**Verificação e Validação de Software**

**Prof. Bernardo Copstein**

**Aluno: Caroline Lewandowski Rodrigues - 20102626** **Turma: 31**

# Estudo Dirigido

(14 questões)

1. **Quais as diferenças entre TDD e o processo de para condução de testes unitários apresentado em aula?**

Os testes unitários apresentados em aula visavam criar e aplicar o drive de teste após desenvolver o código que será testado, e, diferente deste processo, a proposta do TDD é criar os casos de testes antes de iniciar o desenvolvimento: Primeiro criando um teste que irá falhar - Por não ter nenhum código implementado ainda – e então escrevendo um código que fará este teste passar, após isto, deve-se melhorar o código escrito e repetir o processo até estar satisfeito com a implementação.

1. **Em que ponto(s) do processo de TDD você utilizaria as técnicas de geração de casos de teste vistas em aula?**

Algumas técnicas de geração de casos de teste vistas em aula podem ser adaptadas para seguir o processo proposto pelo TDD, desta forma, podem ser utilizadas em todos os pontos do processo em que é demandado algum teste.

1. **Descreva uma aplicação na qual você utilizaria TDD durante o desenvolvimento e uma onde você não usaria. Justifique suas decisões.**

Utilizaria TDD em uma aplicação complexa na qual não há recursos suficientes para desenvolver perfeitamente uma solução, pois ele auxiliaria na elaboração de decisões para iniciar a implementação e permitiria o entendimento sobre o que foi requisitado, enquanto vou avançando conforme me familiarizo com o problema.

Em uma situação de clareza quanto ao problema e a solução que deve ser aplicada, utilizar o TDD não traria benefícios pois seu objetivo é aprender sobre a especificação durante o desenvolvimento, neste caso, aplicá-la não seria significativo para o desenvolvimento.

1. **Apresente cada um dos princípios FIRST de forma sintética.**

Fast – Priorizar receber feedbacks de teste do código rapidamente. Para o caso de estar fazendo testes lentos, deve-se considerar fazer uso de *mocks* para substituir componentes lentos, refazer código para testar partes mais lentas separadamente ou mover os testes mais lentos para outro caso de teste

Isolated – Testes devem ser os mais coesos, independentes e isolados possível. Idealmente, um teste separado deve testar uma funcionalidade ou comportamento separado do sistema.

Repeatable – Teste que mantem o mesmo resultado, não importa quantas vezes executado.

Self-validating – Os testes devem validar os resultados por si só.

Timely – Desenvolvedores devem escrever e rodar testes o quanto possível

1. **Responda: Janete percebeu que dois testes estão com comportamento estranho: quando executados separadamente os dois passam. Entretanto, quando executados em conjunto eles falham. Que princípio(s) FIRST provavelmente não estão sendo seguidos?**

Isolated – Os testes não devem depender de outros testes para rodar, o resultado deve ser o mesmo. Muitas vezes, o detalhe faltando que causa falha nos testes é a configuração de ambiente, para estes casos, os próprios testes devem ser responsáveis por configurá-los.

1. **Considerando as características de um bom caso de teste propostas por Kent Beck, que contribuições adicionais elas trazem em relação aos princípios FIRST?**

Inspiring – Testes sendo aprovados devem passar confiança de que funcionam.

Writable – Simples para escrever, em relação ao custo para ser testado.

Readable - Compreensíveis para os leitores.

Behavioural - Se o comportamento do código mudar, o teste deve ser propenso a mudar também.

Structure-insensitive – Teste não deve mudar resultado se a estrutura do código mudar.

Automated – Rodar sem intervenção humana.

Specific – Se um teste falhar, a causa da falha deve estar óbvia.

Deterministic – Se nada mudar, o resultado também não deve mudar.

Predictive – Se todos os casos de teste passarem, o código deve estar pronto para entrar em produção.

1. **Explique o que vem a ser um “code smell”.**

Termo que indica possíveis sintomas que podem indicar um problema mais complexo no código fonte do sistema.

1. **Descreva cada um dos “code smells” de teste apresentados no texto de maneira resumida.**

Code Duplication - Códigos duplicados reduzem a produtividade pois, se houver alguma mudança, o desenvolvedor deverá replicá-la nas cópias.

Assertion Roulette – Quando um teste falha, os assertions devem ser os primeiros a serem verificados, para o caso de não estarem claros ocorre, é um “sintoma” do code smell.

Resource Optimism – Ocorre quando um recurso já é válido no início da execução, sendo que o teste deve ser o responsável por definir o status correto dele.

Test Run War – Quando dois testes estão em conflito pelo mesmo recurso.

General Fixture – Fixture é um conjunto de valores de entrada usados para componentes do teste, derivam de casos de testes utilizando as técnicas descritas. Em testes complexos, a utilização de diferentes fixtures podem ocasionar problemas.

Indirect Test / Eager Test – Quando uma classe de testa foca em testar muitas classes de uma vez.

Sensitive Equality – Uma assertion ruim pode induzir um teste a falhar quando não devia.

Inappropriate assertions – Uma escolha errada de um assertion pode levar a menos informações sobre a falha apontada, tornando o processo mais difícil.

Mystery Guest – Dependências externas, podendo vir de banco de dados, arquivos no disco ou webservices.

1. **Indique a alternativa correta: “RepoDriller” é um projeto que extrai informações de repositórios Git. Os testes de integração consomem muitos repositórios GIT reais, cada um com uma característica diferente. Por exemplo um repositório contém um “merge commit” outro contém uma operação “revert” etc. Os casos de teste se parecem com o que segue:**

@Test

public void test01() {

// arrange: specific repo

String path = "test-repos/git-4";

// act

TestVisitor visitor = new TestVisitor();

new RepositoryMining()

.in(GitRepository.singleProject(path))

.through(Commits.all())

.process(visitor)

.mine();

// assert

Assert.assertEquals(3, visitor.getVisitedHashes().size());

Assert.assertTrue(visitor.getVisitedHashes().get(2).equals("b8c2"));

Assert.assertTrue(visitor.getVisitedHashes().get(1).equals("375d"));

Assert.assertTrue(visitor.getVisitedHashes().get(0).equals("a1b6"));

}

De que “test smell” esse trecho de código sofre?

1. **Mystery guest**
2. Condition logic in test
3. General fixture
4. Flaky test
5. O código que segue corresponde a um teste automatizado

@Test

public void flightMileage() {

// setup fixture

// exercise contructor

Flight newFlight = new Flight(validFlightNumber);

// verify constructed object

assertEquals(validFlightNumber, newFlight.number);

assertEquals("", newFlight.airlineCode);

assertNull(newFlight.airline);

// setup mileage

newFlight.setMileage(1122);

// exercise mileage translater

int actualKilometres = newFlight.getMileageAsKm();

// verify results

int expectedKilometres = 1810;

assertEquals(expectedKilometres, actualKilometres);

// now try it with a canceled flight

newFlight.cancel();

boolean flightCanceledStatus = newFlight.isCancelled();

assertFalse(flightCanceledStatus);

}

Entretanto o especialista em testes acredita que este teste “não cheira bem” e que pode ser mais bem escrito. Qual dos seguintes itens deve ser a principal preocupação dele?

1. O teste contém código que pode ou não ser executado tornando os testes menos legíveis.
2. **É difícil dizer qual das diversas asserções causam uma falha**
3. O teste depende de recursos externos e tem resultados não determinísticos dependendo de quando e onde executa
4. O leitor do teste não consegue ver a relação de causa e efeito entre o conjunto de casos de teste e a lógica porque parte dele está fora do método de teste.
5. Qual o “code smell” que o trecho de código abaixo apresenta?

@Test

void test1() {

// webservice that communicates with the bank

BankWebService bank = new BankWebService();

User user = new User("d.bergkamp", "nl123");

bank.authenticate(user);

Thread.sleep(5000); // sleep for 5 seconds

double balance = bank.getBalance();

Thread.sleep(2000);

Payment bill = new Payment();

bill.setOrigin(user);

bill.setValue(150.0);

bill.setDescription("Energy bill");

bill.setCode("YHG45LT");

bank.pay(bill);

Thread.sleep(5000);

double newBalance = bank.getBalance();

Thread.sleep(2000);

// new balance should be previous balance - 150

Assertions.assertEquals(newBalance, balance - 150);

}

1. **Flacky test**
2. Test Code duplication
3. Obscure test
4. Long method
5. No código que segue é apresentado um teste real da Apache Commons Lang. Este test foca no método estático “random()” que é responsável por gerar caracteres aleatoriamente. Um detalhe interessante neste teste é o comentário: irá falhar aleatoriamente 1 cada 1000 execuções.

/\*\*

\* Test homogeneity of random strings generated --

\* i.e., test that characters show up with expected frequencies

\* in generated strings. Will fail randomly about 1 in 1000 times.

\* Repeated failures indicate a problem.

\*/

@Test

public void testRandomStringUtilsHomog() {

final String set = "abc";

final char[] chars = set.toCharArray();

String gen = "";

final int[] counts = {0,0,0};

final int[] expected = {200,200,200};

for (int i = 0; i< 100; i++) {

gen = RandomStringUtils.random(6,chars);

for (int j = 0; j < 6; j++) {

switch (gen.charAt(j)) {

case 'a': {counts[0]++; break;}

case 'b': {counts[1]++; break;}

case 'c': {counts[2]++; break;}

default: {fail("generated character not in set");}

}

}

}

// Perform chi-square test with df = 3-1 = 2, testing at .001 level

assertTrue("test homogeneity -- will fail about 1 in 1000 times",

chiSquare(expected,counts) < 13.82);

}

Qual das afirmações que seguem está incorreta em relação ao teste?

1. O teste é “flacky” em função da aleatoriedade na geração de caracteres.
2. **O teste verifica caracteres inválidos e estes caracteres são selecionados na mesma proporção.**
3. O fato de o método ser estático não influencia o fato de ser “flacky”.
4. Para evitar que o teste fosse “flacky” o desenvolvedor deveria ter usado um dublê no lugar da função “random”.
5. Apresente um exemplo de falta de legibilidade de código que pode ser resolvido com o uso do padrão “builder”. Apresente a classe alvo e o teste original. Mostre a aplicação do Builder e como ficou o teste com seu uso.
6. Apresente exemplos de testes que podem se tornar instáveis (“flacky”).