VICTOR HUGO FERNANDES BREDER
CES-28 PROVA 2 – 2017
PROF. TEN CEL BARRETO

QUESTÃO 1

a) A violação da Lei de Demeter consiste no fato que a função apresentada está atuando diretamente nos dados pertencentes a objetos passados, acessando-os e modificando-os diretamente. Para que boas práticas de orientação a objeto sejam seguidas, qualquer objeto deve, sempre que possível, gerenciar os próprios dados e ser sinalizados, através de seus métodos, sobre como alterar seus dados internos.

b) Em vez de passar “int” como valor monetário, deve-se passar um objeto da classe Money, já instanciado e com o valor correto de dinheiro. Dessa maneira, informações como a moeda utilizada podem ser passadas juntas à quantidade de dinheiro, bem como possibilita futura extensão.

Além disso a função inteira deve passar a ser um método de BankAccount, já que esta fundamentalmente acessa e modifica dados de uma conta bancária. Torná-la método desta classe reduz o acesso apenas aos dados necessários e também a deixa localizada com seu caso de uso de forma coesa.

Também é possível que um método de logging seja introduzido na classe Person para que o próprio objeto dessa classe seja responsável por acessar seu nome e realizar o logging de sua atividade.

```
class BankAccount {
      [...]
      public void processTransaction(Money amount) {
            this.setBalance(amount);
            getOwner().log(SET_BALANCE);
      }
      [...]
}
```

c) Eliminou-se dependência entre BankAccount e as implementações de Money e Person. Isto é, qualquer objeto passado à função, bastando que siga a interface de Money, irá se comportar da maneira correta. Além disso, basta que Person possua um método log que recebe a ação que deve ser registrada.

Mudança no formato do log, como acrescentar outras informações do dono da conta, não implicarão mais alterações na função processTransaction. Além disso, acrescentar uma moeda associada ao valor monetário, também não implicará mudança na função apresentada e a lógica para lidar com isto estará incluída exclusivamente em setBalance.

d) Com a solução apresentada, prioriza-se a comunicação com objetos através de suas interfaces, deixando de acessar a atuar diretamente sobre implementações e dados pertencentes a objetos. Além disso, a função tem seu acesso limitado às informações que necessita para realizar sua finalidade. Por fim, uma modelagem encapsulada de dinheiro é utilizada em preferência ao tipo inteiro.

QUESTÃO 2

a) II. Verdadeiro. O teste que cobre este caso especial deve ser introduzido na fase RED, de maneira que o teste criado falhe e demonstre a necessidade desta correção na base de código. Em seguida, na fase GREEN, deve-se modificar a base de código de maneira a cobrir este caso especial. Quando o teste escrito na fase anterior passar, haverá certeza de que todas as partes afetadas pelo caso especial foram cobertas. Em seguida, na fase BLUE, “maus cheiros” no código acrescentado podem ser corrigidos através de refatoração, mantendo todos os testes passando.

b) I. Verdadeiro. O princípio de TDD é manter a base de código funcionando em cada passo do desenvolvimento para assim, ser possível apresentar periodicamente código útil e funcional e não apenas esperar que tudo funcione no final. Além disso, esta metodologia reduz o custo de implementação de novas funcionalidades e de refatoração do código existente, já que os testes garantem que funcionalidades antigas continuam funcionando.

c) II. Falso. TDD permite esta flexibilidade e favorece implementação incremental.

d)
I. Verdadeiro.
II. Falso. TDD não aplica limitações a documentação, é apenas uma metodologia de desenvolvimento de software.
III. Falso. Testes correlatos devem ser mantidos próximos e na mesma classe. O restante da afirmativa está correto.
IV. Verdadeiro. Documentação e comentários requerem manutenção, assim como código. Assim, documentação na forma de teste apresenta a vantagem de alertar sempre que correções se fazem necessárias.
V. Falso. TDD requer que os testes estejam escritos como código e de forma que rodem de maneira automatizada, sem necessidade de atuação humana. Além disso, não é necessário que tudo esteja especificado antes de programar. Testes cobrindo cada nova funcionalidade ou correção de bug devem preceder a implementação do código, mas é favorecido um processo iterativo, e não “waterfall”.
VI. Falso. É encorajado que a refatoração foque no novo código introduzido, mas não há proibição de refatorar outras partes para manter o bom estado da base de código e possivelmente torná-la mais flexível para alterações futuras.

QUESTÃO 3

a) O princípio SOLID violado de maneira direta é o de Inversão de Dependência. Da maneira que a classe Alarm foi escrita, ela depende diretamente da implementação concreta da classe Sensor. Para corrigir isto, o ideal seria que a classe Alarm possuísse dependência a uma interface SensorInterface que por sua vez seria implementada por Sensor. O objeto concreto deve ser então passado no construtor de Alarm, sendo possível alterá-lo por um Sensor mockado, desde que respeite a mesma interface SensorInterface. Assim, há um desacoplamento entre as classes Sensor e Alarm.

b) [Implementado em src/Q3/TireMonitor/LetraB e test/Q3/TireMonitor/LetraB]

c) Modificar a visibilidade dos membros de Alarm para private, de maneira que não sejam acessíveis fora da própria classe. Isto evita dependência de código externo com a implementação da classe, garantido que interações ocorrem apenas através de sua interface.

Antes	Depois
`Sensor sensor = new Sensor();` `boolean alarmOn = false;`	`private SensorInterface sensor;` `private boolean alarmOn = false;`

Mudar o acesso das constantes de Alarm para public static, de maneira que seja visíveis para o restante do programa. Isto permite que demais partes do programa referenciem os valores constantes de maneira clara e semântica como, por exemplo, da maneira realizada nos testes implementados na letra B.

Antes	Depois

`private final double`
`  LowPressureThreshold = 17;`
 	
`private final double`
`  HighPressureThreshold = 21;`
	`public static final double`
	`  LOW_PRESSURE_THRESHOLD = 17;`
	
	`public static final double`
	`  HIGH_PRESSURE_THRESHOLD = 21;`

d) Secundariamente, a alteração também permitiu que a classe Alarm se tornasse aberta a extensão mesmo sem alterar a sua implementação. A classe Alarm funcionará para qualquer objeto que implemente SensorInterface da maneira esperada, permitindo que seja utilizada de várias maneiras, inclusive com sensores de teste, como foi demonstrado com o sensor mockado, com sensores reais e com sensores diferentes, sem necessidade de alteração da implementação. Isto é o chamado Open-Closed Principle de SOLID e leva a reutilização de código.

QUESTÃO 4

a) [Implementado em src/Q4/letraA]

b) [Implementado em test/Q4/letraB]

c) [Implementado em src/Q4/letraC e test/Q4/letraC]