CES-28 Prova 1 - 2017

Sem consulta - individual - com computador - 3h

Dicksiano C. Melo
COMP19

QUESTÃO 1 - CIRCLE X ELLIPSE

a)

Temos que Ellipse é a superclasse e Circle é subclasse. Pela herança, espera-se que os métodos de Ellipse sejam herdados por Circle. Porém, o método mencinado não deveria ser utilizado por Circle, pois ele deixaria de ser um circulo.

b)

O método getRadius() deveria ficar apenas na clase Circle, uma vez que ele não faz sentido para a classe pai. Quanto ao método *stretchMaior(), há 2 possibilidades:*

1) Ser sobrescrito na classe círculo para lançar uma exceção. Isso evitaria o comportamento indesejado e avisaria o usuário que ele está utilizando o método de forma errada.

2) Ser sobrescrito na classe círculo e aumentar o raio do círculo. Essa solução não faz muito sentido, porém permitiria que o usuário utlizasse esse método independente do tipo de Ellipse, seja círculo ou não. Essa solução faz sentido se considerar que o diâmetro de um círculo equivale ao seu eixo maior, depende da interpretação e do uso.

QUESTÃO 2 - TDD

[ ] Podemos dizer que o exemplo a seguir é um bom exemplo de TDD?

Recebemos um código legado bastante grande de um projeto anterior, desenvolvido sem nenhum teste, e refatoramos o mesmo, criando testes. È

iniciado pelo desenvolvimento de testes triviais, passando por testes simples, testes de unidade, até chegar em testes maiores, com o objetivo de nos certificarmos de que o código funciona e posteriormente permitir a evolução e manutenção desse código.

Falso. Pelo conceito de TDD, primeiro são criados testes que falham e depois ocorre a implementação da solução. O problema desse exemplo é que já existe uma solução que ainda não foi testada e isso inverte a lógica do TDD. No TDD, primeiro são criados testes que inicialmente falham, depois é implementada a solução do problema que faz com que os testes passem e depois o código é refatorado. No exemplo, a soulção já está pronta antes dos testes, o que não obedece a lógica do TDD:

1. Criar testes que falhem;

2. Fazer os testes passarem;

3. Refatorar.

[ ] TDD supõe uma serie de ferramentas de desenvolvimento. A comparação do TDD versus um desenvolvimento não-TDD seria muito menos favorável se não existissem ferramentas e IDEs "bonitinhas" para automatizar testes, inclusive facilitar a leitura dos resultados dos mesmos, verificar rapidamente o que passou e não passou, facilitar inclusive varias refatoracoes comuns, e ferramentas de diff e controle de versão para reverter eventuais erros e/ou encontrar as últimas mudanças com data e responsável. Inclusive podemos considerar isso como uma das razoes porque o TDD demorou algumas décadas para aparecer, e não apareceu nos primórdios da computação.

Verdadeiro. A afirmação não é completamente absurda, porém discordo de alguns aspectos. Tanto o desenvolvimento TDD, quanto o desenvolvimento não-TDD se beneficiam das ferramentas citadas. De fato, a TDD necessita de ferramentas que automatizem os testes, pois a cada refatoração deve-se verificar se o software continua passando nos teste. Contudo, o desenvolvimento não-TDD também utiliza esse tipo de ferramenta para automatizar os seus testes, verificar quais testes estão passando e etc. Por fim, o fato do TDD demorar para aparecer pode estar relacionado com a não existência de tais ferramentas. Porém, faz muito sentido pensar que ele demorou a aparecer devido aos fato de sua lógica ser pouco usual, uma vez que em engenharia

costuma-se implementar uma solução e depois testá-la, enquanto no TDD o teste surge antes da solução.

[ ] Refatorações no TDD são relativamente infrequentes, acontecem apenas quando é detectado algum erro que deve ser corrigido. Uma refatoração é sempre retrabalho e o resultado de algum erro humano.

Falso. Ora, a Refatoração faz parte da lógica da metodologia TDD. Veja:

1. *Criar testes que falhem;*
2. *Fazer os testes passarem;*
3. *Refatorar.*

Refatorações são passos esperados na metodologia TDD, pois ela parte de uma solução inicial mais simples até chegar ao resultado final e, assim, são necessárias refatorações constantes.

[ ] Há alguns casos limite tão comuns que praticamente sempre devemos testar pelo menos vários deles, especialmente quando se usam estruturas de dados. Caso vazio, cheio, apenas um elemento, ultimo e primeiro, usar o índice zero versus índice 1, etc. Para algumas estruturas de dados, pode também ser importante testar os casos de número de elementos par e ímpar, ou entrada ordenada e desordenada. Quando se implementa uma pilha, por exemplo, testar pelo menos algumas dessas condições deve ser um reflexo automático para o programador TDD.

Verdadeiro. Testar o software sobre várias condições é crucial para garantir o sucesso do projeto. Uma vez que durante as condições de uso reais, as estrurutas de dados poderão ser requisitadas por esses tipos de entradas, é fundamental testar essas situações antes que o software entre em pleno funcionamento. Uma vez que o programador TDD desenvolve o projeto a partir dos testes, espera-se que ele tenha em mente essas situações e crie testes para elas.

PARTE III - IMPLEMENTAÇÃO

[IMPLEMENTAÇÃO] – QUESTÃO 3 UM BAR COM MAU CHEIRO.

Meu design permite divide as responsabilidades:
1) A classe Drink representa as bebidas que são preparadas misturando vários ingredientes
2) A classe Beverage representa as bebidas que já estão prontas, por exemplo, uma cerveja ou um refrigerante.
3) A classe Item serve como interface entre os dois tipos de bebidas. Além disso, caso o Pub deseje vender comida, cigarros ou outros items, essa classe permite facilmente expandir.
4) As responsabilidades do método `computeCost()` foram divididas. O método `isManyDrinks()` checa se o cliente está pedindo muitas bebidas e o método `hasDiscount()` checa se o cliente possui desconto. Essas duas mudanças permitem que o código fique mais limpo e mais legível. Em vez de usar vários if-else para checar o preço da bebida, o conjunto de items fica dentro de uma ArrayList. Assim, é permitido adicionar novas bebidas.
5) É possível modificar o nome e o preço dos items e no caso dos Drinks é possível modificar sua composição, ou seja, adicionar ou retirar ingredientes.
6) É estranho que um preço seja inteiro. Eles devem ser double. Para não alterar os testes eu usei a função *Math.ceil() apenas no valor do resultado de `computeCost()`. Os outros preços são double.*

[IMPLEMENTAÇÃO] – QUESTÃO 4 –CONTROLE POSITIVO DE TRENS.

a) Teste a inicialização do objeto **ControladorPTC**. **(1.0 PT).**
b) Construa um caso de teste, quando o trem não se encontra em um cruzamento, ou seja, o método ***isCruzamento()*** de **Sensor** retorna falso. Verifique o comportamento se deu certo. **(1.0 PT).**
c) Construa um caso de teste, quando o trem se encontra em um cruzamento e a velocidade é superior 100Km/h, ou seja, o método ***isCruzamento()*** de **Sensor** retorna verdadeiro. Além disso, o usuário localizado no Painel do Condutor deve informar que leu a mensagem, ou seja, o retorno do método ***enviaMsgPrioritariaPainel()*** deve ser verdadeiro. Verifique o comportamento se deu certo. **(1.0 PT).**
d) Construa um caso de teste, quando o trem se encontra em um cruzamento e a velocidade é inferior a 20Km/h, ou seja, o método ***isCruzamento()*** de **Sensor** retorna verdadeiro. Além disso, o usuário localizado no Painel do Condutor nãp deve confirmar a leitura da mensagem, ou seja, o retorno do método ***enviaMsgPrioritariaPainel()*** deve ser falso. Verifique o comportamento se deu certo. **(2.0 PT).**

Está no arquivo!