CES-28 Prova 2 - 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 - Demeter não é só procurar por vários '.' na mesma linha.

[livro The Pragmatic Programmer cap 5]

```
void processTransaction(BankAccount acct, int value) {
        Person *who;
        Money amt;
        amt.setValue(value);
        acct.setBalance(amt);
        who = acct.getOwner();
        // saves log of all transactions
        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: A violação da Lei de Demeter ocorre na variável temporária who, na linha "*who = acct.getOwner();*". Isso é uma violação pois o método está conversando com um objeto estranho à ela, isto é, esse objeto não é um parâmetro, não é uma variável de instância (atributo do próprio objeto), não é criado localmente, não é o próprio objeto e também não é global. Assim, essa conversa com um objeto estranho quebra a Lei de Demeter, gerando uma dependência implícita no meio do código.

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: Não utilizar a variável "*Person *who;*" e a classe Account precisa de um método getOwnerName(), que retorna o que antes era necessário no *who->name()* .Assim o código fica:

```
void processTransaction(BankAccount acct, int value) {
        Money amt;
        amt.setValue(value);
        acct.setBalance(amt);
        logService(acct.getOwnerName(), SET_BALANCE);   // CORRIGIDO
}
```

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].
RESPOSTA: Foi eliminada a dependência entre a classe que contém o método processTransaction (a classe que está sendo programada, cujo nome não foi explicitado, suponha TransactionManager) e a classe Person do método name()
Reescrevendo: Foi eliminada a dependência name() que a classe TransactionManager tinha com a classe Person.
Isso melhorou por haver uma diminuição das dependências da classe TransactionManager, diminuindo o seu acoplamento. Isto é, caso a classe Person mude, isso não afetaria a classe TransactionManager.
Assim, se a classe Person mudar o seu método name(), isso não afetará mais o nosso código refatorado, já que essa responsabilidade foi delegada para BankAccount.

d) Explique a melhoria realizada acima em relação a pelo menos um dos princípios SOLID ou GRASP [0.5].
RESPOSTA: Com relação a um princípio GRASP, a melhoria foi realizada de forma a contemplar o princípio Low Coupling, isto é Baixo Acoplamento, uma vez que as dependências foram diminuídas. Isso também ajuda a aumentar a coesão do código, que é outro princípio GRASP de High Coesion, isto é, há uma forte relação do que o método faz com a sua responsabilidade.

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 ha 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.
II. **VERDADEIRO:** 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.
III. Necessariamente precisamos incluir esse caso nos testes já existentes, através de asserts extra.
IV. Implementamos esse novo caso na fase BLUE do TDD, ou seja, considerando-o como uma refatoração, pois não existe teste para ele.

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 (VERDADEIRO)
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 ( )
II. F (FALSO)

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

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

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.

III. **VERDADEIRO** 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.

IV. **VERDADEIRO** 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.

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.

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

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)
RESPOSTA: O pricípio SOLID violado foi a Dependency Inversion Principle (DIP), isto é, o código do alarme de Joãozinho (alto nível) depende de uma implementação (baixo nível) e não de uma abstração.
A solução de Joãozinho dificulta o teste pois fica mais difícil injetar a dependência (Sensor) dentro do Alarme, isto é, uma vez que o sensor já é criado dentro do Alarme, fica mais difícil criar mocks (fakes, stubs) de Sensor e passar para o Alarme, testando sua resposta para cada tipo de sensor diferente.
No caso de Joãozinho a situação ficou ainda mais difícil pois ele gera número aleatórios no sensor dele, não é um stub que já tem uma resposta enlatada, e assim ele não consegue prever o comportamento que o alarme deveria ter para poder testar corretamente.
Na posterior integração, Joãozinho terá que alterar o código do Alarme para que o mesmo utilize o sensor real no lugar do sensor falso, gerando mais trabalho para si.

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)
RESPOSTA: Fiz uma injeção de dependência, passando um Isensor (interface de sensor) como parâmetro para o construtor do Alarme.

OBS: O resultado final desse exercício está na package Q3.refatorado

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)

OBS: FORAM FEITAS NO CÓDIGO DO ECLIPSE TAMBÉM NA package Q3.refatorado

ANTES:

```
boolean alarmOn = false;
```

DEPOIS:

```
private boolean _alarmOn = false;

public void turnOffAlarm() {
     this._alarmOn = false;
}
```

Essa refatoração visa preservar os atributos internos da classe Alarme, impedindo que outros possam quebrar o seu encapsulamento. Nesse caso, colocou-se a variável de instância alarmOn como privada, porém ela apresenta um getter isAlarmOn() e um método para desligar o alarme turnOffAlarm(). Isso impede que alguém ligue o alarme externamente, isto é, projete o encapsulamento da classe Alarme.

ANTES:

```
if      (psiPressureValue      <      LowPressureThreshold      ||
HighPressureThreshold < psiPressureValue)
```

DEPOIS:

```
if (isPressureOffChart(psiPressureValue))
…
private boolean isPressureOffChart(double psiPressureValue) {
     return    psiPressureValue    <    LowPressureThreshold    ||
HighPressureThreshold < psiPressureValue;
}
```

Essa refatoração visa deixar o código mais legível, promovendo uma operação mais complicada a um novo método, cujo nome indica explicitamente o que está acontecendo internamente. A operação realizada era verificar se a pressão estava fora do intervalo solicitado (OffChart).

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)

RESPOSTA: Sim, além da inversão de dependência comentado anteriormente, as refatorações contribuíram para melhorar o entendimento do código e preservar o seu encapsulamento:

HIGH COESION e SINGLE RESPONSABILITY para a extração do método isPressureOffChart(), pois o método check() não precisa saber como faz o cálculo da pressão, essa responsabilidade é do isPressureOffChart(), e isso também contribui para aumentar a coesão do código, que trabalha junto com o mesmo objetivo do Alarme.

LOW COUPLING para o encapsulamento da variável alarmOn, provendo métodos getters e setters (parciais) para a mesma, uma vez que isso diminui a dependência de outras classes a essa variável, e caso a lógica interna do alarme mude, não serão necessárias outras mudanças para esse código.

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)

"DEVEMOS TESTAR O QUE O CÓDIGO FAZ HOJE E NÃO O QUE GOSTARÍAMOS QUE ELE FIZESSE"

RESPOSTA: **package Q4.refatorado1**

=> A Calculadora deve saber apenas como calcular o total, e não imprimir.

=> O imprimirRelatorio() só tem a responsabilidade de imprimir o relatório, não precisando calcular nada e nem saber como sistema operacional funciona. Apliquei o princípio de Inversão de Dependência, fazendo com que o imprimirRelatorio dependa apenas de uma abstração de impressora, e não de uma implementação.

=> O sistema operacional ficou isolado, a cargo do cliente que for escolher a impressora que lhe desejar, uma vez que não vejo porque o RelatorioDespesas tenha que se relacionar com o sistema operacional. O cliente que deve perguntar para o SO quais impressoras estão disponíveis, escolher, e falar para o RelatorioDespesas qual ele escolheu, algo que ocorre por exemplo quando se quer imprimir um documento na vida real: abre uma janelinha perguntando qual impressora devemos escolher, e escolhida a impressora, o programa se resolve, imprimindo o documento.

=> Criou-se uma interface de Imprime que é uma abstração das implementações de baixo nível.

Opcionalmente, de forma extra, criou-se um Overload do método imprimirRelatorio() caso a minha simplificação tenha sido considerada indesejada. Embora eu julgue a primeira implementação mais simples como a melhor.

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)
RESPOSTA: Teste realizado, graças à Injeção de Dependência.

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)

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

RESPOSTA: **package Q4.refatorado1**

Os meus testes não precisaram mudar, uma vez que eu já havia criado uma abstração para a impressora, bastou adicionar métodos do sistema operacional, para que houvesse uma lista de impressoras e não apenas uma impressora.
O sistema operacional é completamente desacoplado do RelatorioDespesas, não exigindo nenhuma mudança.