CES-28 Prova 3 - 2017

Sem consulta - individual - com computador - 3h

Gabriel Adriano de Melo

Obs.:

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, podem usar sites de ajuda online para procurar exemplos da sintaxe para os testes, e o próprio material da aula com pdfs, exemplos de codigo e labs, inclusive o seu código, mas sem usar código de outros alunos.
3. 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!
4. Só precisa implementar usando o Eclipse ou outro ambiente Java as questões ou itens indicados com o rótulo **[IMPLEMENTAÇÃO]**! Para as outras questões, você pode usar o Eclipse caso se sinta mais confortavel digitando os exemplos, mas não precisa de um código completo, executando. Basta incluir trechos de código no texto da resposta.
5. Submeter: a) Código completo e funcional da questão **[IMPLEMENTAÇÃO]; b)** arquivo PDF com respostas, código incluso no texto para as outras questões. Use os números das questões para identificá-las.
6. No caso de diagramas, vale usar qualquer editor de diagrama, e vale também desenhar no papel, tirar foto, e **incluir a foto no pdf dentro da resposta, não como anexo separado**. Atenção: use linhas grossas, garanta que a foto é legível!!!!

Joãozinho programa Interpolação **[IMPLEMENTAÇÃO]**

O *package* InterpV0 inclui uma aplicação de interpolação numérica. Há duas classes que implementam métodos de interpolação (não precisa lembrar os detalhes de CCI22, basta lembrar o conceito de interpolação). E há outra classe MyInterpolationApp que realiza todo o trabalho. A proposta principal desta questão é transformar o package de Joãozinho em 3 *packages* Model, View e Presenter que implementam o padrão arquitetural MVP.

Deve incluir uma view funcional, mas que imprime no console, e com métodos que simulam entrada do usuário humano. Por exemplo, se o usuário humano deveria digitar um inteiro, basta haver um método set(int value). Quando a main() chamar este método, simulamos entrada de usuário.

[IMPLEMENTAÇÃO] RODAR a Main.java no package default. Deve garantir que:

1. **[2 pt] O conceito de camadas seja seguido estritamente, e cada camada esteja em um package separado**.

[IMPLEMENTADO] Cada camada foi criada em uma package separada, sendo criadas as packages: model, presenter, view. O conceito de MVP em camadas foi seguido: apenas o Presenter conhece o model, o view conhece apenas o presenter.

[IMPLEMENTAÇÃO] RODAR a Main.java no package default.

2. **[2 pt]** Que seja possível adicionar outras implementações da camada View, com as mesmas responsabilidades, e usar várias instâncias de Views diferentes ao mesmo tempo com a mesma instância de Presenter e Model, **sem necessitar mudar o código de Presenter ou Model**.

[IMPLEMENTADO] Há acoplamento abstrato, as camadas de baixo (Presenter e Model) não dependem das camadas de cima (View). Model não sabe nada de View, Presenter tem apenas um acoplamento abstrato.
Para poder usar várias instâncias de Views diferentes ao mesmo tempo com a mesma instância de Presenter e Model, utilizou-se o D.P. Observer. Assim, o Presenter (Observable) notifica as views (Observers) quando é realizado o cálculo.

[IMPLEMENTAÇÃO] RODAR a Main.java no package default.

3. **[2 pt] SUBQUESTÃO [IMPLEMENTAÇÃO]:** (esta parte envolve um padrão de projeto além do MVP). Seja possível implementar e escolher outros algoritmos de interpolação, **sem precisar mudar nada no código além de uma chamada de método para registrar o novo algoritmo.** *As camadas superiores apenas precisam escolher uma String correspondendo ao nome do método de interpolação desejado.*

[IMPLEMENTADO] Utilizou-se o padrão criacional Abstract Factory, que cria os objetos por meio do parâmetro de String que ela recebe, na classe InterpolationFactory. Como ela tem um HashMap, basta fazer uma chamada ao seu método addInterpolationMethod para acrescentar um novo algoritmo.

[IMPLEMENTAÇÃO] RODAR a Main.java no package default.

[1 pt] Para cada uma das responsabilidades de MyInterpolationApp, indicadas com comentários no código e listadas abaixo, indique marcando uma colunas entre M, V ou P neste documento em qual camada deve ser incluída CADA responsabilidade. **DEVE CORRESPONDER AO SEU CÓDIGO**:

	M	V	P
1. RESPONSABILITY: DEFINIR PONTO DE INTERPOLACAO (LEITURA ENTRADA DE USUARIO HUMANO)		X	
2. RESPONSABILITY: DEFINIR QUAL EH O ARQUIVO COM DADOS DE PONTOS DA FUNCAO (LEITURA ENTRADA DE USUARIO HUMANO)		X	
3. RESPONSABILITY: ABRIR E LER ARQUIVO DE DADOS	X		
4. RESPONSABILITY: IMPRIMIR RESULTADOS		X	
5. RESPONSABILITY: DADO O VALOR DE X, EFETIVAMENTE LER O ARQUIVO			X
6. RESPONSABILITY: DADO O VALOR DE X, EFETIVAMENTE CHAMAR O CALCULO			X
7. RESPONSABILITY: CRIAR O OBJETO CORRESPONDENTE AO METODO DE INTERPOLACAO DESEJADO			X
8. RESPONSABILIDADE: EFETIVAMENTE IMPLEMENTAR UM METODO DE INTERPOLACAO	X		

GRASP x SOLID

[1pt : 0.5 por princípio] Para a solução do exercício da interpolação, explique como a solução final promove 2 princípios GRASP ou SOLID (não vale os princípios que apenas definem menor acoplamento e separação de responsabilidades, High Coesion, Low Coupling, Single Responsability).

- **Open Closed Principle: O código é fechado para modificações mas aberto para extensões, isto é, ao se acrescentar uma nova funcionalidade no código, digamos, uma nova view, herda-se (ou se implementa) os métodos necessários dessa nova view. Também se quiser acrescentar mais algoritmos, basta implementar a interface do InterpolationMethod e estender o comportamento do InterpolationFactory, acrescentando-se uma interpolação nele.**
- **Dependency Inversion Principle: As camadas dependem da injeção de dependência da outra camada abaixo. Assim, é necessário que se injete um presenter em uma view, para que a mesma possa funcionar. O Presenter não depende da implementação de View, apenas de sua abstração (no caso, foi utilizada a abstração de Observer).**

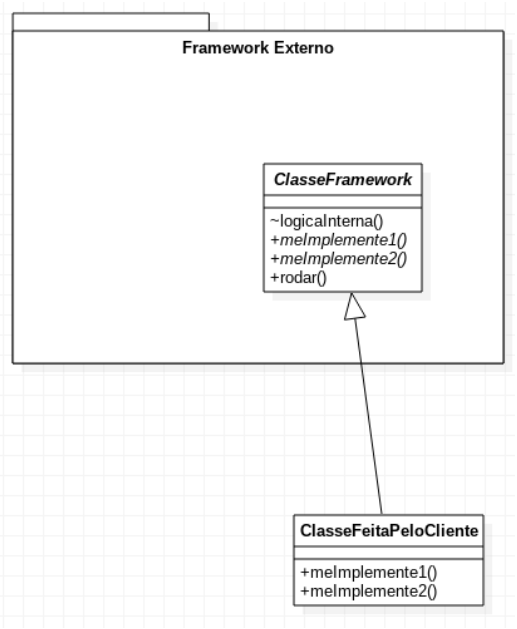
DPs são tijolos para construir Frameworks

[2 pt: 2 * { a) [0.5] b [0.5] }]
Escolha **2 (dois)** DPs que ao serem aplicados como parte do código de um Framework, promovam:

a) o **reuso de código**
b) a **separação de interesses** (separation of concerns), entre o código do framework e o código do programador-usuário do framework.

Explique conceitualmente como cada um 2 DPs promove os 2 conceitos a) e b). Vale usar diagramas UML na explicação, mas *deixe claro o que deve ser implementado pelo framework e o que deve ser implementado pelo programador-usuário do framework.*

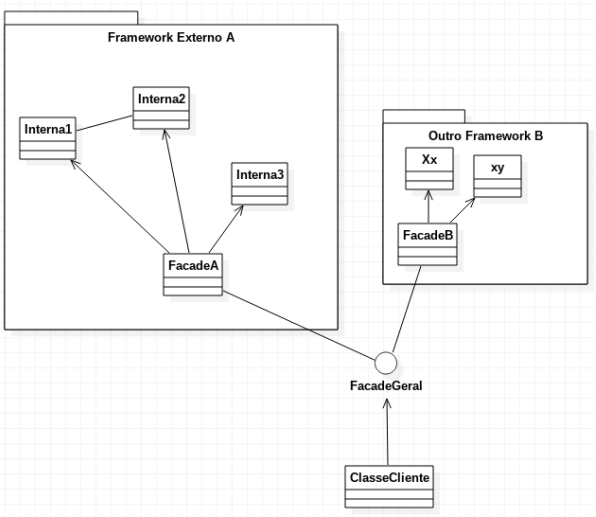
- **Hook Methods/Class:**
  - **Promove o reúso do código, o programador só precisa herdar a classe (ou implementar interface) e implementar os métodos necessários para fazer funcionar.**
  - **Também esconde os métodos internos da classe Pai (lógica interna de execução), promovendo a separação de interesses.**



No exemplo ao lado, o cliente deve estender a ClasseFramework e implementar os métodos meImplemente1() e meImplemente2(). Assim, quando a ClasseFeitaPeloCliente for executada (no método rodar), que chamará a lógica interna, que sabe, por exemplo, que o meImplemente1() deve ser executado antes do meImplemente2().

Assim, foi promovido o reúso de código (métodos herdados) e também a separação de interesses (o cliente não precisa saber nada da lógica interna)

- **Facade:**
  - **Promove as separação de interesses no pacote, diminuindo as dependências. Assim, o cliente só precisa utilizar e ter contato com apenas uma classe facade o pacote que ele for utilizar, diminuindo as suas dependências.**
  - **Se ele estiver utilizando vários pacotes cujas facades implementem uma mesma interface, ele pode se valer de polimorfismo para promover o reúso do seu próprio código.**



No exemplo ao lado, o cliente pode usar dois frameworks distintos (que tem a mesma facade), reaproveitando assim o seu código, utilizando-se de polimorfismo durante a execução, no qual ele poderia trocar de framework. Também há a separação de interesses, pois novamente o cliente não precisa saber das classes internas.

Abusus non tollit Usum

Conceito	Consequência do Abuso do conceito Marque o número apropriado conforme lista abaixo
Singleton DP	1 **(2)** 3
Dependency Injection	**(1)** 2 3
Getters and Setters	1 2 **(3)**

1. Excessiva quantidade de código e classes auxiliares para inicializar objetos:
2. Acoplamento excessivo e código difícil de entender devido à proliferação de Dependências e conflitos de nomes.
3. Confusão semântica dependendo da ordem de chamada de métodos, resultando em objetos com estado inválido.

a) **[0.5]** Associe cada conceito à consequência do seu abuso, marcando os números apropriados na a tabela acima, conforme a lista acima.
Associado!

b) **[1]** Escolha Singleton ou Dependency Injection e explique a causa da consequência, explicando o contexto do abuso do conceito.
Dependency Injection: Se eu tiver que injetar dependência em todo os objetos que eu crio vou ter classes auxiliares demasiadamente.

Isto é, se forem criadas injeções de dependências desnecessárias, a criação de novos objetos que requerem essas dependências fica cada vez mais trabalhoso, sendo necessário mais código e classes auxiliares para a construção desses objetos que requerem essas injeções.

Assim um abuso seria criar injeção de dependências para objetos de implementação única, e, em uma tentativa frustrada de criar um código genérico, o programador se veria forçado a ficar injetando dependências de um objeto que simplesmente poderia ser interno.

c) **[0.5]** Para o mesmo conceito escolhido em b), explique um contexto de uso apropriado, em que há razões claras para se utilizar o conceito sem incorrer nas consequências negativas.
Dependency Injection: Facilitar os testes, padrão de estratégia, mudar o comportamento reaproveitando código, desacoplamento de implementações.

Quando o objeto a ser injetado for algo que possa ser dinamicamente mudado na execução, isto é, quando existe mais de um objeto (de classes distintas) possíveis a serem injetados, o que mudaria o comportamento da classe que recebe a injeção.

Por exemplo, uma View do MVP que é injetada no Presenter, podendo assim mudar de uma interface de texto para uma interface gráfica apenas por meio da implementação da View, sem afetar o Presenter.