Slides do professor ICO

Conceitos e Processos de Testing

Qualidade do software

- O grau que um produto de software
  - Cumpre os requisitos funcionais definidos
  - Atende às expectativas do cliente em relação aos atributos do sistema
  - Cumpre as melhores práticas na indústria.

Software "Axiomas"

- Pessoas falíveis - software com erros
- Pessoas falíveis sob pressão - software com (mais) erros
- Construção de software complexo - combinatória de falhas

→ Nunca é seguro supor que um pedaço de software é livre de erros.

Ser sistemático em relação à qualidade do software

- Não deixe ao acaso: implementar Software Quality Assurance (SQA)
- SQA é um conjunto de atividades (metodológicas) para controlar e monitorizar o processo de desenvolvimento de software para atingir os objetivos do projeto com um certo nível de confiança em termos de qualidade.
- Software Quality Control (SQC): remover os defeitos. Avalia se produtos de software estão dentro dos padrões de qualidade definidos, recorrendo a inspeções formais e a diferentes tipos de teste.
- SQA != SQC. SQC tem como objetivo detetar e corrigir defeitos, SQA visa prevenir.

→ Práticas SQA

- Testar
- Gestão da configuração do software (gestão de versões)
- Melhoria do código (Avaliações/comentários, partilhas, análise estática)
- Emissão e acompanhamento de tarefas de gestão
- Integração contínua
- Métodos formais

Verificação Vs. Validação

- Verificação: Estamos a implementar o sistema de uma forma certa?
  - Verificar "work-products" face às respetivas especificações
  - Verificar módulos de consistência

- Validação: Estamos a implementar o sistema certo?
  - Verificar "work-products" face às necessidades e expectativas dos utilizadores.

Testa-se para...

- Obter informações sobre o processo de construção
  - Estaremos a fazer tudo bem?
  - Não é um passo final / reprovação processo de verificação

- Gerir riscos (ganhar confiança nos resultados)
  Exemplo: Temos uma cobertura de teste suficiente?

- Responder à questão fundamental:
  "O produto está pronto para lançamento?"

Princípios Gerais - Filosofia de Testes

- Testar mostra a presença de bugs
  ↳ Os testes podem não mostrar que o software é livre de erros.

- Testar exaustivamente é (geralmente) impossível
  ↳ Testar todas as combinações possíveis no código?
  O desafio é saber quando parar.

- Testes precoces
  ↳ Testes que dão informações valiosas sobre o processo de construção
  ↳ Quanto mais cedo o problema é encontrado, menos custa a ser resolvido
  ↳ Se o teste for adiado, existe um grande risco de não ser feito

- O teste é dependente do contexto
  ↳ Diferentes circunstâncias implicam diferentes testes

Níveis de Teste

- Unit Testing (Testes Unitários)
  - Units - programas/módulos desenvolvidos em isolamento
  - O código escrito para a unidade atende às suas especificações, antes da sua integração com outras unidades.
  - O teste unitário requer acesso ao código que está a ser testado
  - Objetos de teste: Programas (módulos "fine-grain")

- Integration Testing (Testes de Integração)
  - Agrupar as unidades para criar o sistema
  - Objetivo: Expor os defeitos nas interfaces e nas interações entre os componentes integrados do sistema.
  - Objetos de teste: Principalmente o interface do código

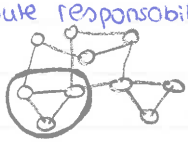
- System Testing (Testes de sistema)
  - Focado no comportamento de todo o sistema/produto
  - Geralmente realizado por uma equipa que é independente do processo de desenvolvimento
  - Nenhuma referência ao código
  - Objeto de teste: sistema

- Acceptance Testing (Testes de Aceitação)
  - Juntar os utilizadores finais para validar que o sistema irá funcionar de acordo com as suas expetativas.
  - Objeto de teste: Sistema completamente integrado, formulários e relatórios.

→ Unit - menor componente que pode ser compilado (classes, componentes, sub sistemas)

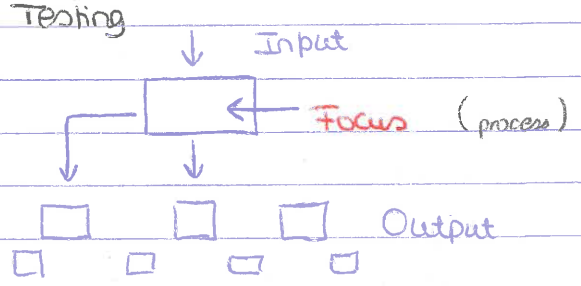
→ Goal - Verifica o funcionamento isoladamente de partes do software que são separadamente testáveis. Processamento de precisão: input - process - output

→ Testing - Normalmente responsabilidade do desenvolvedor (exceto em situações críticas)



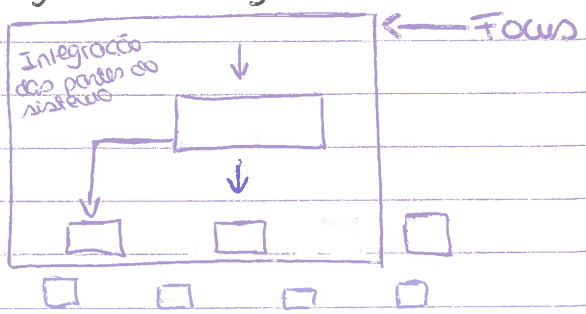
→ White Box ou Black Box - pelo desenvolvedor da unidade ou por um "tester" especial

Unit Testing



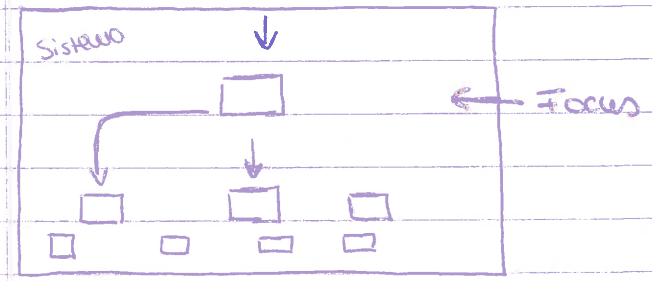
→ Ênfase na unidade
- Funciona como esperado?
- Produz o esperado?

Integration Testing



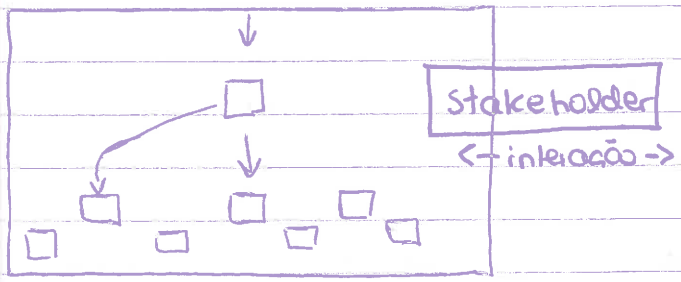
→ Ênfase na integração
- As partes interagem corretamente?
- Funciona como esperado?
- Produz o esperado?

System Testing



→ Ênfase no sistema
- Funciona como esperado?
- Produz o esperado?

Acceptance Testing



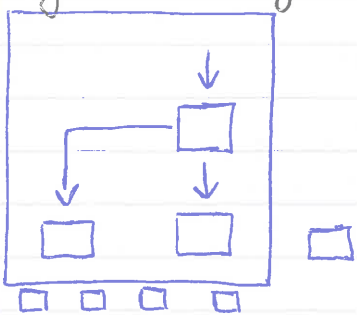
→ Enfâse no stakeholder
- O stakeholder está satisfeito?

Estará bom o suficiente? – Acceptance Testing

- Deve estar concluído com êxito
  - Antes que o produto seja lançado ou que substitua uma versão anterior
  - A conclusão pode ser uma exigência contratual antes do sistema ser pago.

- Closed Box: pelo cliente
- O sistema é testado como um todo
- O ênfase é se o sistema cumpre os seus requisitos
- Utiliza informação real em situações reais, com utilizadores reais (administradores e operadores)

Regression Testing

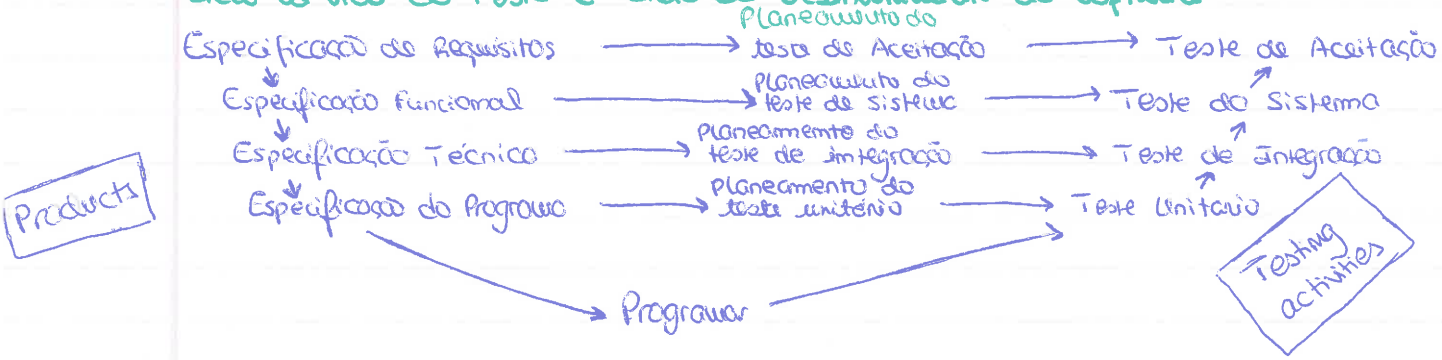


→ Ênfase nas mudanças:
- As partes continuam a interagir corretamente?
- Continua a produzir o esperado?
- Continua a funcionar como esperado?

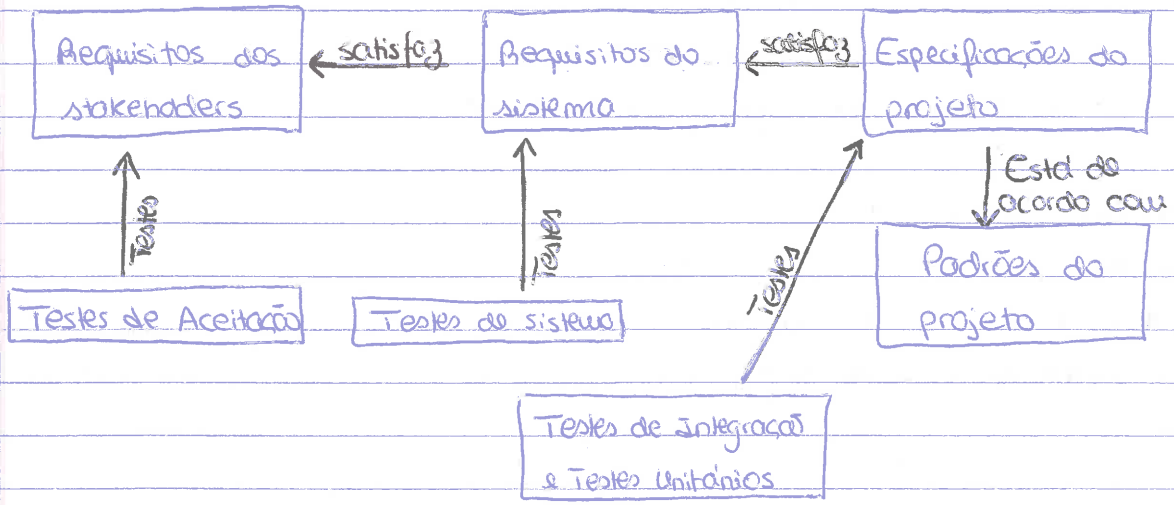
Etapas necessárias para testar o projeto

1. Deliberar sobre a condição de teste, que normalmente seria uma pequena parte da especificação para o nosso software em teste.
2. Projetar um caso de teste que irá verificar a condição de teste.
3. Escrever um procedimento de teste para executar o teste, ou seja, colocá-lo em modos de iniciar o teste, os valores de entrada e verificar o resultado.

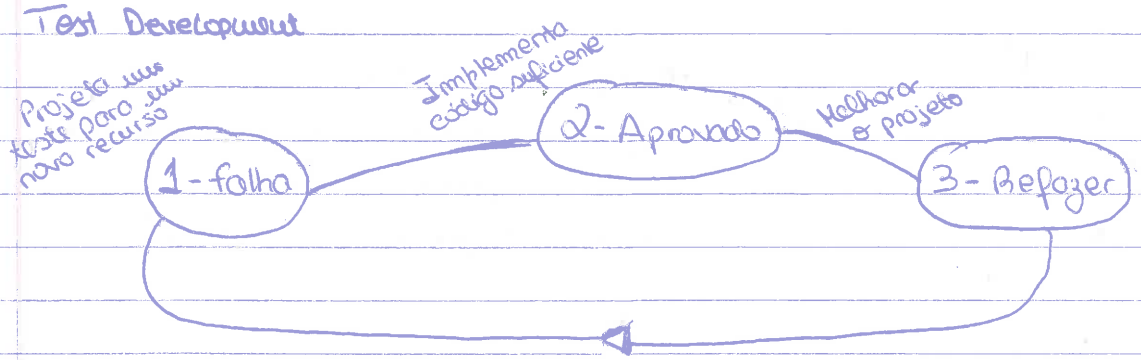
Ciclo de Vida do Teste e Ciclo de Desenvolvimento do Software



Traceability of specifications



Test Development



Testes Unitários utilizando JUnit

Testes Unitários

- Verificar contratos das APIs
- Testes focados e concisos
- Domínio de objetos

Ferramenta de testes JUNit

- ferramenta em Java, integrado em IDEs

JUnit em resumo:

- Testes separados em instâncias de classes e em "load classes" para cada teste unitário - para evitar efeitos colaterais
- Anotações JUnit para fornecer inicialização e recuperação de métodos
  @Before, @BeforeClass, @After, @AfterClass
- Variedade de métodos "assert" para tornar fácil a verificação dos resultados dos seus testes.
- Integração com as principais ferramentas de compilação (Ant, Maven) e IDEs (Eclipse, Netbeans)

→ JUnit Workflow 1

- Criação de casos de teste com a anotação @Test nas classes de teste

```
@Test
public void addiction () {
   assertEquals ("Error: Wrong Outcome adding 7+5:", 12,
      simpleMath.add (7,5));
}

@Test
public void subtraction() {
   assertEquals (9, simpleMath.substrat (12,3));
}
```

Classe Assert

- assertArrayEquals ("message", A, B) → Confirma a igualdade dos arrays A e B.
- assertEquals ("message", A, B) → Confirma a igualdade dos objetos A e B. Este assert invoca o método equals() sobre o primeiro objeto em relação ao segundo.
- assertSame ("message", A, B) → Confirma se o objeto A e B são o mesmo objeto. Considerando que, o método assert anterior verifica se A e B têm o mesmo valor (usando o método equals) e o método assertSame para verificar se o objeto A e B são os mesmos, utilizando o operador ==.
- assertTrue ("message", A) → Confirma se a condição A é verdadeira.
- assertNotNull ("message", A) → Confirma se o objeto A não está nulo.

→ Junit Workflow 2 - Utensílios

```
@Before
public void runBeforeEveryTest() {
    simpleMath = new SimpleMath();
}

@After
public void runAfterEveryTest() {
    simpleMath = null;
}
```

Anotações e suas Descrições

- @Test - A anotação Test informa ao Junit que o método public void ao qual está ligado pode ser executado, como caso de teste.
- @Before - Vários testes precisam de objetos semelhantes criados antes de serem executados. Anotar um método public void com @Before faz com que esse método seja executado antes de cada método de teste.
- @After - Se alocarmos recursos externos num método Before, precisamos de liberto-los depois de testá-los. Anotar um método public void com @After faz com que esse método seja executado depois do método teste.
- @BeforeClass - Anotar um método public static void com @BeforeClass faz com que ele seja executado uma vez antes de qualquer um dos métodos de teste na classe.

@AfterClass - fará' executar o método depois de todos os testes terem sido concluídos. Pode ser utilizado para actividades de limpeza.

@Ignore - A anotação Ignore é usada para ignorar o teste e esse teste não será executado.

→ Junit Workflow 3 - Espera de Excepções e "timeOut"

```
@Test ( expected = ArithmeticException.class )
public void divisionWithException (){
        // divide por zero
simpleMath. divide (1,0);
}

@Test ( timeout = 1000 )
public void infinity (){
  while (true)
    ;
}
```

Testes com Parâmetros

- Anotar test class com @RunWith (Parameterized.class)
- Criar uma anotação no método public static com @Parameters que retorna colecção de objetos como conjunto de dados de teste.
  - @Parameters public static java.util.Collection.xxxxxx ();
  - Os elementos da colecção são arrays (com o mesmo length)
- Criar um construtor público que leva o que é equivalente a uma linha de dados de teste.
  - O número de elementos do array deve coincidir com os parâmetros no construtor
- Criar uma variável de instância para cada "coluna" dos dados de teste.

→ O test case será chamado uma vez por cada linha de código.

Contrato Stack (Pilha)

push (x): adiciona o elemento x ao topo

pop: remove o elemento do topo

peek: retorna o elemento do topo - sem retirá-lo

size: retorna o número de elementos na pilha

isEmpty: retorna se a pilha tem ou não elementos (True ou false)

Testes Unitários na implementação Stack

- Verificar o contrato do stack
  - A stack está vazia na implementação
  - A stack tem tamanho 0 na construção
  - Depois de "n" pushes para a stack vazia, n>0, a stack não está vazia e o seu tamanho é n.
  - Se houver algum pop, o tamanho da stack reduz um elemento.
  - Se houver algum peek, o valor devolvido é x e o tamanho mantém-se.
  - Se o tamanho da stack for "m" e houver "n" pops, a stack é vazia e o tamanho é 0.
  - Fazer pop numa stack vazia vai lançar um "NoSuchElementException"
  - Fazer peek numa stack vazia vai lançar um "NOSuchElementException"
  - Se a stack for limitada, e estiver completamente cheia, fazer um push vai lançar um IllegalStateException.

Test Suite (Conjunto de testes)

Test Suites e IDEs

- Internamente, o Suite padrão cria uma instância para cada método @Test. Em seguida, o Junit executa todos os métodos @Test independentemente dos outros para evitar efeitos colaterais.
- O suite padrão criado se nenhum declarado (executar todos @Test)
- A melhor prática é o lançamento de excepções para o código teste que ainda não foi implementado.
- As anotações @BeforeClass / @AfterClass devem ser públicos e estáticos. Será executado apenas uma vez no caso de teste.

Melhores Práticas

Um aspeto fundamental nos testes unitários é o seu "textura fino" (?). Um teste unitário examina de forma independente cada objeto.

- Quando um objeto interage com outros objetos complexos, podemos rodear o objeto sob teste com objetos de teste previsíveis.
- assertXxxxx (useful message on fail, expected, actual)
- Escolher nomes de métodos com nexo.

Anti-Padrão: Não combinar métodos de teste

- Cada teste unitário é igual a um método @Test
  - Se precisarmos de usar o mesmo bloco de código em mais de um teste, o melhor é extraí-lo como "Utilitário".
  - Se todos os métodos podem partilhar código, o melhor é colocar dentro de "Fixture".