# Estudo 2ª Frequência
# MDS

## →Gestão de configurações

Software muda frequentemente:

-Cada versão tem de ser mantida e gerida ;

-Sistemas vistos como séries de versões

Cada versão:

-Implementa alterações

-Correção de falhas

-Adaptações

Gestão de configurações:

-Ferramentas e processos para gerir alterações/mudanças

-Essencial: Facil de perder rasto das alterações , alterações que devem fazer parte de uma versão do sistema

Atividades:

-Gestão de alterações: Manter o "rasto" dos pedidos de alterações feitas pelos clientes

-Gestão de versões: Manter o rasto das várias versões dos componentes do sistema

-Criação do sistema(system building): Processo de construir componentes do sistema, dados e bibliotecas

-Gestão de releases: Prepara software para ser entregue e manter resgisto de todas as versões

Processos Ágeis e Gestão de Configurações:

-Essencial nos processos ágeis: Sistemas alterados várias vezes ; Várias versões por dia

-Versões de Componentes: Projeto/repositorio partilhado com todos os membros

-Processo de Desenvolvimento: Código copiado do repositório para o Desktop(clone)

-Modificação do Código

-Nova versão é testada(Unit Test)

-Código enviado para o repositorio partilhado(push)

Etapas de Desenvolvimento:

-Desenvolvimento: Adicionadas funcionalidades ; Gestão de configurações

-Testes: Versão entregue inteiramente para testes ; Não são adicionadas novas funcionalidades ; Correção de bugs, fixes, melhoria de desempenho, …

-Release: Nova versão entregue ao cliente para utilização ; Podem surgir novas versões da Relase para corrigir erros

Sistemas multi-versão:

-Grandes

-Várias versões funcionais

-Várias versões do sistema: Em diferentes etapas de desenvolvimento

-Podem existir várias equipas de desenvolvimento: Diferentes versões do sistema

Terminologia:

-Baseline: coleção de versões de componentes que fazem um sistema. São controladas, isto é, significa que não podem ser modificadas

-Branching: criação de uma nova codeline de uma versão

-Codeline: é um conjunto de versões de um componente de software

-Configuration(version) Control: é o processo de garantir que as versões do sistema e os componenetes são gravados e mantidos

-Mainline: é uma sequencia de baselines que representam diferentes versões do sistema

-Merging: criação de uma nova versão de um componente de software atraves de merde de versões separadas em diferentes codelines.

-Release: uma versão do sistema disponivel e entregue ao cliente

-Repository: é uma base de dados partilhada com versões de componentes de software e meta-informação

-System Building: a criação de um executavel de uma versão do sistema

-Workspace: área de trabalho privado onde o software pode ser modificado sem afetar outros trabalhadores

Gestão de Versões:

-Processo de manter o rasto das diferentes versões

-Pode ser visto como um processo usado para gerir codeline e baseline

-Deve garantir que alterações feitas não interfiram com alterações feitas por outros programadores

Codelines e Baselines

Codeline: sequencia de versões do código, onde as mais recentes são derivadas das mais antigas ; aplicam-se aos componentes do sistema

Baseline: definição de um sistema especifico ; especifica as versões dos componentes que são incluidos no sistema

Sistemas de gestão de versões(VCS)

-Identificação de versões e releases

-Gestão do historico de alterações

-Permitir desenvolvimento individual

-Permitir desenvolvimento de vários projetos ao mesmo tempo

Tipos de VCS:

-Centralizados: SVN

-Distribuidos: Git

Controle de versões centralizado:

-Programadores copiam componentes ou pastas(check-out)

-Após terminarem alterações: compiam componentes para o repositório(check-in)

-Vários programadores a trabalharem no  mesmo componente: cada pessoa copia o componente do repositorio

Controle de versões distribuido:

-Repositório “master”

-Em vez de apenas copiar coisas necessárias: Fazer clone do repositorio

-Repositorio privado(clonado do principal)

-Quando terminadas alterações: Commit ; update do repositorio privado ; Push das alterações

Construção do sistema – System Building

-Processo de criar uma versão completa e executavel do sistema

Ferramentas de construção do sistema e de gestão de versões: Fazer check-out de versões de componentes

-Descrição da baseline feita na ferramenta de construção do sistema

Plataformas de construção:

-Plataformas de Desenvolvimento

-Build Server

-Ambiente de reprodução

Build System:

-Funcionalidades: script para gerar build ; recompilação mínima, automação , …

System Build-Métodos Ágeis:

-Check out da mainline

-Construir o sistema e correr todos os testes de forma automática

-Fazer alterações aos componentes do sistema

-Construir o sistema no ambiente de desenvolvimento e correr todos os testes

-Depois de passar os testes, fazer check in de gestão de versões

-Construir o sistema no build server e correr testes

-Se os testes passarem no build server, fazer commit das alterações

Integração continua:

-Vantagens: Descoberta e correção de erros causados pela interação entre vários programadores ; Sistema +recente na mainline é a versão atual do sistema

-Desvantagens: Se o sistema for muito grande, a construção de testes pode levar muito tempo ; Se a plataforma de desenvolvimento for diferente da de execução, pode não ser possível executar os testes na área de trabalho do programador

**→Testes de Software**

-Testes de Desempenho:

-Podem fazer parte dos de release

-Refletem o perfil de utilização

-Variam a carga do sistema

-Testes de stress: tipo de desenvolvimento ; sistema é sobrecarregado

-TDD:

-Intercalar testes com desenvolvimento

-Testes escritos antes de implementar o código

-Código desenvolvido de forma incremental

-Base dos métodos ágeis

-Atividades:

-Indentificar incremento da funcionalidade a implementar

-Implementar teste para novo incremento

-Executar novo teste

-Implementar a funcionalidade e voltar a executar o teste

-Quando passar nos testes, escolher novo incremento para implementar

Beneficios:

-Todos segmentos de códigos associados a um testes, pelo menos

-Testes de regressão

-Debug simplificado

-Testes servem de documentação do sistema

-Testes de Releases:

-Testar uma release do sistema

-Objetivo:

-Mostrar que está pronto a ser usado

-Mostrar que implementa o que é especificado

-Testes black-box:

-Ignora-se implementação

-Testes criados a partir de especificações

-Verificar saidas de acordo com entradas

-Testes de utilizador:

-Utilizadores indicam como testar  o sistema

-Tipos: Testes Alfa ; Testes Beta ; Testes de aceitação

Testar programas:

-Serve para mostra descobrir problemas e erros ; Mostrar o que um programa faz ou deve fazer

-Como testar software: Executar programa com dados artificiais ; Verificar resultados e detetar erros ou anomalias

-Testes revelam erros, mas não a sua ausência

-Testes fazem parte do processo de verificação e validação

Objetivos:

-Demonstrar que o software está de acordo com os requisitos

-Encontrar comportamentos do software que não estão de acordo com os requisitos

Processo de testes – objetivo:

-Testes de validação: demostrar que o software está de acordo com os requisitos ; Bom teste mostra que o sistema funcionar como deve ser

-Testes de defeito: Encontrar erros e problemas ; Bom teste faz com que se exponham os erros e defeitos

Inspeções e Testes:

-Inspeções: análise e verificação por forma a encontrar problemas ; Verificação estática ; Técnica eficaz para achar erros ; Não necessitam de executar o sistema

-Vantagens: Os erros podem esconder outros erros ; Versões imcompletas inspecionadas sem custo adicional ; Pode ser util para encontrar atributos de qualidade

-Testes de software: Observação do comportamento do sistema ; Verificação Dinâmica

-Inspeções VS Testes:

-São atividades complementares ;

-Ambos usados em validação e verificação ;

-Inspeções podem verificar se sistema de acordo com especificações

-Inspeções não podem verificar caracteristicas não funcionais: Desempenho, usabilidade, …

Testes – Etapas:

-Durante fase de Desenvolvimento: Development testing ; Sistema testado durante desenvolvimento ; Procurar erros e defeitos

-Testes de release: testes a versão completa do sistema(release) ; equipa de testes diferente ; antes de entregar ao user

-Testes de utilizador: testes feitos por utilizadores reais

Testes Unitários:

-Testar componentes individuais de forma isolada

-Processo de testes de defeitos

-”Unidades” podem ser: funções ou métodos individuais ; objetos com vários atributos e métodos ; componentes com interfaces bem definidos

Testes objetos de classes:

-Cobertura completa dos testes envolve: testar todas as operações ; testar o objeto em todos os estados

-Herança pode tornar dificil o desenho dos testes

Testes autónomos:

-Automatizar sempre os testes unitários

-Frameworks de testes: Criar e executar testes / ex.: Junit ; Fornecem classes de testes genéricos ; Executar todos os testes implementados

Componentes:

-Setup: Inicialização do sistema(input , output esperado)

-Chamada/Execução: Método, função, ...

-Verificação/Assertion: comparação do output obtido com o output esperado

Estratégias de Testes:

-Particionar os Testes

-Baseado em regras/guias

Testes de componentes:

-Acede-se à funcionalidade do objeto através do seu interface

-Testar componentes compostos: focar em mostrar que interface tem comportamento de acordo com especificação ; assumir-se que testes unitários estão terminados

Testar interfaces:

Objetivos: Detetar falhas ; Erros de interface ; Suposições erradas

Erros de interface:

-Má utilização: chamar componente usando interface de forma errada

-Interface mal percebido: usar interface de forma errada, devido a más suposiçoes

-Problemas de sincronização: acesso a dados desatualizados

Testes de sistema:

-Teste de sistema durante desenvolvimento: integrar componentes ; criar versão do sistema ; testar o sistema integrado

-Foco: testar interações entre componentes

-Objetivo: testar se componentes são compativeis ; se dados corretos transferi-los através de interfaces ; testar comportamento global do sistema

Testes baseados em UseCases:

-Use Cases: interações do sistema ; podem ser usados como base dos testes do sistema

-Cada UseCase: envolve vários componentes ; testar UseCase força interação dos componentes

Testes de regressão:

-Testar sistema para verificar se alterações não produziram novos erros

-Considerando processo manual(dificil e despendioso)

-Considerando processo automático(simples, tds testes executados sempre que alteração no sistema)

-Todos os testes devem passar com sucesso antes do “commit”

**→Testes Unitários Junit**

Junit: Framework de testes ; Integração com IDEs ; Execução automática de testes ; Baseado em anotações

Conceitos:

Classes de testes:

-Uma classe de testes para cada classe

-Pelo menos um método de testes para cada método

Convenções:

-Usar os mesmos nomes dos package

-Nomes baseados no da classe a ser testada: Ex.: MyCalculator ; MyCalculatorTest

Métodos de Teste:

Estrutura:

-Setup do testes: init das vars ; def do resultado esperado

-Execução: método a ser testado

-Assert: Comparação esperado com o obtido

Conveções:

-Nomes descritivos

-Exemplo: deveDividirDoisNumeros()

deveLancarExcepcao()

Mecanismos Base:

Anotações: método para descobrir, orgnizar, ativar testes

Assertations(afirmações): método usado para ver se teste passou ou não ; comparação

Métodos de Teste:

-Executados em ambiente isolado: não devem depender de outros testes

-Baseados em anotações: @org.junit.jupiter.api.Test

Organização do Projeto:

-Pasta “src/main/java” : pasta com source do projeto; pasta com classes a serem testadas

-Pasta “src/test/java” : pasta com as classes de testes

Anotações Junit:

-@BeforeAll : Executado antes de tds os testes de uma classe de teste

-@AfterAll : Executado depois de tds os testes de uma classe de teste

-@BeforeEach : Antes de cada testes da classes de testes ; Setup de tds os testes ; Setup do ambiente de execução

-@AfterEach : Depois de cada teste ; Limpeza de tds os testes

-@Test : Usado para “marcar”/anotar um método de testes

-@Test(timeout = <delay>) : Definir tempo max de exec ; Se tempo exceder, o teste falha

-@Test(excepted = <exception>.class) : Teste passa se for lançada excepção

-@Ignore : Usado para ignorar testes

Assertions:

-assertEquals : Verifica se 2 objetos são iguais ; Opcional – Incluir mensagem de falha

-assertEquals for arrays : True – quando arrays tem o mesmo tamanho ; tds os elementos nas mesmas posições são iguais

-assertNull / assertNotNull : Verifica se um objeto é null / verifica se um objeto não é null

-assertSame / assertNotSame : Verifica se 2 objetos são o mesmo / verifica se 2 objetos não são o mesmo ; Equivalente – operador == && !=

-assertTrue / assertFalse : verifica se uma condição é verdade / verifica se uma condição é false

-fail : obriga teste a falhar ; util para forçar teste a falhar perante condições proibidas

Dicas:

-Usar vars de classes nas classes de teste: usar vars de classes como se fossem classes normais

-Ser exaustivo: testar máximo possível ; exemplo: testar todos os dias do ano

-Lidar com excepções: garantir lançamento de excepções quando devem ser lançadas

-Documentar os asserts: util quando testes falham

Verificar a cobertura de testes:

-Periodicamente

-Tipos de cobertura: Statement ; Branch ; “caminhos”

-Farrementas para verificar a cobertura de testes: EclEmma ; CodeCover ; NoUnit ; Jester ; …

Como correr testes:

-Diretamente no IDE: Eclipse – Run As Junit Test

-Usando o Maven: Numa consola executar – nvm test

**→Mocks**

-Testes Unitários:

-Testar unidades: Métodos ; funções ; ...

-Testar de forma isolada: não depender de outras unidades

-Problema: unidades dependes de outras ; dificil de isolar unidades

-Solução: simular unidades ; fazer “mocking” das unidades

-Mocking

-Criar objetos mocks: obj simulados

-Manualmente: através de código ; criação de classes “dummy” ; druante testes usadas classes “dummy”

-Forma automática: frameworks de mocking ; forma correta de criar obj mock

-Frameworks de Mocking:

-criar objetos falsos a partir de classes reais

-usados em testes unitários

-substituirem objetos reais

-”enganarem” um objeto

-”peça de teatro”

-Replicam o processo manual

-Alguns frameworks de Mocking para Java : Mockito ; PowerMock ; EasyMock

-Especificar comportamento dos objetos mock: especifico ; definido para cada teste

-Comportamentos típicos: Quando X incovado, retorna Y ; Quando X invocado com args Y e Z , retorna W

-Quando usar:

-Interação com métodos sem comportamento deterministico

-Interação com metódos com efeitos secundários

-Invocar operações externas

-Forçar erros “estranhos” e dificeis de reolver

-Uso Típico:

1)Fazer mock das dependências da classe a ser testada

2) Executar cód da classe a ser testada

3) Verificar se cód foi executado como esperado

-Mockito:

-Criar objetos mock: Anotação - @Mock

-Especificar valor de retorno:

-”when thenReturn”

-”when thenThrow”

-Invocações não especificadas devolvem valores nulos de acordo com o tipo:

-Null para objetos

- 0 para numeros

- falses para booleanos

-Especificar comportamentos:

-Valores de retorno

-Diferentes para cada método: Dependendo dos argumentos ; API “fluente”

-Verificar invocação de métodos:

-Behavior testing: verificar comportamentos ; não se verificam resultados

-Verificar de condições especificadas foram compridas: se um método foi invocado de acordo com parâmetros especificos

-Injeção de Mocks:

-Injeção de: construtores ; métodos ou atributos ; com base no tipo ...

-Testar métodos estáticos: mockito não permite ; necessário recorrer a outros frameworks ; powermock(disponibiliza classe para ser testada ; permite usar todas operações do mockito

**→Test Driven Development(TDD)**

-Metodologia de desenvolvimento baseada em testes: Escrita de código intercalada com testes

-Testes escritos antes da implementação

-Código escrito de forma incremental

-Metodologia introduziad como parte das metodologias ágeis: XP , Metodologias baseadas em planos

Processo TDD:

-Atividades: identificar incremento das funcionalidades necessário ; escrever testes para a funcionalidade ; executar o teste ; implementar a funcionalidade ; quando passarem(continuar para a próxima funcionalidade)

-Vantagens:

-Cobertura do código de testes: todos os fragmentos tem pelo menos 1 teste associado ; todo codigo tem pelo menos um teste

-Teste de regressão : conjunto de testes criado de forma incremental

-Debug mais simples: qd teste falha, mais facil detetar o local do problema ; descrevem o que o código deve fazer

-Documentação do sistema: testes são forma de documentação ; descrevem o que o código deve fazer

**→Desenho da Arquitetura**

-Perceber forma como sistema deve estar organizado: desenhar estrutura geral do sistema

-Etapa crítica; ligação entre desenho do sistema e requisitos ; indentifica compoenentes estruturais do sistema e relações entre si

-Output do desenho da arquitetura: modelo da arquitetura ; descreve organização do sistema através de conjunto de componentes

-Métodos ágeis e arquitetura do sistema:

-Fase Inicial: deve ser desenho geral da arquitetura – consenso comum

-Refactoring da arquitetura : afeta componentes ; processo dispendioso ; deve ser evitado

-Niveis de abstração:

-Pequena escala:  como um prog é decomposto em componentes

-Grande escala: arq de sis complexos empresariais ; sis que imcluem outros sis ; sis distribuidos

-Representação da arquitetura:

-Simples e informal: diagramas de blocos ; mostre entidades e relações

-Críticas: falta de semantica ; não mostram relações entre entidades nem props das entidades

-Tipos de Utilização:

-Forma de facilitar discussão sobre o desenho da arquitetura

-Forma de documentar a arquitetura

-Decisões de desenho:

-Processo criativo: Depende do sis a ser desenvolvido

-Existem decisões comuns: abragem tds tipos de desenhos ; afetam caracteristicas não funcionais

-Reutilização da arquitetura:

-Sis do mesmo domínio: tem arquiteturas semelhantes ; refletem mesmo conceitos

-Arquitetura de um sistema: pode ser desenhada usando vários padrões

-Visões da arquitetura:

-Que notações devem ser usadas?

-Cada modelo mostrar apenas 1 perspetiva do sistema

-Quais versões uteis estamos a desenhar e a documentar?

-Possiveis Visões:

-Lógica

-Processos

-Desenvolvimento

-Física

-Representação das perspetivas:

-UML

-ADL's(Architecure Description Languages)

-Padrões de arquiteturas:

-Forma de representar, partilhar e reutilizar conhecimento

-Padrões de arquiteturas

-Devem incluir info de quando são ou não uteis

-Podem ser representadas usando descrição tabular ou gráfica

-Arquitetura por Camadas:

-Usada para modelar interface de sub – sistemas

-Organiza sistema num conj de camadas

-Permite desenvolvimento incremental de subsistemas em diferentes camadas

-Arquitetura de repositório:

-Dados partilhados são guardadas num repositorio ou base de dados central e podem ser acedidos por tds subsistemas

-Cada subsistema mantem sua base de dados própria

-Quando existe grande quantidade de dados: modelo partilha usando um repositório ; mecanismo eficiente de partilha de dados

-Arquitetura Cliente Servidor:

-Modelo distribuido: mostra como dados e processos são distribuidos por vários componentes ; pode ser implementado apenas num único computador

-Conjunto de servidores standalone: fornecem serviços especificos, ex.: gestão de dados

-Conjunto de clientes que invocam esses serviços

-Rede que permite ligação entre os clientes e os servidores

-Arquitetura Pipe and Filter:

-Transformações funcionais: processam input ; produzem output

-Pipe and filter model: como conhecido na shell Linux/Unix

-Existem variantes deste modelo muito usadas: transformações sequenciais

-Não é aplicável a sistemas interativos