CES-28 Prova 1 - 2016

Sem consulta - individual - com computador - 3h

Obs.:

1. Qualquer dúvida de codificação Java só pode ser sanada com textos/sites oficiais da Oracle ou JUnit.
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 código 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 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 confortável 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 identifica-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 testa com JUnit! [2.0]
OO e reuso [1.5]
UML Família [2]
Mau cheiros simples [2]
Caixa eletrônico [6] [IMPLEMENTAÇÃO]

Joãozinho testa com JUnit! [2.0]

Considere os seguintes testes de Joãozinho como representativos de um conjunto grande de testes similares em um projeto maior. Suponha que todas as variáveis foram inicializadas coerentemente em um método @Before que não aparece aqui.

```
// esta classe: testes que devem passar, ou seja, ficarem verdes
public class GreenTests2 {
… // omitimos o @Before e outros @Tests similares
@Test
void TestaSistemaCadastroClientes() {

        // comeca sem clientes
```

```
        assertTrue("comeca sem clientes"
                        ,0,Cadastro.getNumberOfClients());

        //cadastro normal
        Cadastro.add(cli1);
        assertTrue("cadastro normal",Cadastro.hasCPF(cli1.getCpf()));
        assertTrue("cadastro normal",1,Cadastro.getNumberOfClients());

        // delecao normal
        Cadastro.del(cli1);
        assertFalse("delecao normal",Cadastro.hasCPF(cli1.getCpf()));
        assertTrue("delecao normal",0,Cadastro.getNumberOfClients());

        //cadastro de lista
        Cadastro.add(listWith10Clients);
        assertTrue("cadastro de lista",Cadastro.hasCPF(cli2.getCpf()));
        assertTrue("cadastro de lista",
                        11,Cadastro.getNumberOfClients());

        //delecao de lista
        Cadastro.add(sublistWith3Clients);
        assertFalse("delecao de lista",Cadastro.hasCPF(cli2.getCpf()));
        assertTrue("delecao de lista",
                        8,Cadastro.getNumberOfClients());


        …
        // inclui outros asserts e try-catches para, entre outros:
        // cadastro/busca/delecao estrangeiro (classe irma de cliente)
        // cadastro/busca/delecao de listas de estrangeiros
        // etc. varios outros asserts nao mostrados aqui.

}//end of method @Test TestaSistemaCadastroClientes()

}// end of GreenTests2 class
//************************************************

// esta classe: testes que devem dar erro, ou seja, ficarem vermelhos
public class RedTests2 {
… // omitimos o @Before e outros @Tests similares
```

```
@Test
Void FalhaSeTentaAdicionarClienteRepetido(){
        Cadastro.add(cli1.deepCopy());
}
@Test
Void FalhaSeTentaAdicionarClienteNull(){
        Cadastro.add(null);
}
@Test
Void FalhaSeTentaAdicionarClienteComNomeVazio(){
        Cadastro.add(clientWithEmptyName);
}
@Test
Void FalhaSeTentaAdicionarClienteComNomeInvalido(){
        Cadastro.add(clientWithInvalidName);
}

}// end of RedTests2 class
```

Depois de implementar inúmeros métodos @Test como esse Joãozinho então reclama que os testes com JUnit não ajudam na implementação do seu grande projeto:
Joãozinho: *"Nomeei os meus métodos @Test com nomes coerentes, o mesmo para todas as variáveis. Declarei e inicializei variáveis no método @Before para encurtar o código dos testes. Pensei em todos os casos inválidos e implementei testes e asserts() para testá-los, todos estão comentados e organizados. Escrevi os testes antes do código. Mesmo assim:*

1. *Erro em um teste Green me indica pelo nome do teste que há um problema, por exemplo, no cadastro de clientes, mas isso já sei pela exceção e saída do programa. Ok, pelo menos detecta o erro, mas não ajuda a identificar rapidamente o bug.*
2. *Isso quando não erro a mensagem no fail() ou assert(). É comum eu fazer copy-paste e esquecer de mudar a mensagem e/ou o comentário, que por sinal são repetidos muitas vezes, e então quando falha me confunde todo!!*
3. *Escrevi as classes de teste inteiras antes do código, conforme o TDD. Quando começo a implementar, é comum perceber que a API pode ser melhorada. Mas cada vez que refatoro a API, isso quebra muitos trechos de código espalhados no corpo dos métodos @Test!!!*
4. *Me disseram que estava escrevendo testes muito grandes e que devia separa-los em testes menores e organizados em classes de teste diferentes. Mas em primeiro lugar,*

os testes já estão organizados em verdes e vermelhos, cada grupo em sua classe de teste. Mesmo assim, preciso olhar todos os testes para garantir que todos os testes das classes Green passam e das classes vermelhas dão erro. As cores ajudam, mas ainda preciso olhar manualmente o resultado das classes GreenTestsX e RedTestsX. Você só viu o código da GreenTests2 e RedTests2!

5. *E, em segundo lugar, os testes vermelhos já são minúsculos, e mesmo assim, ainda não me permitem entender bem o que aconteceu, ou garantir que a exceção que eles soltam é realmente a exceção correta e desejada.*

A minha conclusão é que não adiantou nada usar o JUnit, só me deu trabalho extra para manter os testes e não me ajudou muito. Não podemos fazer TDD sem testes? "

Joãozinho está certo, ou ele poderia organizar os seus testes de forma que o seu processo de desenvolvimento seja mais efetivo (menos trabalho nos testes em si e mais ganho) considerando um processo TDD?
Convença Joãozinho considerando explicitamente pelo menos 4 dos pontos levantados por ele, e mostrando método(s) @Test completos como exemplo para convencer o Joaozinho (**não** precisa testar ou repetir todos os itens testados acima, apenas mostre método(s) @Test coerente(s) como exemplo para o Joãozinho, em número apenas suficiente para exemplificar o que desejar demonstrar). Vale acessar manuais on-line do JUnit. Não vale xingar o Joaozinho!

OO e reuso [1.5]

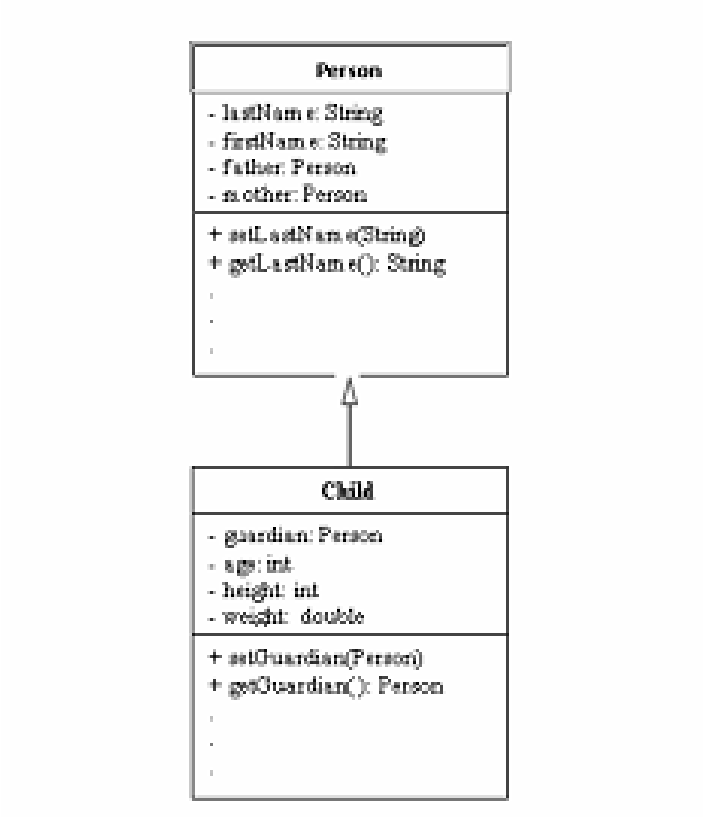
Herança, polimorfismo, overriding methods (substituição e refinamento), method overloading, encapsulamento (ou modos de acesso) são conceitos de OO. Escolha 3 deles e explique com trechos de código comentados como cada um dos 3 conceitos pode facilitar o reuso de código (um exemplo basta).

Não precisa executar código, ou mesmo escrever código completo. Mas precisa de trechos de código que demonstrem a resposta (pode usar "…" para indicar partes não relevantes e omitidas do código). Comentários não precisam ser / comentários java */, podem ser textos livres, indicar trechos com setinhas, etc.*

UML Família [2]

O desenho parcial de um aplicativo Java para um centro de cuidados infantis é dado no seguinte diagrama UML. Note-se que o diagrama não é completo. Como você representaria os seguintes relacionamentos no projeto: pai, mãe e guardião?

Usando como base o diagrama abaixo apresentado e de posse de conceitos de orientação a objeto, represente os conceitos pai, mãe e guardião em UML. Para isso reveja o diagrama, inclua as relações necessárias (associações, agregação, composição e herança), bem como as multiplicidades necessárias e navegabilidade.



Mau cheiros simples [2]

Utilizando as técnicas de refatoração, reescreva o código abaixo de modo a eliminar os mau-cheiros encontrados. Mostre passo a passo, indicando em amarelo as mudanças, e

explicando o sentido cada refatoração (é suficiente dar a cada refatoração0 um nome apropriado).

```
public class MediaDeVendas_Questao {
        public int totalDeVendas;
        public double valorTotalVendido;

        public void calcularMediaDeVendas() {
        System.out.println("Vendas de Agosto");
        this.totalDeVendas = 372;
        this.valorTotalVendido = 1726493.00;
        double valorMedioPorVenda = this.valorTotalVendido / this.totalDeVendas;
        System.out.println("Valor Médio por Venda em Agosto: " + valorMedioPorVenda);
        System.out.println("Vendas de Setembro");
        this.totalDeVendas = 472;
        this.valorTotalVendido = 1836917.15;
        valorMedioPorVenda = this.valorTotalVendido / this.totalDeVendas;
        System.out.println("Valor Médio por Venda em Setembro: " + valorMedioPorVenda);
        }
}
```

Caixa eletronico [6] [IMPLEMENTAÇÃO]

Escreva um diagrama UML com as classes definidas pelos requisitos abaixo, especificando multiplicidades, e especialmente os tipos de associação, agregação e composição. O código arquitetural deve corresponder ao diagrama UML!
Criar, utilizando TDD, uma classe chamada CaixaEletronico, juntamente com a classe ContaCorrente, que possuem os requisitos abaixo:

- A classe CaixaEletronico possui os métodos logar(), sacar(), depositar() e saldo() e todas retornam uma String com a mensagem que será exibida na tela do caixa eletrônico.
- A classe CaixaEletronico possui, e não pode existir sem que possua, instâncias das interfaces Hardware e ServicoRemoto.

- Existe uma classe chamada ContaCorrente que possui as informações da conta necessárias para executar as funcionalidades do CaixaEletronico. Essa classe faz parte da implementação e deve ser definida durante a sessão de TDD.
- As informações da classe ContaCorrente podem ser obtidas utilizando os métodos de uma interface chamada ServicoRemoto. Essa interface possui o método recuperarConta() que recupera uma conta baseada no seu número e o método persistirConta(Conta c) que grava alterações, como uma mudança no saldo devido a um saque, ou um depósito. Não há nenhuma implementação disponível da interface ServicoRemoto e deve ser utilizado um Mock Object para ela durante os testes.
- A Conta pode ter depósitos pendentes associados a ela. Cada deposito possui um valor e outras informações sobre onde e quando foi feito. Como depósitos são feitos através de envelopes, na verdade um deposito não aumenta o saldo, apenas fica registrado na ContaCorrente como pendente. O banco, até o dia seguinte, depois de verificar o envelope, ira remover a marcação de depósito e aumentar o saldo, mas isso não é feito no nosso sistema.
- O método persistirConta() da interface ServicoRemoto deve ser chamado apenas no caso de ser feito algum saque ou depósito com sucesso. Sucesso inclui sucesso na utilização do hardware, por exemplo, se o hardware não conseguir entregar o dinheiro, o saque não deve ser tornado persistente no servidor.
- O método persistirConta() deve ser chamado uma vez a cada operação de deposito ou saque. Nao se deve realizar várias operações locais e se tentar persisti-las de uma vez so.
- Ao executar o método saldo(), a mensagem retornada deve ser "O saldo disponível é R$xx,xx. \n " com o valor do saldo. Se houverem depósitos pendentes ainda deve ser impresso uma segunda linha “Total de depósitos a conferir: R$xx,xx.“
- Ao executar o método sacar(), e a execução for com sucesso, deve retornar a mensagem "Retire seu dinheiro". Se o valor sacado for maior que o saldo da conta (sem contar os depósitos pendentes), a classe CaixaEletronico deve retornar uma String dizendo "Saldo insuficiente".
- Ao executar o método depositar(), e a execução for com sucesso, deve retornar a mensagem "Depósito recebido com sucesso".
- Ao executar o método login(), e a execução for com sucesso, deve retornar a mensagem "Usuário Autenticado". Caso falhe, deve retornar "Não foi possível autenticar o usuário"
- Existe uma interface chamada Hardware que possui os métodos pegarNumeroDaContaCartao() para ler o número da conta do cartão para o login (retorna uma String com o número da conta), entregarDinheiro() que entrega o dinheiro

no caso do saque (retorna void) e lerEnvelope() que recebe o envelope com dinheiro na operação de depósito (retorna void). Não tem nenhuma implementação disponível da interface Hardware e deve ser utilizado um Mock Object para ela durante os testes.

- Todos os métodos da interface Hardware podem lançar uma exceção dizendo que houve uma falha de funcionamento do hardware.

1. [1] Crie testes para deposito, saque, e saldo.
2. [2] Crie testes que verifiquem, para uma sequência de operações de deposito e saque, que (a) o sistema remoto foi chamado o número correto de vezes; (b) a conta possui o numero correto de depósitos pendentes
3. [2] Crie testes para tentativas de saque sem fundo. O saque só pode ser feito sobre o saldo, sem contar o total de depósitos pendentes. Esse caso também deve ser testado, e deve ser verificado que o hardware foi chamado apenas para os saques validos e não para os inválidos (pelo menos contar o número de vezes).
4. [1] Criar testes também para os casos de falha, principalmente na classe Hardware que pode falhar a qualquer momento devido a um mau funcionamento.

Lembre-se de usar o TDD e ir incrementando as funcionalidades aos poucos. Deve entregar os testes incrementais que demonstrem o TDD.
Você deve entregar o código final, incluindo os testes e os mock objects criados. Organize em packages o código e os testes.

Não precisa detalhar campos das classes, nem no papel, nem na UML, nem no código. Os campos existem para explicar a necessidade das classes para o aluno e ajudar a dar contexto. Pode implementar uma classe vazia ou apenas um campo simplificado representando os diversos dados da entidade quando escrever os outros campos seria apenas adicionar algumas atribuições. Por exemplo, uma Cliente poderia ter apenas um inteiro CPF, e omitir nome e endereço, ou um formato complexo para o CPF por brevidade.
