**Revisão para prova – Testes de Software**

**O que é teste de unidade**

Teste de unidade consiste em um processo de isolamento de uma unidade específica de um software, com o objetivo de criar um teste que posas validar uma operação unitária desse módulo, ou seja, criar uma validação para uma unidade extremamente específica de uma parte do software.

**1. O que é Mock?**

Mock consiste em uma prática no qual se cria uma imitação e um objeto real para que seja possível realizar testes de uma unidade sem que seja necessário que suas dependências tenham de ser executadas, isolando o modulo testado através dessa técnica de criar um objeto fake que simula o comportamento de outro objeto do sistema.

O mock facilita o processo de teste pois ele simula comportamentos complexos de um sistema, abstraindo a implementação de partes do sistema, o que facilita com que processos como TDD sejam implementados para processo de testagem de software e garantia de que o escopo do teste de unidade seja atendido.

**2. O que é Stub?**

Enquanto um stub apenas provê respostas prontas para as chamadas que serão feitas durante o teste.

**3. O que é mock object?**

Mock Object consiste em um objeto que simula um comportamento específico de uma parte do sistema para realizar testes sem a necessidade de integração das funcionalidades, gerando um objeto fake que consegue simular um comportamento.

**4. Quando devemos usar mock?**

Devemos usar Mock para realizar testes de unidade em uma aplicação visto que ela possibilita com que se crie objetos que simulem certos comportamentos da aplicação para que se possa testar desde casos simples até casos mais complexos e difíceis de se replicar sem toda a complexidade de reproduzir isso através dos processos normais.

**5. Apresente um exemplo de código de uso de mock**

A screenshot of a computer program

Description automatically generated with low confidence

**6. O que é refatoração?**

Refatoração consiste no processo de reescrita/reestruturação de um código, sem que haja mudança na interface ou na resposta dessa feature, ou seja, uma modificação na maneira em que se implementa a lógica por trás de uma feature sem que se modique suas interfaces de comunicação.

**7. Quando devemos refatorar um código?**

Devemos refatorar um código para corrigir algum bug, otimizar algum trecho ou simplesmente melhorar a legibilidade de um trecho de código.

**8. Apresente e explique 5 refatorações**

1. O extract method consiste em, ao encontrar um trecho de código dentro de uma função, que executa alguma atividade específica e que pode ser extraída daquele método, se remova aquele trecho e crie uma função que irá executar exatamente aquela função, só que de maneira mais organizada, legível, reutilizável e menos acoplada.
2. O hide delegate method consiste na criação de uma função intermediária que irá esconder a chamada de uma funcionalidade pela outra, de maneira a reduzir o acoplamento entre elas.
3. O método de refatoração self encapsulae method consiste na criação de methodos `get` e `set` para acesso e manipulação de dados na aplicação
4. O método de refatoração decompose condicional consiste na quebra de uma operação condicional muito complexa em sub-operações mais simples de maneira que a operação final continua sendo executada, porém em mais passos e de forma mais legível.
5. O rename method consiste em um método de refatoração em que o nome de um método e alterado para uma definição mais específica do significado do que aquela função realmente deve executar

**9. O que é Bad Smell?**

Bad smells são maus cheiros, ou seja, é qualquer sintoma subjetivo do código que pode indicar a existência de um bug ou possível falha que possa existir na aplicação.

**10. Como podemos solucionar um Bad Smell?**

A refatoração de código normalmente é o melhor caminho para se resolver um bad smells visto que, ao se reestruturar e organizar a lógica da aplicação, os sintomas que indicam a possibilidade da existência de bugs são removidos assim como as possíveis falhas existentes.

**11. Apresente e explique 5 Bad Smells**

* Código duplicado é um bad smell pois ele indica ausência de modularização além da possibilidade de maior acoplamento ou até maior dificuldade de manutenção da função que está duplicada.
* Long/God method é um bad smell visto que a superconcentração de funcionalidades em um único método normalmente é um forte indicador de possa existir um bug naquele código visto que ele apresenta grande acoplamento, pouca modularidade, dente outros aspectos ligado a grande concentração de responsabilidade em um mesmo método.
* Long/God class é um bad smell que aponta que uma classe tem métodos em excess e por isso está lidando com responsabilidades em excesso, provavelmente, de maneira desnecessária.
* Long Parameter List é um bad smell visto que, uma grande quantidade de parâmetros pode ser um indicador de que aquele método, ou necessita de várias funcionalidades ou que está executando funções em excesso, for do seu escopo de atuação.
* Divergent Change consiste em um bad smell que pode apontar excesso de responsabilidades de uma classe, pois, se é necessário fazer múltiplas modificações em uma classe por motivos diferentes, isso aponta um acúmulo de responsabilidades naquele método.

**12. Apresente e explique 3 métricas POO**

* Weighted Methods per Class (WMC)

Faz a contagem dos pesos relativos a uma classe, sendo que esses pesos podem ser atribuidos por vários indicadores como quantidade de linhas de código, complexidade ciclomática, número de parâmetros, dentre outras

* Depth of Inheritance Tree (DIT)

Faz a contagem de quantos níveis de herança existem entre as classes da aplicação com o objetivo de mensurar o quão profundo as relações de herança são nesse sistema.

* Number of Children (NOC)

A quantidade de filhos diretos, herdam diretamente da classe em questão. (Exatamente o contrário do DIT)

* Coupling between Object Classes (CBO)

Faz a contagem das quantidades de classes que são chamadas pela classe em questão.

* Response for a Class (RFC) Lack of Cohesion of Methods (LCOM)

Faz a contagem do número de métodos que seriam executados caso um objeto receba uma mensagem, ou seja chamado.

* **Lack of Cohesian (LOC)**

Faz a contagem do número de métodos em uma classe que fazem acesso a variáveis comuns, ou seja, de acesso público.

**13. Apresente e explique 3 métricas de produto**

As métricas de produtos valiam atributos de um software, e podem ser subdivididos em 2 categorias, sendo as métricas dinâmicas (São verificadas somente com o software em execução, ou seja, precisam da aplicação funcionando) e as métricas estáticas (Podem ser calculadas sem a necessidade de a aplicação estar funcional).

* Fan-in / Fan-out
* Fan-In
  + Consiste na quantidade de funções que chama uma função específica.
* Fan-Out
  + Consiste na quantidade de funções que são chamadas pela função em questão.
* Length of Code

Contabiliza o tamanho de uma aplicação, sendo que, normalmente, quanto maior uma aplicação, mais complexa ela é e mais propensa ela está a ter bugs e erros.

* Cyclomatic Complexity

Mede o fluxo de controle da aplicação, calculando a compreensibilidade do código em questão.

* Vocabulary Size

Faz a contabilização do número de identificadores (Como nome de classes) em um projeto.

* Depth of Conditional Nesting

Contabiliza a quantidade de operações que estão aninhadas de maneira a torná-la mais difícil de ser compreendida.

**14. Apresente uma estratégia utilizando métricas para detectar 3 tipos de bad smells**

Podemos utilizar a estratégia de Length of code como maneira para detectar alguns bad smells como God method, God class e DuplicatedCode. Visto que todos esses bad smells estão associados a repetição e quantidades excessivas de linhas de código na aplicação.

**15. Qual a definição de teste F.I.R.S.T?**

**É um conjunto de características que qualificam um teste:**

1. **Fast (Rápido)**
2. **Isolated (Isolado)**
3. **Repetable (Repetível)**
4. **Self-verifying (Auto-Verificavel)**
5. **Timely – (Oportunos)**

16. O que é TDD?

TDD ou Test Driven Development é uma metodologia de desenvolvimento de software no qual os testes são construídos anterioemnte da aplicação, e que de forma interativa, vão sendo melhorados de acordo com a progressão do desenvolvimento da aplicação.

**17. Quais as implicações de uso do TDD?**

É relizado um desenvolvimento mais orgânico da aplicação, fornecendo resposta rápida para pequenas modificações na aplicação gerando alta coesão e baixo acoplamento na aplicação desenvolvida.

**18. Quais são as três leis do TDD?**

1. Você não deve escrever qualquer implementação funcional antes que  
   você tenha escrito um teste que falhe
2. Você não deve escrever mais que um teste unitário para demonstrar uma  
   falha
3. Você não deve escrever mais do que o necessário para passar por um  
   teste que está falhando

**19. Quais são os oito estágios do TDD?**

* Escreva um teste simples
* Compile. Não há implementação, então falhará
* Implemente apenas o código necessário para fazer o teste compilar
* Execute o teste e veja que ele vai falhar
* Implemente apenas o código necessário para fazer o teste passar
* Execute e veja o teste passar
* Refatore o código para torná-lo mais claro
* Repita a partir do estágio 1

**20. Apresente um exemplo de uso do TDD passo a passo.**

Passo 1: Escrever um teste inicial (falhando)

```java

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result);

}

}

```

Passo 2: Executar o teste e verificá-lo falhar (esperado)

Neste ponto, o teste deve falhar porque a implementação do método `add` ainda não existe.

Passo 3: Escrever a implementação mínima para fazer o teste passar

```java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

```

Passo 4: Executar o teste novamente e verificar se ele passa (esperado)

Neste ponto, o teste deve passar, pois a implementação do método `add` foi adicionada.

Passo 5: Escrever um novo teste para um cenário adicional (falhando)

```java

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result);

}

@Test

public void testSubtraction() {

Calculator calculator = new Calculator();

int result = calculator.subtract(5, 3);

assertEquals(2, result);

}

}

```

Passo 6: Executar os testes e verificar se o novo teste falha (esperado)

Neste ponto, o novo teste deve falhar porque a implementação do método `subtract` ainda não existe.

Passo 7: Escrever a implementação mínima para fazer o novo teste passar

```java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

```

Passo 8: Executar os testes novamente e verificar se todos passam (esperado)

Agora, tanto o teste de adição quanto o teste de subtração devem passar, pois a implementação do método `subtract` foi adicionada.

Passo 9: Refatorar, se necessário

Após a conclusão do TDD, você pode refatorar o código, se necessário, para melhorar sua estrutura, legibilidade ou desempenho. Certifique-se de que os testes continuem passando após a refatoração.

O processo do TDD continua com a repetição desses passos, adicionando novos testes, fazendo-os falhar, escrevendo a implementação mínima para passar e repetindo o ciclo até que a funcionalidade seja completamente implementada.

Lembre-se de que este é apenas um exemplo simples para ilustrar o processo do TDD. Em cenários mais complexos, você pode precisar escrever testes mais abrangentes e lidar com casos de borda adicionais.