Teste e Qualidade de Software

Resumos
2015/2016

João Alegria | 68661

Resumo Geral

Teste e Qualidade de Software

Qualidade de Software

Qualidade - Grau em que um componente, sistema ou processo atende requisitos especificados e/ou necessidades e expectativas do utilizador/cliente.

Qualidade de Software - A totalidade das funcionalidades e características de um produto de software que afetam a sua capacidade de satisfazer necessidades explícitas ou implícitas.

Fatores Internos e Externos de Qualidade (S. McConnell)	
Externos (aqueles que os utilizadores estão cientes)	**Internos** (aqueles em que o programador está ciente)
Correção (construído livre de falhas)	Manutenção
Usabilidade	Flexibilidade + Portabilidade (adaptável a novos utilizadores ou ambientes)
Eficiência (uso de recursos)	Reutilização (partes podem ser reutilizáveis)
Confiável (longos tempos entre falhas)	Legibilidade e compreensão
Integridade (estado é sempre constante)	Testabilidade
Adaptabilidade (utilizável em novos contextos)	
Robustez (contra adversidades - recuperação de exceções)	

Modelo de Qualidade: Fatores e Funcionalidade	
Funcionais	**Não-Funcionais**
Conveniência: o software é adequado para as tarefas destinadas	Confiável: Quantidade de tempo que o software está disponível para uso
Precisão: resultados/saídas estão corretas e precisas	Usabilidade: avaliação da facilidade de interação do software
Interoperabilidade: capacidade de um componente de software que interaja com outros componentes ou sistemas	Eficiência: software utiliza os recursos do sistema
Segurança: resistente a acessos não intencionais ou modificações	Manutenção: Fácil de reparar/corrigir
	Portabilidade: facilidade de um software correr noutro ambiente

Qualidade de Software: Definição de Trabalho

Grau em que o produto de software:

- cumpre os requisitos funcionais definidos
- atende às expectativas do cliente w.r.t para os atributos do sistema
- atende às melhores práticas da indústria

Qualidade de Software: Sistematização

- **Software Quality Assurance (SQA) / Garantia de Qualidade e Software**
  - conjunto de atividades (metodologia) para controlar e monitorar 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) / Controlo de Qualidade de Software**
  - Remover os defeitos. Avalia se os produtos de um software estão dentro dos padrões de qualidade definidos, recorrendo a inspeções formais e diferentes tipos de testes.

- **SQA != SQC**
  - SQC tem como objetivo detetar e corrigir defeitos
  - SQA visa corrigi-los

Práticas SQA

- Testes
- Gestão e configuração de software (Gestão de versões)
- Melhorias de Código (análise estáticas, …)
- Emissão e acompanhamento de tarefas de gestão
- Integração Contínua
- Métodos Formais

Verificação e Validação	
Verificação	**Validação**
Estamos a desenvolver o sistema da maneira certa?	Estamos a desenvolver o sistema correto?
Verificar a consistência dos módulos (verificar se estamos a desenvolver as *features* corretas para resolver o problema)	Verificar se o sistema atende às expectativas do cliente

Melhoria do código: Análise Estática e Refactoring

Code Refactoring - Refactoring é uma técnica disciplinada para reestruturar um pedaço de código já existente, alterando a sua estrutura interna sem alterar os comportamentos externos.

- Refactoring altera a arquitetura, mas não as funcionalidades
  - Série de pequenas transformações, preservando o funcionamento e correção
    - Melhor estrutura para a concepção atual da arquitetura
    - Redução da complexidade para uma melhor compreensão
    - Remoção de repetições desnecessárias
    - Melhorar o desempenho do código.
- Código mais limpo / mais fácil de entender e manter

- **Exemplos:**
  - Adição de parametros;
  - Substituir condições por polimorfismo
  - Extrair interface
  - Extrair (código duplicado dentro de um) método

"Bad Smells" (Anti-Padrões)

- **Exemplos:**
  - Duplicate Code ➔ Extract method
  - Long Method ➔ Extract method
  - Data Class ➔ Encapsulate field
  - Feature Envy ➔ Move method
  - Long Parameter List ➔ Replace Parameter with Method Call
  - Large Class ➔ Extract class, Extract Interface

Casos comuns de Refactoring

- **Exemplos:**
  - **Extract method** - Seleciona parte do método para formar um novo, substituindo a seleção chamando para o novo
  - **Extract interface** - Cria uma nova interface usando alguns métodos de uma classe, que seguidamente, irá implementar a nova interface
  - **Encapsulate field** - Cria os métodos get e set para os campos e usa apenas aqueles para aceder ao mesmo

NetBeans Refactor

- Rename
- Change Method Parameters
- Encapsulate Fields
- Move / Copy Class
- Extract Interface
- Safely Delete

Análise Estática - Análise estática do código consiste em observar o código base para detetar possíveis falhas sem executá-lo

- **Exemplos:**
  - nomes de variáveis e métodos não descritivos, muito longos ou curtos

Problemas relatados na análise estática

- referenciar uma variável com um valor não definido
- variáveis que não são utilizadas
- código morto / inacessível
- violação dos padrões de programação
- vulnerabilidades de segurança

— — SONARQUBE — —

- Ferramenta mais usada para SCA pois consegue avaliar a arquitetura, uso comentários, duplicações, testes unitários, complexidade, bugs, regras de código, etc
- Pode ser automatizada cm a execução do projeto através de plugins do IDE ou num servidor CI
- O painel do SonarQube é dividido em várias secções onde são apresentados diferentes informações gráficas e estatísticas sobre analises estáticas, código duplicado e cobertura.
- Este painel é apenas informativo não tendo permissões para alterar código

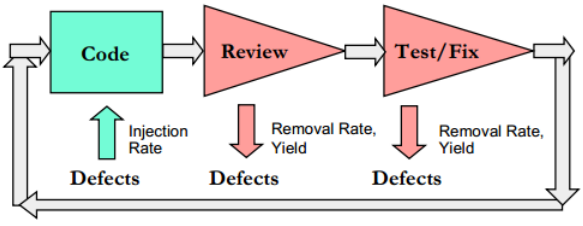
Cobertura de Código

- Parte do código que está a ser testada (por testes unitários)
  - LOC (lines of code)

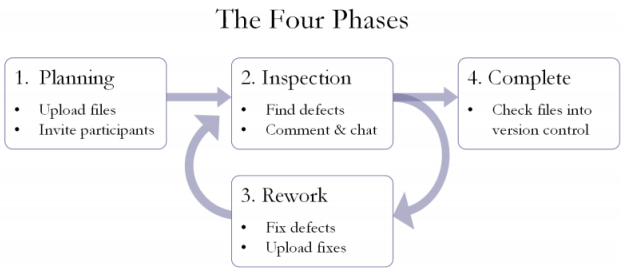
Melhoria do código: Revisão de Código

Code Review - Destina-se a encontrar e a corrigir erros que se encontram na fase inicial do programa, melhorando a qualidade de software.

- **Objetivos:**
  - Descobrir erros em funções, lógica ou outro tipo de representação
  - Verificar que o software sob avaliação atende os seus requisitos
  - Desenvolver projetos mais manejáveis



Ciclo de Vida

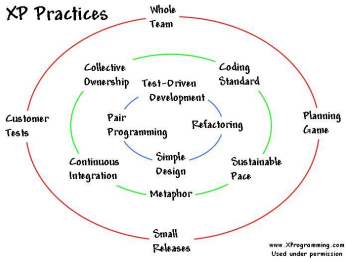


Problemas a encontrar com a utilização do Code Review

- Desvios dos padrões
- Requisitos de defeitos: os requisitos são ambíguos, ou existem elementos em falta
- Defeitos na arquitetura: a estrutura não cumpre os requisitos
- Manutenção insuficiente: o código é muito complexo para se manter
- Especificações da interface incorretas

Técnicas superficiais para a Revisão de Código

- **Over-the-Shoulder**
  - o desenvolvedor e autor revêm o código em conjunto e o autor acompanha as alterações do código
- **Email pass-around**
  - Alerta por e-mail de alterações de código
- **Pair Programming**
  - Dois autores desenvolvem código em conjunto
  - Revisão de código contínua
  - Não posso verificar o código que "eu" fiz
  - Uma forma de "revisão de código contínua"



- **Tool-Assisted Review**
  - Autores e revisores usam ferramentas especializadas, projetadas para revisão de código. Recolhem alterações, discutem a sustentabilidade e visualização as diferenças

Melhoria do código: Tratamento de Erros (Java)

Estratégias de Tratamento em Java

-**Programação Defensiva**
  - Evitar causas de erros
  - Verificar se existe "casos problemáticos" antes de invocar funções
  - Verificar os resultados
- **Programação / Arquitetura por contrato**
  - Adicionar restrições específicas no código
- **Lógica de exceções separada do fluxo "normal"**

Exceções em Java

- Lidar com circunstâncias e exceções
- Separado do fluxo do programa
- Normalmente são mal utilizadas
- Utilizar exceções economiza-se tempo e problemas

• ***Melhores práticas para manipular as exceções***

- 3 tipos de throwables:
  - Checked Exceptions
    - lançar uma checked exception, passamos a obrigação de tratar o erro
    - evitar o uso desnecessário de checked exceptions
  - Runtime Exceptions
    - indicam erros de programação
    - indicam pré-condições foram violadas
    - todas as "unchecked throwables" implementadas devem estender `RunTimeException`
  - Erros
    - Convenção de que os erros são reservados pela JVM

Software de Gestão e Configuração

Version Control System (VCS)

- um sistema que regista alterações para um ficheiro ou conjunto de ficheiros ao longo do tempo para que recupere versões específicas mais tarde

Feature-Branch Workflow

- Usa um repositório central e um ramo (*branch*) mestre
- Os desenvolvedores criam um novo ramo sempre que começam a trabalhar num novo recurso.

Pull Requests

- Além de isolar o desenvolvimento de recursos, os branches tornam possível discutir mudanças através de *pull requests*. Quando alguém concluí um recurso, não é feito *merge* imediatamente para o *master*. Em vez disso, é feito *push* ao branch de recurso para o servidor central e é feito um pedido de *pull request* para fundir as alterações em *master*. Isto dá aos desenvolvedores a oportunidade de rever as alterações antes de se tornarem uma parte principal do código-base.

Teste de Software

Teste - Tentativa de provar que alguma característica específica do software e/ou hardware está ausente de modo que este pode ser estabelecido por meios humanos perceptíveis.

Teste de Software - O processo de execução de um sistema de software para determinar se ele corresponde a sua especificação no ambiente pretendido

Vocabulário

• **Condição de Teste** - Alguma característica do software pode verificar um teste ou um conjunto de testes (por exemplo: uma função, transação, feature, atributo de qualidade, elemento estrutural)
• **Caso de Teste** - Recebe o sistema como ponto de partida para o teste (condições de execução), em seguida, aplica-se um conjunto de valores de entrada que deve alcançar um determinado resultado (resultado esperado), e sai do sistema como ponto final (pós-condição de execução)
• **Procedimento** - Ações necessárias em sequência para executar um teste

A Natureza dos testes

Teste ≠ Depuração
• **Depuração** - Processo que os developers passam por identificar a causa de erros ou defeitos no código e fazer as correções
• **Teste** - Exploração sistemática de um componente ou sistema, com o principal objetivo de encontrar e relatar defeitos

Estática vs. Dinâmica
• **Estática** - Teste onde o código não é exercido
• **Dinâmico** - Execução de software, utilizando os dados de entrada

Níveis/Fases de Teste

- **Testes Unitários** - Cada módulo faz o que é suposto fazer?
  - Fase em que se testam as menores unidades de software desenvolvidas (métodos, classes, pequenos trechos de código). O objetivo é encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo.

- **Testes de Integração** - O resultado é o esperado quando as partes são colocadas juntas?
  - Na fase de teste de integração, o objetivo é encontrar falhas provenientes da integração interna dos componentes de um sistema. Geralmente os tipos de falhas encontradas são de transmissão de dados (exemplo: um componente A pode estar aguardando o retorno de um valor X ao executar um método do componente B; porém, B pode retornar um valor Y, gerando uma falha)
  - Objetivo: expor os defeitos nas interfaces e nas interações entre componentes integrados no sistema

- **Testes de Aceitação/Validação** - O programa satisfaz os requisitos?
  - Geralmente, os testes de aceitação são realizados por um grupo restrito de utilizadores finais do sistema, que simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado, de maneira a determinar a satisfação dos critérios reclamados pelo cliente.

- **Testes de Sistema** - Todo o sistema funciona como esperado?
  - Objetivo: Executar o sistema sob ponto de vista de seu utilizador final, percorrendo todas as funcionalidades em busca de falhas em relação aos objetivos originais.

Como Testar

- **Teste Funcional**
  - Teste baseado em especificações como testes comportamentais (teste da caixa-preta)
  - Precisa de uma especificação

- **Teste Não-Funcional**
  - Teste para as qualidades do sistema / atributos (ex: usabilidade, performance)
  - Precisa de uma especificação

- **Teste Estrutural**
  - Medir quantos testes foram realizados contra algum conceito estrutural (exemplo: user stories testadas)

- **Teste de Regressão**
  - Verificações que corrige erros não apresentam funcionalidade inesperados no sistema (correções foram bem sucedidas)

Testes Unitários

— — JUNIT — —

Framework de testes unitários em Java, integrada em vários IDEs.

- Separa as classes de testes das classes principais para cada teste unitário para evitar efeitos colaterais
- As anotações JUnit fornecem métodos de recursos de inicialização e recuperação: `@Before, @BeforeClass, @After, @AfterClass`
- A variedade de métodos assert possibilita verificar o resultado dos testes
- Integrado com várias ferramentas (Ant, Maven) e IDEs (Eclipse, NetBeans)

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

   @Test
   public void testAdd() {
      Calculator calculator = new Calculator();
      double result = calculator.add(10, 50);
      assertEquals(60, result, 0);
      }
}
```

JUnit Workflow 1 - Criar testes 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.substract(12,3));
}
```

JUnit Workflow 2 - Fixação: Configuração para o estado do objeto conhecido

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

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

JUnit Workflow 3 - Exceções e *timeout*

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

@Test (timeout = 1000)
public void testVerySlowPrint() {
   messageUtil.printMessage();
}
```

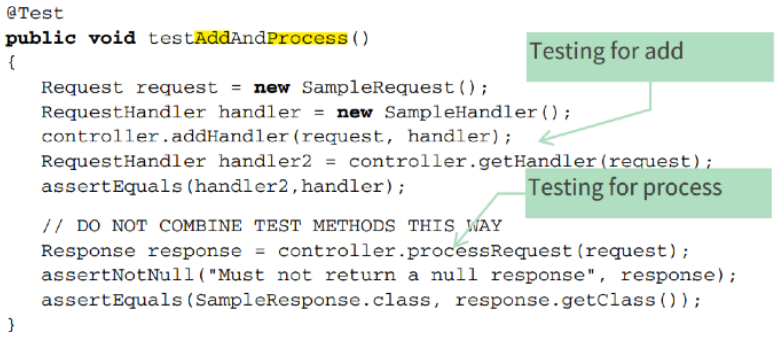
Class Assert	
assertArrayEquals(“message”, A, B)	Confirma a igualdade dos arrays A e B
assertEquals(“message”, A, B)	Confirma a igualdade de objetos A e B. Este assert chama o método equals() no primeiro objeto contra o segundo
assertSame(“message”, A, B)	Confirma se os objetos A e B são o mesmo objeto. Enquanto que o método assert anterior verificava que A e B têm o mesmo valor (usando o método equals), o método assertSame verifica se os objetos A e B são o mesmo objeto (usando o operador = =)
assertTrue(“message”, A)	Confirma se a condição de A é verdadeira
assertNotNull(“message”, A)	Confirma que o objeto A não é nulo

Anotações e suas Descrições	
@Test	A anotação Test indica ao JUnit que o método public void que está anexado pode ser executado como um *caso de teste*
@Before	Vários testes precisam de objetos idênticos criado anteriormente para poderem ser executados. Anotar um método public void com @Before faz com que o método seja executado antes de cada método Test
@After	Se são alocados recursos no método Before, é necessário libertá-los depois que o teste é executado. Anotar um método public void com @After faz com que o método seja executado após o método Test
@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	Esta anotação fará executar o método depois de terem terminados todos os testes. Pode ser usado para realizar atividades de limpeza
@Ignore	A anotação Ignore é utilizada para ignorar um dado teste e esse não será executado

Vocabulário JUnit	
Assert	Permite definir as condições desejadas para testar. Um método assert é silencioso quando a sua proposta é bem-sucedida, mas se a proposição falhar, lança uma exceção
Test	Um método com uma anotação de @Test define um teste. Para executar este método, o JUnit constrói uma nova instância da classe que contém e, em seguida, chama o método anotado
Test Class	Uma classe de teste é um container para os métodos @Test
Suite	A Suite permite agrupar as classes de teste
Runner	A classe Runner executa testes. JUnit 4 é compatível com versões anteriores e irá executar testes do JUnit 3.

Não combinar métodos de testes - Um teste unitário é igual a um método @Test

- Se for necessário usar o mesmo bloco de código em mais de um teste, devemos transformar num utilitário
- Se todos os métodos partilham código, deve-se colocá-lo numa fixação.



Propriedades de um bom teste

- **Automático** - Pode ser executado por uma ferramenta de automação
- **Completo** - Atende os objetivos de cobertura desejados (completo e cuidadoso.
- **Repetitivo** - Capaz de ser executado repetidamente e continuar a produzir os mesmos resultados independentemente do ambiente
- **Independente** - Não depende nem interfere com outros testes

(não se pode confiar num teste unitário para fazer o trabalho de instalação para outro teste unitário)

Resumindo

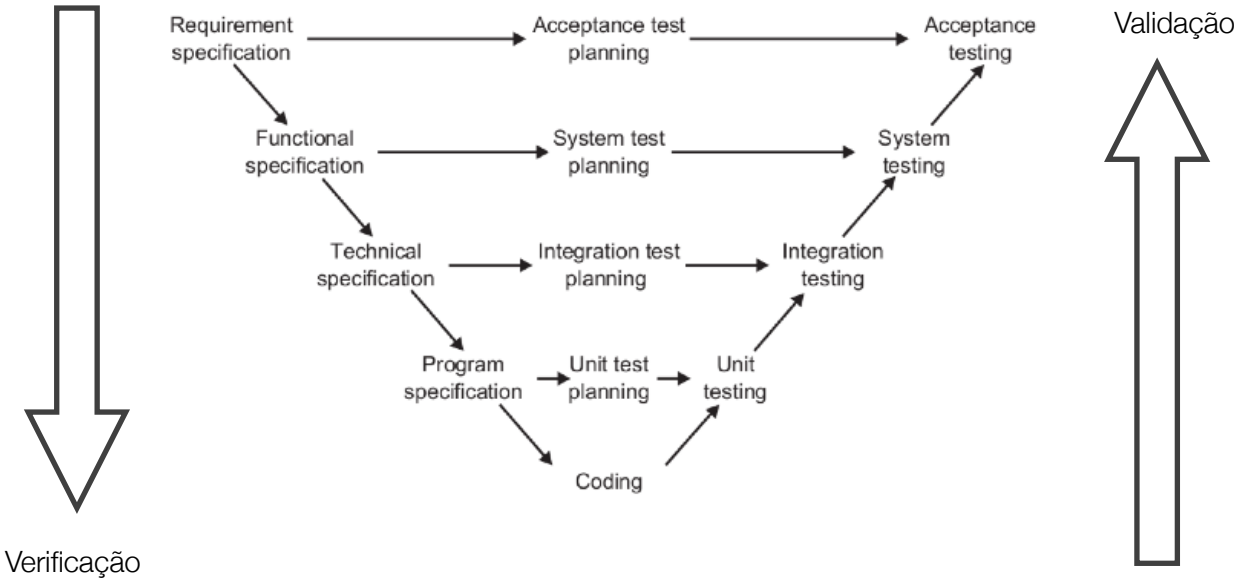
- JUnit cria uma nova instância da classe em teste antes de invocar cada método @Test
- Métodos Assert
  - assertXXX (“mensagem”, expetativa, obtido)
- Três fases:
  - Setup - @Before
  - Verificação - @Test, assertX
  - Destruir - @After

V-Model

- No V-Model a execução dos processos acontece de forma sequencial em forma de V
- Também é conhecido como um modelo de verificação e validação
- O V-Model é uma melhoria do modelo em cascata e baseia-se na associação de uma fase de testes para cada fase de desenvolvimento correspondente. Isto significa que, para cada fase do ciclo de desenvolvimento, há uma fase de teste diretamente associada
- Este é um modelo altamente disciplinado e a próxima fase começa apenas após a finalização da fase anterior

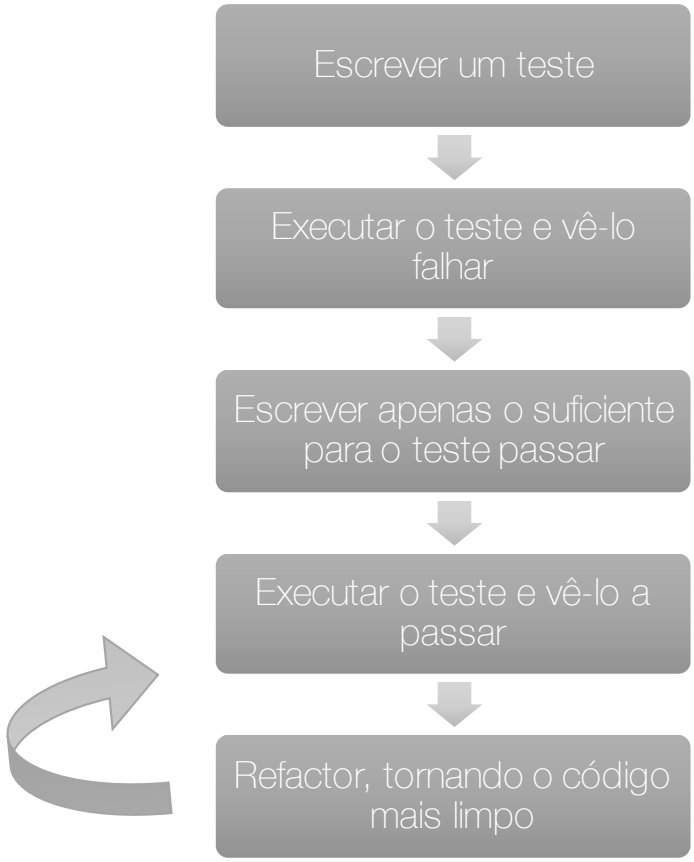
Desvantagens

- Não considera o paralelismo que geralmente ocorre em projetos de maior complexidade
- Não considera as diversas dimensões do projeto
- Há ciclos de revisão em etapas tardias do processo, quando se encontra erros, a sua correção é dispendiosa



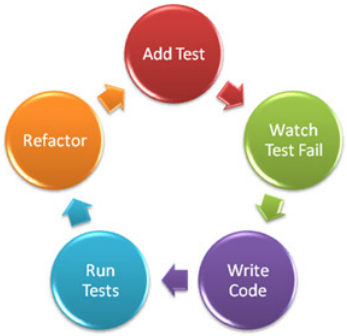
Test Driven Development (TDD) / Desenvolvimento Orientado a Testes

TDD - É uma técnica de desenvolvimento ágil onde o desenvolvimento deve ser orientado a testes, onde cada teste unitário deve ser escrito antes que uma funcionalidade do sistema o seja. O objetivo é fazer com que o teste passe com sucesso, significando que assim a funcionalidade está pronta e conta com garantia de qualidade.



- Técnica de desenvolvimento de software que se baseia num curto circulo de repetições.

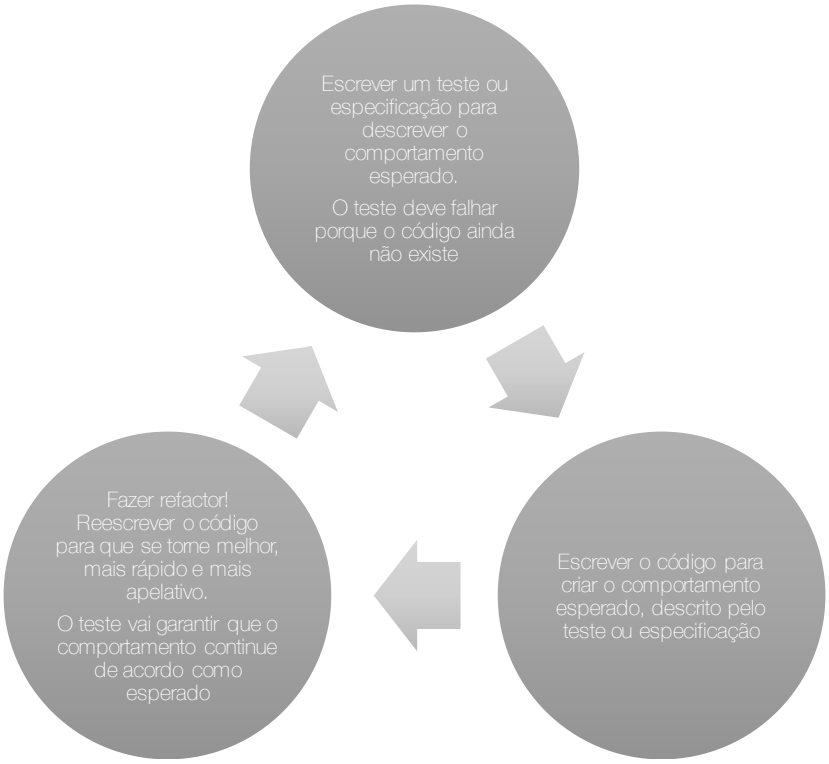
- Adicionar Testes
- Verificar se ocorre falhas
- Escrever código
- Correr testes
- Refactoring o código
- Repetir



Behaviour Driven Development (BDD) / Desenvolvimento Orientado ao Comportamento

BDD - É uma técnica de desenvolvimento ágil que encoraja a colaboração de developers, setores de qualidade e pessoas não técnicas.

- O desenvolvimento deve ser orientado aos comportamentos que o sistema deve apresentar. Desta forma, um comportamento (requisito/especificação) é priorizado em relação ao teste unitário, o que não exclui a execução do fluxo TDD neste processo.



Stories e Scenarios - Unidade básica da funcionalidade e, consequentemente, entrega.

- Captura uma característica do sistema
- Define o âmbito do recurso
- Os critérios de aceitação

- Também são usadas como base para estimarmos o que temos de planear

Stories		
Âmbito da feature		**Critérios de Aceitação**
Título	Descrição da Story	Cenário: Situação de "negócio"
Narrativa	• Como (papel) • Eu quero (feature) • Assim que (benefício)	• Dado (descrever contexto) e (descrever mais contextos) • Quando (dado um acontecimento) • Então (resultado) e (outro resultado)

— — CUCUMBER — —

- O cucumber não é apenas uma ferramenta para automatizar testes de software, é possível escrever cenários que ilustram as regras de um negócio, permitindo envolver analistas de negócios escrevendo cenários antes do desenvolvimento, fazendo com que os desenvolvedores sejam guiados por uma especificação sem ambiguidade.
- Objetivo: Simplificar a comunicação entre todas as partes
- Modo:
    - Os requisitos são expressos usando exemplos concretos
    - Cria exemplos de comportamento que são executáveis
    - Os exemplos são elaborados de forma colaborativa (analistas de negócios, desenvolvedores e testers)
    - Os exemplos podem ser utilizados como testes de aceitação (com etapas de preparação adicionais)

Testes Unitários com Objetos Simulados (Mock Objects)

Estratégias comportamentais em "objetos falsos"

- ***Stubs***
  - Fornece respostas enlatas para chamadas feitas durante o teste
  - Geralmente não responde a todos para qualquer coisa fora do que está em teste
- ***Mocks***
  - Objetos pré-programados com as expectativas, isto é, a especificação das chamadas que são esperadas para receber
  - Verificação compara as chamadas recebidas com as expectativas
- ***In-Container Testing***
  - Containers são ativados para ativar o ambiente de teste
  - Requer mecanismos para implementar e executar testes num container

Stubs vs. Mocks	
Stubs	**Mocks**
- Implementa uma versão simplificada do objeto - Implementação explícita do comportamento / lógica - "Rude"	- Fornece um objeto para responder a parte do contrato do objeto alvo - Nenhuma implementação explícita - Comportamento especificado pelas expetactivas - "Refinado"

Workflow

1. Criar um objeto Mock, dando uma interface
2. Definir expectativas (especificação das chamadas esperadas)
   - os métodos a utilizar, parâmetros e valores de retorno
   - ordem (e tempos) dos métodos de chamada
3. publicar a especificação (EasyMock.replay)
4. Verificar que as expectativas foram satisfeitas (EasyMock.verify)

Quando utilizar objetos Mock?

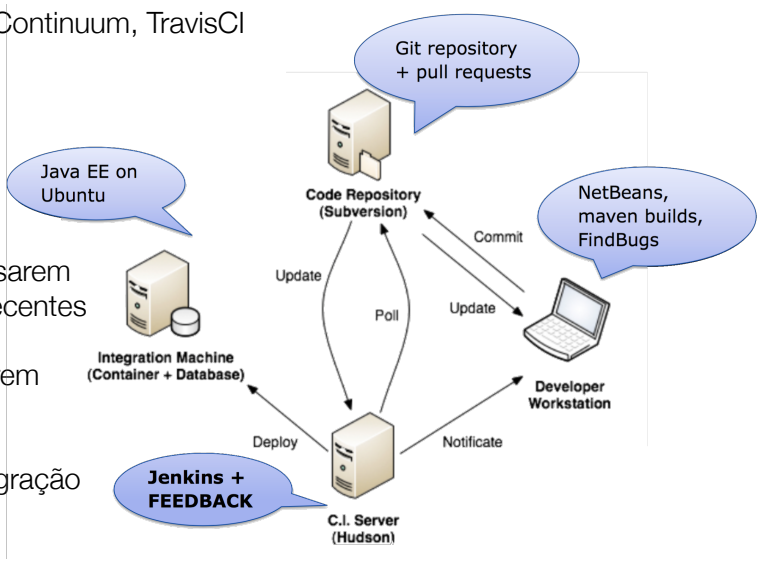
- Resultados não determinísticos
  - hora / temperatura atual
- Estados difíceis de criar ou reproduzir
  - um erro de rede ou da base de dados
- É lento
  - grande serviço de rede
- Algo que ainda não existe ou que pode mudar o comportamento
- Tinha que se incluir informações e métodos exclusivamente para fins de teste

Continuous Integration (CI) / Integração Contínua

Integração Contínua - Consiste em construir um projeto regular e automático com execução de testes automatizados, testes de qualidade, mecanismos de integração e implementação de binários em máquinas de execução ou de repositórios partilhados.
- CI torna o processo de desenvolvimento mais suave, mais previsível e menos arriscado.
- Ferramentas:
  - Hudson / Jenkins, CruiseControl, Apache Continuum, TravisCI

Workflow

5. Checkout / atualizar do SCM
6. Codificar um novo recurso
7. Executar automaticamente na máquina local
   - Repetir os pontos 2 e 3 até os testes passarem
4. Juntar a cópia local com as mais recentes mudanças do SCM
   - Reparar e reconstruir até os testes passarem
5. Commit
6. Executar numa máquina "clean"
   - Correção de bugs e problemas de integração imediatos



Sem feedback, o Continuous Integration é inútil

Práticas de CI (Fowler's)

1. Manter num único repositório
2. Automatizar o Build
3. Tornar o Build automático a nível de testes
4. Todos fazem commit todos os dias
5. Cada commit deve construir uma linha principal na máquina de integração
6. Manter a Build rápida
7. Testar num ambiente de produção clone
8. Torná-lo acessível a todos
9. Todos podem ver o que está a acontecer
10. Automatizar a implementação

Feedback Contínuo

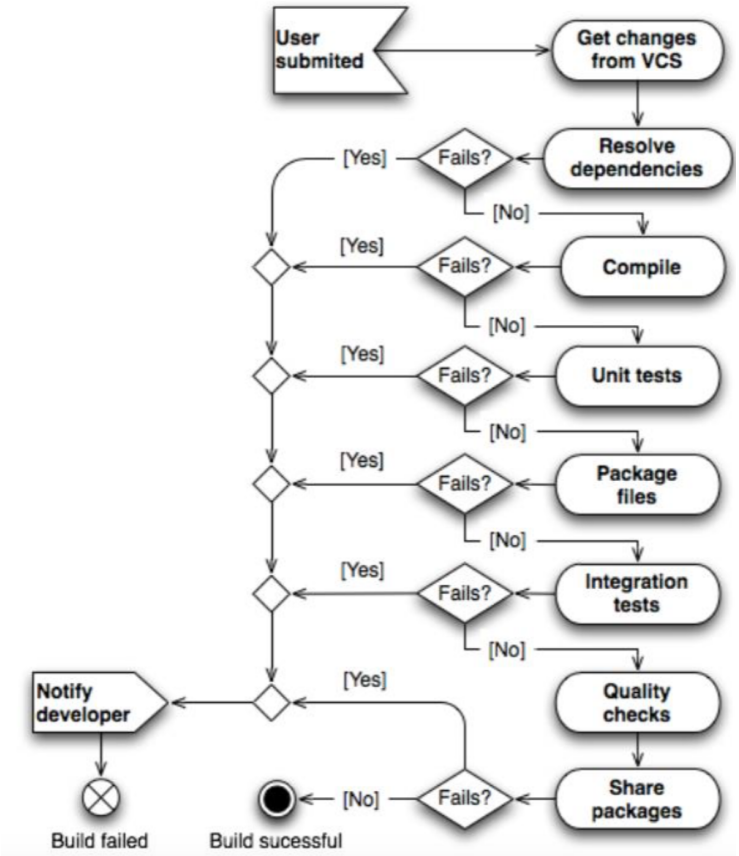
- O processo de integração é o mais frequente e o menos doloroso
- Os erros são mais fáceis de detetar em fases anteriores, perto do ponto onde foram introduzidas
  - O mecanismo de deteção de tais erros torna-se mais simples, porque o passo natural no diagnóstico do problema é verificar qual foi a última alteração submetida
  - Problemas seguidos de commits individuais são mais fáceis de corrigir do que corrigir vários problemas de uma só vez, depois de commits em massa
- Deve haver um mecanismo eficaz que informe automaticamente programadores, testers, administradores de base de dados e gestores sobre o estado da construção
- Feedback tem como objetivo gerar a reação de uma forma mais rápida e precisa

Construção “Contínua”: O Processo de Construção

- O processo de compilação é uma série de etapas que transforma os vários componentes do projeto numa aplicação pronta para ser implantada
- As instruções de compilação estão descritas num ou mais arquivos de descrição
  - POM.xml
- Entrada
  - source-code, código de teste, dependências, documentação, etc
- Saída
  - Arquivos executáveis, documentação do utilizador, bibliotecas, relatórios

Testes Contínuos

- Verificações de qualidade em todos os níveis do sistema que envolvem todos os indivíduos, e não apenas os elementos da equipa de QA
- A maioria dos testes podem ser automatizados e devem ser executados num pipline CI a ser realizado repetidamente:
  - Testes unitários, testes de integração, testes de regressão, testes de sistema, testes de carga e desempenho
- Ferramentas de compilação podem ter um papel fundamental a automatizar testes



— — JENKINS — —

- Ferramenta de integração contínua open-source
- É útil em ambiente organizacional uma vez que possibilita:
  - Builds periódicos
  - Testes automatizados
  - Builds em ambientes diferentes do desenvolvedor
  - Análise de Código

Testes Funcionais com Selenium

— — SELENIUM — —

- Selenium numa aplicação web.
- É um software de testes para aplicações web. O Selenium pode ser usado para automatizar os critérios de aceitação que manipulam a interface de utilizador.
- Possui um ferramenta que grava e apresenta testes sem aprender a linguagem do script, o que permite receber scripts de todas as plataformas. Este envia comandos ao browser a partir dos scripts e retorna os resultados (WebDriver)
- Os testes podem ser construídos em várias linguagens (Java, Ruby, Python, C#)
- Compatível com a maioria dos browsers

Teste de Integração com o Arquillian

— — ARQUILLIAN — —

- Arquillian é uma plataforma de testes funcionais e de integração, que pode ser usada para testar a camada de negócio “business logic” em Java.
- Com o objetivo principal de tornar os testes de integração e os funcionais tão simples quanto escrever testes unitários, o Arquillian traz os testes para o ambiente de execução, o que libera o desenvolvedor da gestão do ambiente de execução.
- O Arquillian suporta a integração com containers Java EE como o Glassfish e containers de servlets como o Tomcat/Jetty, algém de suportar a execução de testes em serviços cloud.
- É possível gerar aplicações para uma variedade de plataformas incluindo Java EE 5 e 6, ambientes de servlets, Embedded EJV e CDI standalone.

Testando JavaEE com Arquillian

- Arquillian faz testes de integração mais fáceis:
- Arquillian traz o teste para o tempo de execução para que seja preciso gerir o tempo de execução do teste (ou a construção).
    - Responsabilidades:
        - Gestão do ciclo de vida dos containers
        - Agrega o caso de teste com as classes e recursos dependentes num arquivo ShrinkWrap
        - A implementação do arquivo(s) para o container(s)
        - Enriquecer o caso de teste, fornecendo as dependências
        - Capturar os resultados e devolvê-los ao runner de teste para relatar

Teste de Desempenho com o JMeter

Testes de Desempenho - Os testes de desempenho são idênticos aos testes de carga mas com o intuito de testar o software a fim de encontrar o seu limite de processamento de dados no seu melhor desempenho. No teste é avaliada a capacidade de resposta em determinados cenários e configurações, determinando assim a sua escalabilidade e confiança.

Testes de Carga - Identifica os níveis máximos os quais um sistema/aplicação pode realizar com sucesso em termos de carga de trabalho e número de utilizadores virtuais.

— — JMETER — —

- JMeter é uma ferramenta utilizada para testes de carga em softwares.
- Para a realização de testes, a ferramenta JMeter disponibiliza diversos tipos de requisições e assertions (para validar o resultado dessas requisições), além de controladores lógicos como loops(ciclos) e controlos condicionais para serem utilizados na construção de planos de teste, que correspondem aos testes funcionais.
- O JMeter disponibiliza também um controlo de threads, chamado Thread Group, no qual é possível configurar o número de threads, a quantidade de vezes que cada thread será executada e o intervalo entre cada execução, que ajuda a realizar os testes de carga.

Elementos JMeter	
Test Plan	Script JMeter
Thread Group	Simula um grupo de utilizadores
Sampler	Ação que causa um request ao servidor e aguarda pela resposta. São processados pela ordem em que aparecem na árvore - HTTP Request ; JDBC Request ; Java Request ; SOAP/XML Request
Timer	Adiciona um intervalo (atraso)
Listener	Componente que mostra os componentes das amostras, o resultado pode ser exibido numa árvore, tabela, grafo ou simplesmente num ficheiro log