshold = 17;
        private final double HighPressureThreshold = 21;
public Alarm(Sensor sensorNovo) {
        sensor = sensorNovo;//Agora temos uma agregação.
            }
```

Depois:

```
        private final double LowPressureThreshold;
        private final double HighPressureThreshold;
public Alarm(Sensor sensorNovo,double low, double high) {
        sensor = sensorNovo;//Agora temos uma agregação.
        LowPressureThreshold = low;
        HighPressureThreshold = high;
            }
```

Melhor porque agora podemos definir dinamicamente os limites inferior e superior, podendo, assim, criar diferentes alarmes para diferentes situações.

```
package Q3.TireMonitor;

import java.util.Random;

//The reading of the pressure value from the sensor is simulated in this
implementation.
//Because the focus of the exercise is on the other class.

public class Sensor {
 //public static final double OFFSET = 16;
       private static final double OFFSET = 16;
    private SampleGenerator _generator;

 public Sensor(SampleGenerator gen) {
        _generator = gen;
 }

 public double popNextPressurePsiValue() {
     double pressureTelemetryValue;
     pressureTelemetryValue = samplePressure( );

     return OFFSET + pressureTelemetryValue;
 }
public double getOffset() {
       return OFFSET;
}
 private double samplePressure() {
     // placeholder implementation that simulate a real sensor in a real tire
     //Random basicRandomNumbersGenerator = new Random();
     //double pressureTelemetryValue = 6 * basicRandomNumbersGenerator.nextDouble() *
basicRandomNumbersGenerator.nextDouble();
     double pressure = _generator.samplePressure();
        return pressure;
        }
```

Agora podemos gerar sensores com diferentes algoritmos de cálculo da Pressão sem necessitar alterar o código do sensor, e sem perder o algoritmo anterior. Podemos testar diferentes algoritmos de cálculo de pressão com a mesma classe Sensor. Isso é possível graças ao uso de conceitos como herança, polimorfismo e delegação que fizemos. Além disso, alteramos o acessibilidade da variável offset para private para não ocorrer alterações nessa variável.

d) A nova solução também melhorou em relação a outros princípios SOLID ou GRASP, mesmo que secundariamente? Explique em relação a um deles, além do principal citado em a). (1.0)

Sim, a solução retirou a responsabilidade de criar o sensor da Classe Alarme que não devia criar sensores o que condiz com o princípio de Single Responsability e High Coesion. Também

podemos citar o princípio de creator que delega corretamente a criação de objetos para as classes que mais fazem sentido fazê-las. Podemos dizer que o acoplamento diminuiu entre as duas classes, pois uma agregação é uma relação mais fraca do que a composição que existia anteriormente entre elas o que condiz com o Low Coupling do GRASP.

QUESTÃO 4 – TDD EM CÓDIGO LEGADO.

[baseado no livro: “Working Effectively with Legacy Code”]

Muitos de vocês estão ou já estiveram trabalhando com software legado e, na imensa maioria dos casos, a situação do código é de ruim para péssima. Outras vezes, o código é tão complexo, que é difícil (ou perigoso) mexer nele, pois pode encadear uma série de comportamentos não desejáveis e incontroláveis no mesmo. Muitas vezes em códigos legados prevalece a seguinte arquitetura “Big Ball of Mud”, ou, na melhor das hipóteses, uma tentativa de implementação de uma arquitetura em 3 camadas.

Por consequência desse descuido com a separação de responsabilidades do software, o código acaba sendo “macarrônico”, com métodos e classes gigantes, realizando todo o tipo de tarefa, desde validações da UI, passando pelas regras de negócio e chegando à persistência. O cenário acima é fruto, dentre vários fatores, de código projetado sem *testabilidade* em mente. Por isso tudo, retroalimentar a base de código legado com testes de unidade é o pior caso possível para a introdução de testes.

Um livro acerca do assunto foi construído por Michael Feathers, onde ele apresentou técnicas para realizar esta árdua tarefa. Uma das técnicas que ele apresentou foi batizada de *“Characterization Tests”,* que consiste em um teste que caracteriza o comportamento ATUAL de um pedaço de código, ou seja, **DEVEMOS TESTAR O QUE O CÓDIGO FAZ HOJE E NÃO O QUE GOSTARÍAMOS QUE ELE FIZESSE.** Sem isso em mente, tentaríamos, ao escrever o teste, corrigir algum bug ou fazer alguma alteração drástica no design, perdendo o foco do objetivo principal.

Um requisito essencial para trabalhar com códigos legados é saber gerenciar as dependências que seu código possui em relação a outro código (por ex., outra classe, em OO). Como dito anteriormente, código legado normalmente não é escrito pensando-se em testabilidade, isso significa que o código é altamente acoplado com dependências de baixo nível (classes que fazem I/O, classes que precisam de um contexto em *runtime* para executarem corretamente, etc.).

Dado os códigos abaixo, perceba que temos a classe RelatorioDespesas que tem por finalidade imprimir um relatório com todas as despesas existentes numa viagem.

Classe RelatórioDespesa

```
public class RelatorioDespesas {
	public void ImprimirRelatorio(Iterator<Despesa> despesas) {
		float totalDespesa = 0.0f;
		while (despesas.hasNext()) {
			Despesa desp = despesas.next();
			float despesa = desp.getDespesa();
			totalDespesa+= despesa;
		}

		Calculadora calculadora = new Calculadora ();
		calculadora.imprime(totalDespesa);
	}
}
```

Classe Despesa

```
public class Despesa {
	float total= 0.0f;

	public Despesa(float total) {
		this.total= total;
	}

	public float getDespesa() {
		return total;
	}

}
```

Classe Impressora

```
public class Impressora {
	public void Imprimir(String conteudo) {
		if (conteudo==null) {
			throw new IllegalArgumentException("conteudo nulo");
		}
		else
			System.out.println(conteudo);
	}
}
```

Classe Calculadora

```
public class Calculadora {
	public void imprime(float totalDespesa) {
		String conteudo = "Relatório de Despesas";
		Conteúdo+=("\n Total das despesas:" + totalDespesa);

		SistemaOperacional so = new SistemaOperacional();
		so.getDriverImpressao().Imprimir(conteudo);
	}
}
```

Classe SistemaOperacional

```
public class SistemaOperacional {
	public Impressora getDriverImpressao() {
		return new Impressora();
	}
}
```

a) **[IMPLEMENTAÇÃO]** Ao analisar as classes pode-se verificar que as classes possuem comportamentos bem estranhos, ou seja, dado o nome das classes elas tem responsabilidades ou em excesso ou ausentes, ou até mesmo em locais errados. Dessa forma, aplicando a lei de demeter e boas práticas de refatoração, remova os mau-cheiros existentes no código, de forma a ajustar as responsabilidades, bem como dependências existentes. (2.0) (Feito no eclipse)

b) **[IMPLEMENTAÇÃO]** Construa um teste unitário que permita que seja testada a classe RelatorioDespesas sem que as demais classes afetem ou possam influenciar nos testes (inserção de erros ou comportamentos / complexidade desnecessárias). (2.0)
(feito no eclipse)

c) **[IMPLEMENTAÇÃO]** Dado que impressoras matriciais não são mais presentes em nosso ambiente, apenas existindo impressoras laser ou jato de tinta, refatore o código de forma que seja possível usar qualquer tipo de impressora em RelatorioDespesas (ou seja para a classe RelatorioDespesas deve ser transparente qual o tipo físico de impressora a ser usada), onde o tipo correto é verificado através de um parâmetro do Sistema Operacional que deve reconhecer qual é a impressora correta. ATENTE, NÃO GERE DEPENDÊNCIAS DESNECESSÁRIAS. (1.0)

Resposta:

Consideramos que a questão não exige que RelatórioDespesas escolha uma das impressoras disponíveis para impressão, apenas quer que imprima o conteúdo não importando qual impressora faça isso. Portanto, eliminou-se a chama de getImpressora() da classe RelatórioDespesas, e criamos um método imprimir no SistemaOperacional. Assim, RelatórioDespesas apenas pede pro SistemaOperacional imprimir o conteúdo e o SistemaOperacional se vira para Imprimir. Além disso, escrevemos o construtor do SistemaOperacional para que ele possa receber todas as impressoras disponíveis.

a. Seus testes precisaram mudar? Caso sim, ajuste o código.

Resposta:

Dadas as modificações citadas acima, precisamos mudar o código o que foi feito e indica em comentário que foi feito uma refatoração.