Dicksiano Carvalho Melo

P2 CES28 - COMP19

QUESTÃO 1

a)

A LoD afirma que uma classe deve acessar apenas:

1) Objetos passados como parâmetro
2) Objetos instanciados pela própria classe
3) Objetos da mesma classe
4) Objetos que sejam atributos dessa classe

```
void processTransaction(BankAccount acct, int value) {
	Person *who;
	Money amt;
	amt.setValue(value); // amt é instanciado por processTransaction e value é passado como parâmetro
	acct.setBalance(amt); // acct é passado como parâmetro e  amt é instanciado por processTransaction
	who = acct.getOwner(); // Violação! Está acessando um objeto da classe Person encapsulado dentro de BankAccount
	// saves log of all transactions
	logService(who->name(), SET_BALANCE); // Violação! Está acessando um objeto (provável que name() seja string que é nativa de Java, aí não teria problema) encapsulado dentro da classe Person
}
```

b)

```
void processTransaction(BankAccount acct, int value, string name) {
	Money amt;
	amt.setValue(value);
	acct.setBalance(amt);
	// saves log of all transactions
	logService(name, SET_BALANCE); // o nome já é passado como parâmetro
	                               // Não precisa saber da existência da
	                               // classe Person
}
```

c)
Nesse caso, não há mais dependência com a classe Person. Imagine que BankAccount seja modificada e não possua mais uma entidade Person e sim uma entidade IPerson (que pode ser implementada por uma pessoa física ou uma pessoa jurídica, por exemplo), não haveria problema, pois não mais acoplamento com essa classe, apenas com o nome desse objeto que faz-se sempre necessário, uma vez que uma BankAccount deve ser relacionada com algo (uma empresa, por exemplo) ou com alguém (pessoa física), mas em ambos os casos (algo ou alguém) essa entidade precisa de um nome para ser identificado.

Outra modificação seria alterar a classe BankAccount para que o método setBalance() pudesse receber ou a classe Money ou um int. Assim, não seria necessário criar a variável “amt” e, dessa forma, o método processTransaction não iria mais depender da classe Money, diminuindo ainda mais o acoplamento.

d)
Claramemente houve diminuição do acoplamento (**Low Coupling**) , pois não há mais a necessidade de conhecimento da classe Person para que o método funcione. Além disso, processTransaction() não tem mais responsabilidade de criar a instância “who”, diminuindo sua responsabilidade e tornando-a mais coesa (**High Cohesion**).

QUESTÃO 2

a) **Item II**

Deve-se criar um novo teste que não irá passar no estado atual do software (RED!). Depois disso, segue o TDD: fazer o código cumprir essa funcionalidade e, desse modo , o teste passa (GREEN!) e refatora se possível/necessário.

b) **Verdadeiro!**

Espera-se que o software continue funcionando para os testes mais antigos. Caso contrário, há quebra no modelo unidirecional do TDD

c) **Verdadeiro, com ressalva**

Depende, pode ser que as funcionalidades já implementadas não precisem ser mudadas, apenas funcionalidades futuras. Por exemplo, imagine 5 funcionalidades: 1, 2, 3 ,4 e 5 (nesta ordem!). Já implementamos 1 e 2. Percebe-se que seria melhor fazer 1, 2, 5, 4 e 3. Como ainda não houve inicio da implementação de 3, 4 ou 5, ocorreira apenas um reprojeto, mas isso não quebraria a lógica do TDD (Red → Green → Refactor). Porém, digamos que 3 e 4 já estam implementados e a mudança no projeto cause que os testes de 3 parem de funcionar. Nesse caso, testes que passavam pararam de funcionar e houve quebra na sequência do TDD.

O que não ficou claro para mim no enunciado foi se ocorreria quebra no TDD como no segundo exemplo ou não.

d)
I – Sim, é válido utilizar os testes para entender o funcionamento do código.
III – Sim, não faz sentido tentar utilizar um teste como documentação se ele for pouco legível.
IV – Correto. Pode acontecer do programador atualizar um software e não atualizar a documentação e isso poderia causar diversos transtornos ao usuário. Utilizando testes que passam como documentação, o usuário poderia notar que houveram mudanças no software.

Itens falsos:
II – Falso, isso não é uma exigência
V – Falso, espera-se que os testes iniciem pelos casos mais simples. A medida que o software evolui, surgirão casos mais complexos. Não é uma exigência que seja conhecido a priori “ todos o comportamento da classe, aos mínimos detalhes, incluindo exatamente como se comportar em todos os casos limite e possíveis exceções”. Até mesmo porque é frequente que novos casos testes especiais tenham sido esquecidos antes de iniciar as implementações e só sejam lembrados após o início das implementações. Seria excelente que tudo estivesse no papel antes de iniciar as implementações, mas na prática isso é pouco viável.
VI – O ideal seria refatorar apenas as novas funcionalidades mais recentes. Porém, se um “mau-cheiro” for detectados apenas em iterações avançadas do TDD deve-se refatorá-lo. Na prática, é pouco viável querer restrintgir refatorações apenas das últimas funcionalidades implementadas.

QUESTÃO 3 - Joãozinho programa um Alarme!

a)

O princípio ferido é o **Dependency Inversion Principle**. O problema é que Alarme (abstração) depende de Sensor (detalhe). Isso é um ruim, pois Joaozinho vai ter que primeiramente testar a saída da classe Sensor (esforço desnecessário!) e quando for integrar o sistema real, terá que modificar a classe Alarme. O ideal é que Alarme dependesse de uma Interface, pois seria mais fácil de testar mockando a Interface e utilizando respostas enlatadas (Stub) e, quando fosse implementar o sistema real, bastaria que o sensor real implementasse a interface utilizada. Desse modo, não haveria nenhuma alteração na classe Alarme quando houvesse integração com o sistema real.

b) Implementação! Há comentários no código!

c)

```
	private final double _LowPressureThreshold;
	private final double _HighPressureThreshold;
	private _sensor;
	private _alarmOn;

	public Alarm(ISensor sensor, double LowPressureThreshold, double
HighPressureThreshold) {
		this._sensor = sensor;
		this._alarmOn = false;
		this._LowPressureThreshold = LowPressureThreshold;
		this._HighPressureThreshold = HighPressureThreshol;

	}

	// Construtor default
	public Alarm(ISensor sensor) {
		Alarm(sensor, 17, 21); // Chama o construtor com parametros default
	}
```

As pequenas modificações foram:
1) Criar um novo construtor que permita que os limites sejam passados na inicialização do objeto. Dessa forma, a classe se torna mais geral, pois com os limites fixos (17 e 21) essa classe só poderia ser utilizada para o mesmo sistema. Com a modificação, a classe Alarm pode ser aplicada em vários outros sistemas com limites diferentes. Caso o usuário não passe os valores no construtor, então são utilizados os valores default (17 e 21). Isso utiliza o princípio **Open/Closed Principle**, pois torna a classe aberta para extensões.

2) Colocar o “_” em todas os atributos privados. Isso facilita a leitura do código.

d)

Sim, melhorou em relação ao **Open/Closed Principle**, pois agora a classe Alarm pode ser estendida para ser utilizada com qualquer outro tipo de sensor que implemente a interface ISensor. Mas ela continua fechada para modificações, uma vez que seu comportamento interno permanece inalterado. Além disso, percebe-se que no código de Joaozinho, Alarm era responsável por intanciar um objeto da classe Sensor, ou seja, tinha a responsabilidade de um Creator. Agora, ele recebe esse sensor como parâmetro e, desse modo, ocorreu melhora em relação ao **Single Responsibility Principle**.

QUESTÃO 4 – TDD EM CÓDIGO LEGADO

Implementação! Há comentários no código!

a) Alteradas as responsabilidades:

Calculadora → apenas calcula o valor total das despesas
Sistema Operacional → apenas escolhe impressora
Impressora → apenas imprime uma mensagem se essa não for vázia
Despesa → classe utilizada para guardar informações sobre uma dada despesa
RelatorioDespesas → apenas recebe um conjunto de despesas e utiliza as classes anteriores como colaboradoras

b) Fiz o unit test utilizando respostas enlatadas para as classes que circundam RelatorioDepesas. No final, eu testo se a mensagem que foi sem impressa na impressosora é a mesma que eu esperava.

c) Fiz com que SistemaOperacional retornasse não um tipo de impressora, mas uma interface. Essa interface deve possuir apenas um método imprimir(). Assim, a classe RelatorioDespesas não enxerga essa alteração. Meu teste não mudou, pois como eu utilizo respostas enlatadas, eu escolho qual impressora é retornada pelo sistema operacional.