GABARITO CES-28 Prova 1 - 2016

Joãozinho testa com JUnit! [2.0]

1) Se cada funcionalidade e cada caso fosse separado em um método @Test diferente, ao falhar um teste, seria fácil notar qual é o bug no programa, ainda mais se o nome colocado no @Test for da forma When(...)Then(...). Se Joãozinho criasse um método @Test para cada funcionalidade, o próprio JUnit mostrará uma lista de testes que falharam, com nomes mais específicos e padronizados e será possível ver cada um que falhou ou passou e rapidamente identificar o problema talvez olhando só para o nome dos testes, e tudo de uma vez só, sem a necessidade de ficar debugando o código. [0,3]

2) Um teste simples, pequeno, com nome padronizado não precisa da mensagem no fail ou assert. Nao tem o que errar em um *copy-paste*. [0,3]

3) Não devem ser escritos todos os testes antes do código de produção. Os testes devem ser feitos antes da adição de uma nova funcionalidade no código de produção. O TDD impõe que os testes sejam feitos durante a implementação do código e não antes, conforme ele fez. O certo seria ir produzindo testes que falhassem à medida que o código fosse sendo criado e fazer apenas a refatoração necessária para o novo teste passar. [0,3]

4 e 5). Olhe os exemplos, embora o JUnit admita outras possíveis sintaxes para tratar exceção, não precisa de *try-catch*, fica tudo curtinho, direto. Mesmo que faça *copy-paste*, não tem erro, fica tudo verde, e não precisa olhar cada um, basta checar se há 0 (zero) erros e 0 (zero) *failures* na soma final. [0,6]

[0.5]

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
void WhenCadastroNormalThenHasCPF() {
        Cadastro.add(cli1);
        assertTrue(Cadastro.hasCPF(cli1.getCpf()));
}

@Test
void WhenCadastraClienteJaCadastradoThenThrowException() {
        Cadastro.add(cli1);
        thrown.expect(ClientAlreadyExistsException.class);
        Cadastro.add(cli1.deepCopy());
}

@Test
void WhenCadastroNullThenThrowException() {
        thrown.expect(NullClientException.class);
        Cadastro.add(null);
}
```

OO e reuso [1.5]

a) **Herança:** [0,3]
   - *Dalmatian extends Dog*
   - Simples, atributos e métodos de *Dog* (dependendo do modo de acesso) são diretamente reusados no código de *Dalmatian*.

b) **Polimorfismo:** [0,3]
- *O* exemplo do acoplamento estático envolve herança e polimorfismo.
- *O* código que chama *shape.setColor()* é reusado para Quadrado, Circulo, *ShapeTeste*, e qualquer outro *Shape* que venha a ser implementado como filho de *AbstractShape*.

c) **Overriding Methods:** [0,3]
- É mais explícito com refinamento. Todo o código do método da classe pai é executado durante a execução do método da classe filho. Mas mesmo com substituição, o código que chama *dog.bark()* é o mesmo código, mesmo que *dog* na verdade seja um *dalmatian*.

d) **Overloading:** [0,3]
- Exemplo método *play()*, com versões *play(Object)* e *play(Midi)*.
- Como Java utiliza o tipo estático do parâmetro para decidir qual método chamar, o código que chama o método *play()* deve ser modificado se desejarmos mudar o método a ser chamado, ou seja, *play(meumidi)* é diferente de *play(outroobjeto)*, então não há reuso tão explicito e direto.
- Mas, se uma das versões *overloaded*, é apenas um *Adapter* que chama outras versões, temos reuso simples e direto via *overloading* como no exemplo acima.

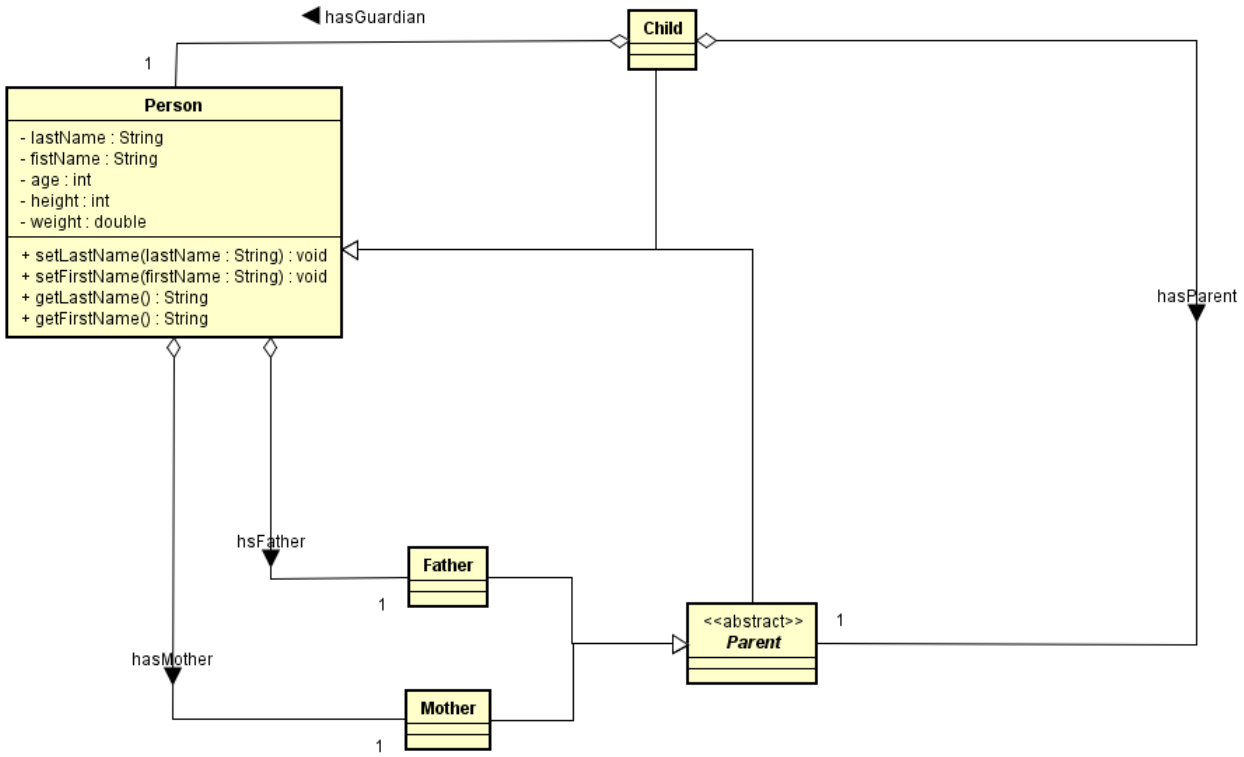
```
BigBlock(Thing myThing) {
        this(myThing, 0, 0, 0); // Assuming 0 is the default value for x, y and z
}
```

   - Este construtor mais simples reusa um construtor mais complexo, especificando um caso especial do caso geral.

e) Encapsulamento: [0,3]

- *O protected* permite que os filhos acessem e reusem atributos e métodos sem expor os métodos publicamente. Assim, *protected* facilita o reuso do código da classe pai pelos filhos.
- *Os private e protected* facilitam o reuso do próprio objeto pelos clientes, já que podemos garantir determinados comportamentos da classe.

UML Familia [2]



Mau cheiros simples [2.0]

```
public class MediaDeVenda_Resposta {

	private int totalDeVendas; //Encapsulate Field
	private double valorTotalVendido; //Encapsulate Field

	public void calcularMediaDeVendas() {
		setTotalDeVendas(372);
		setValorTotalVendido(1726493.00);
		calcularMediaPorVenda("Agosto");
		setTotalDeVendas(472);
		setValorTotalVendido(1836917.15);
		calcularMediaPorVenda("Setembro");
	}

	private void calcularMediaPorVenda(String mes) {////use extract
method
		System.out.println("Vendas de " + mes);
		double valorMedioPorVenda = this.valorTotalVendido /
this.totalDeVendas;
		System.out.println("Valor Médio por Venda em " + mes + ": "
+ valorMedioPorVenda);
	}

	public int getTotalDeVendas() {////use extract method
		return totalDeVendas;
	}

	public void setTotalDeVendas(int totalDeVendas) {//use extract
method
		this.totalDeVendas = totalDeVendas;
	}

	public double getValorTotalVendido() {//use extract method
		return valorTotalVendido;
	}

	public void setValorTotalVendido(double valorTotalVendido) {//use
extract method
		this.valorTotalVendido = valorTotalVendido;
	}

}
```

FeedBack P1 Implementacao

Exceções

Se esta especificado que hardware deve soltar excecoes, isso deve acontecer. Comentei na primeira aula que quem nao sabe lidar com excecoes devia aprender.

Interfaces

Alguns ainda estao confusos com o conceito de interfaces e/ou classes abstratas. É parecido, mas interfaces em java nao podem definir corpos para os metodos, nao tem “{ }”, só “;” . (O Java 8 relaxou um pouco a diferenca, mas ainda eh diferente)
De novo, quem esta confuso deve ir atras agora, vamos usar muito ambas as coisas.

Testes, nomes, mokito

Um bom exemplo de como o codigo pode ficar legivel. Primeiro, note que o nome do teste define especificamente o que queremos testar. Depois, a linha do verify ficou bem interessante!

```
public void QuandoSacaSemFundoEntaoNaoEntregaDinheiro(){
       ....
       verify(_hardware, never()).entregarDinheiro();
       //  ^-- gostei dessa linha!
```

Cuidado com o obvio

Quando se faz um saque, havendo fundos, o saldo muda depois do saque, certo?

Teste o SUT, nao teste o mock

SUT = System Under Test
Nao adianta verificar se o hardware funciona.
Definir no mokito que hardware.entregarDinheiro() retorna true, e depois fazer assertTrue(hardware.entregarDinheiro())
Isso nao eh um teste, esta testando o mock, e nao o SUT
O teste seria chamar o ATM, que usa o hardware, e precisa que o hardware retorne true para funcionar, e verificar que o ATM funciona.

Da mesma forma, nao adianta setar o mock pra lancar uma execao em entregarDinheiro, e depois testar se o mock joga essa excecao. Precisa chamar o ATM, em uma situacao onde a excecao seja jogada pelo hardware dentro do ATM, e verificar se o ATM se comporta da forma esperada.

Compare strings com equals(), nao startsWith()

```
assertTrue(mystring.startsWith(“resultado desejado”));
```

Cuidado com comparar strings com startsWith. Em muitos casos, pode ser que a string tenha outras coisas no final e a comparacao ficaria errada. Se existe string.StartsWith(), tambem existe String.equals(). A menos que realmente se deseje comparar so o comeco da string.

Se estiver preocupado com possiveis espacos no final, use String.trim()
```
assertTrue(mystring.trim().equals(“resultado desejado”));
```
Ou, melhor ainda, porque eh mais facil de ler:
```
assertEquals(“resultado desejado”, mystring.trim());
// aceita mystring mesmo que tiver espacos extra
```

Isso usa o fato que assertEquals chama o metodo .equals() dos seus parametros.
http://stackoverflow.com/questions/1201927/java-is-assertequalsstring-string-reliable

Nao polua o codigo. Pra isso existem os mocks, spy, etc.

Nao vale a pena criar um metodo do hardare, do ATM, ou uma flag para contar chamadas. Alguns, antes de cada chamada de qualquer metodo no hardware, adicionavam uma linha para cuidar de contar as chamadas.
Isso polui o codigo, e ainda por cima pode causar bugs - o erro esta no codigo da funcionalidade, ou no codigo que conta as chamadas?
O mockito permite contar diretamente quantas vezes um metodo do mock foi chamado, e eh isso que interessa! Sem necessitar de trabalho extra para definir flags e contadores manualmente!

Nao polua o codigo com variaveis temporarias, dificulta a leitura.

O que eh mais facil de ler:

```
String resultado = ATM.saque(10);
assertEquals(“Retire seu dinheiro”, resultado);
```

Ou

```
assertEquals(“Retire seu dinheiro”, ATM.saque(10));
```

Lembrem-se, isso eh mais importante quando ha centenas de testes!

Evitem acentos em codigo

Meu eclipse em linux se perdeu completamente com packages chamados “código”. Nem compila, nem conseguia refatorar automaticamente o nome.
Quando ha erros em mensagens de printouts, pelo menos o codigo compila.
Ok, sei que da pra configurar isso, mas se o seu codigo sera compartilhado com outras pessoas, alguem estara usando outro SO, ou usando um SO configurado para ingles ou outra lingua qualquer. Eu sempre instalo o meu linux em ingles justamente pra evitar problemas de locale em todas as coisas que desenvolvo, evita muita dor de cabeça.
Evita retrabalho se nao ha codigo com acentos!