

Resumo FDS - Avaliação 2

Todos os títulos que estão em laranja, estão revisados. Isso pode significar que alguma informação foi alterada ou adicionada dentro do documento. (era madrugada 0 condições de fazer esse documento 100% bonitinho naquela hora)

Lembre-se que esse documento foi feito como forma individual de estudo, caso haja algum erro eu não sou responsável pelo seu erro se você escolher estudar apenas e exclusivamente por esse documento.

05 - Projeto de Software:

- Envolve a decomposição de problemas complexos em partes menores, facilitando sua resolução e implementação.

Project vs Design:

- Esclarece a diferença entre "project" e "design", destacando que no contexto do módulo, "projeto" se refere à elaboração de uma solução dividida em partes menores.

Módulos e Abstração:

- Explica que os módulos são unidades individuais de um software, como pacotes, componentes ou classes, que oferecem abstração ao permitir seu uso sem a necessidade de entender os detalhes internos de sua implementação.

Modelos de Software:

- Destaca o propósito dos modelos de software em preencher a lacuna entre requisitos e código, utilizando uma notação de abstração intermediária para conceber, especificar, entender e documentar uma solução.

UML (Unified Modeling Language):

→ Descreve a UML como uma linguagem gráfica proposta para modelagem de software, apresentando seus diagramas estáticos e dinâmicos como ferramentas para representar diferentes aspectos do sistema.

→ Diagramas UML:

- ◆ Detalha a finalidade dos diagramas de atividades, **que são usados para descrever a lógica de processos de negócios**, fluxos de trabalho ou processamentos, destacando suas características e uso.

Princípios de Projeto de Software:

→ Explica os princípios fundamentais do design de software, como **Integridade Conceitual, Ocultamento de Informação, Coesão e Acoplamento**, e sua importância na criação de sistemas robustos e flexíveis.

→ Princípios SOLID:

- ◆ Descreve cada um dos princípios SOLID (Single Responsibility Principle, Open Closed Principle, Liskov Substitution Principle, Interface Segregation Principle e Dependency Inversion Principle) e como eles orientam a estruturação e organização do código.
- ◆ **SRP (Princípio da Responsabilidade Única):**
 - Uma classe deve ter **apenas uma razão para mudar**, ou seja, deve ter uma única responsabilidade.
- ◆ **OCP (Princípio do Aberto/Fechado):**
 - As entidades de software **devem ser abertas para extensão**, mas **fechadas para modificação**. Isso promove um **design flexível**.
- ◆ **LSP (Princípio da Substituição de Liskov):**
 - Subtipos devem ser **substituíveis por seus tipos base** **sem afetar o comportamento do programa**.
- ◆ **ISP (Princípio da Segregação de Interface):**
 - As interfaces devem ser segregadas, de modo que **as classes implementem apenas os métodos necessários**. Isso evita interfaces monolíticas.
- ◆ **DIP (Princípio da Inversão de Dependência):**
 - **Módulos de alto nível não devem depender de módulos de baixo nível**; ambos devem depender de abstrações. Isso promove a flexibilidade e a extensibilidade do código.

→ **Princípio de Demeter:**

- ♦ Um objeto deve interagir apenas com seus “amigos” mais próximos, limitando seu conhecimento sobre outros objetos. Isso **reduz a dependência entre as partes do sistema**, tornando-o mais modular e fácil de manter.

06 - Arquitetura de Software

“Arquitetura de Software é sobre as coisas importantes, seja lá o que for isso.”
– Ralph Johnson

Arquitetura

- Projeto em mais alto nível, **focado em grandes unidades** como pacotes, módulos, subsistemas, camadas e serviços, em vez de unidades pequenas como classes.

Unidades de Maior Relevância

- **A definição de relevância depende do sistema**. Por exemplo, um mecanismo de persistência é crucial para sistemas de informações, mas pode não ser para sistemas de diagnóstico por IA.

Definição de Arquitetura de Software

- Segundo Neal Ford e Mark Richards, arquitetura de software é **a combinação da estrutura do sistema**, as **características arquiteturais** que o sistema deve prover, as **decisões arquiteturais** e os **princípios de projeto**.

Componentes da Arquitetura de Software

→ **Estrutura do Sistema:**

- ♦ A organização dos componentes do sistema.

→ Características Arquiteturais:

- ◆ As qualidades que o sistema deve ter, como desempenho e segurança.

→ Decisões Arquiteturais:

- ◆ As escolhas feitas durante o projeto do sistema.

→ Princípios de Projeto:

- ◆ Diretrizes que orientam o desenvolvimento do sistema.

Primeira Lei da Arquitetura de Software

- "Tudo em arquitetura de software é uma troca", afirmam Neal Ford e Mark Richards. Isso significa que **cada decisão arquitetural tem prós e contras que devem ser considerados.**

Importância da Arquitetura de Software

→ Debate Linus-Tanenbaum (1992):

- ◆ Linus Torvalds, criador do Linux, defendeu a arquitetura monolítica do Linux.
- ◆ Andrew Tanenbaum, autor do Minix, argumentou a favor de uma arquitetura microkernel, criticando a abordagem monolítica de Linus.
- ◆ Linus destacou que, apesar da teoria favorecer microkernels, a realidade prática do Linux monolítico já era uma solução funcional.

Padrões Arquiteturais

- Modelos pré-definidos que ajudam na organização da estrutura de um sistema. Exemplos incluem:

◆ Camadas:

- Divisão hierárquica do sistema.

◆ Model-View-Controller (MVC):

- Separação das responsabilidades de visualização, controle e modelo.

◆ Microsserviços:

- Pequenos serviços independentes.

◆ Orientada a Mensagens:

- Comunicação assíncrona via fila de mensagens.

◆ **Publish/Subscribe:**

- Sistemas que publicam e assinam eventos.

Categorias de Padrões Arquiteturais

→ **Padrões Arquiteturais:**

- ◆ Usados no projeto em alto nível.

→ **Padrões de Projeto:**

- ◆ Aplicados em médio e baixo nível, tanto em objetos quanto em frameworks.

→ **Idiomas:**

- ◆ Soluções de baixo nível orientadas à implementação, geralmente ligadas a uma linguagem de programação ou tecnologia específica.

Exemplos de Padrões Arquiteturais

→ **"From Mud to Structure":**

- ◆ Estruturas que ajudam a organizar um sistema confuso.

→ **"Layers (Camadas)":**

- ◆ Organização em camadas hierárquicas.

→ **"Pipes and Filters":**

- ◆ Processamento de dados em etapas conectadas por pipes.

→ **"Blackboard":**

- ◆ Solução para problemas complexos usando uma estrutura de colaboração.

→ **"Distributed Systems":**

- ◆ Estruturação de sistemas distribuídos.

◆ **"Broker":**

- Facilitação de comunicação em sistemas distribuídos.

→ **"Interactive Systems":**

- ◆ Implementação de sistemas interativos.

→ **"Model-View-Controller (MVC)":**

- ◆ Separação da lógica de apresentação e lógica de negócios.

→ **"Presentation-Abstraction-Control":**

- ◆ Arquitetura para sistemas interativos.

→ **"Adaptable Systems":**

- ◆ Sistemas que podem ser adaptados a diferentes contextos.

→ **"Microkernel":**

- ◆ Núcleo mínimo com funcionalidades essenciais.

→ **"Reflection":**

- ◆ Capacidade do sistema de inspecionar e modificar sua própria estrutura.

Arquitetura em Camadas

- Sistema organizado em camadas hierárquicas, onde cada camada só pode usar serviços da camada imediatamente abaixo. Essa estrutura facilita o entendimento e a troca de camadas, além de promover o reuso de camadas.

Arquitetura em Três Camadas

- Comum em processos de downsizing nas décadas de 80 e 90, onde sistemas corporativos migraram de mainframes para servidores Unix.

◆ **Camada 1:**

- Interface

◆ **Camada 2:**

- Lógica

◆ **Camada 3:**

- Servidor de Banco de Dados

Arquitetura Model-View-Controller (MVC)

- Surgiu com a linguagem Smalltalk nos anos 80 para implementar interfaces gráficas. Propõe dividir as classes em três grupos: Visão (GUI), Controle (eventos de entrada) e Modelo (dados). Adaptado para a web com frameworks como Ruby on Rails, Django e Spring, para criar sistemas distribuídos.

Arquiteturas Baseadas em Microserviços

- Diferem-se dos monólitos, onde o sistema é um único processo. Em microserviços, os módulos são processos independentes menores, permitindo **escalabilidade, flexibilidade** para releases e falhas parciais. Grandes empresas como Netflix, Amazon e Google utilizam microserviços.

Gerenciamento de Dados com Microserviços

- A arquitetura recomendada evita o acoplamento de dados entre módulos, permitindo que cada módulo **evolua independentemente**.

Arquitetura Orientada a Mensagens

- Utiliza uma fila de mensagens (broker) como intermediário entre clientes e servidores, promovendo **tolerância a falhas, escalabilidade e comunicação assíncrona**.

Arquitetura Publish/Subscribe

- Mensagens são tratadas como eventos, e os sistemas podem **publicar, assinar e serem notificados** sobre esses eventos, permitindo uma **comunicação em grupo eficiente**.

Outros Padrões Arquiteturais

→ "Pipes and Filters":

- ◆ Programas se comunicam por meio de buffers (pipes).

→ "Cliente/Servidor":

- ◆ Comum em serviços de redes.

→ "Peer-to-Peer":

- ◆ Todos nós podemos ser clientes e servidores simultaneamente, usado em compartilhamento de arquivos.

Anti-Padrões Arquiteturais

- Modelos que **não devem ser seguidos**, como "Big Ball of Mud", onde o sistema é uma "bagunça" sem uma estrutura clara, com módulos podendo usar qualquer outro módulo indiscriminadamente.

07 - Testes de Software

Verificação vs Validação

→ Verificação:

- ◆ Processo de confirmar se o sistema está sendo implementado corretamente, de acordo com os requisitos e especificações definidos.
Foco na conformidade com os critérios técnicos estabelecidos.

→ Validação:

- ◆ Processo de **assegurar** que o sistema correto está sendo implementado, aquele que **realmente atende às necessidades e expectativas dos clientes**. Foco na satisfação do usuário final.

Testes de Software

→ Objetivo:

- ◆ **Verificar se um programa gera os resultados esperados quando executado com casos de teste específicos.**

→ Tipos de Testes:

- ◆ **Manuais:**
 - Realizados por um testador humano.
- ◆ **Automatizados:**
 - Executados por scripts de teste, nosso foco principal.

→ Limitação Principal:

- ◆ Testes de software podem revelar a presença de bugs, mas **não garantem a ausência de todos os erros**. (Edsger W. Dijkstra)

Defeitos, Bugs, Falhas e Erros de Contexto

→ Defeito (Defect):

- ◆ Erro no código que pode causar **comportamento incorreto** e impede a funcionalidade plena desse software.
- ◆ **Exemplo:**
 - $\text{área} = \pi * \text{raio} * \text{raio} * \text{raio}$; ao invés de $\text{area} = \pi * \text{raio} * \text{raio}$;

→ Falha (Failure):

- ◆ **Resultado errado que ocorre quando um defeito é executado.**
- ◆ Ocorre quando o software não executa uma função conforme o esperado
- ◆ Está diretamente ligada ao defeito.

→ Bug:

- ◆ Termo informal para um defeito ou um erro no software.
- ◆ Representa problemas no código que causam comportamento não esperado e incorreto.

→ Erro de contexto:

- ◆ Ocorre quando o software não entende ou processa corretamente as entradas ou situações específicas.
- ◆ Resulta de suposições incorretas ou mal interpretadas sobre a funcionalidade e aplicação do código, ou sobre o ambiente e os dados com os quais o software interage.

Testes com Métodos Ágeis

→ Automatizados:

- ◆ Preferência por testes automatizados para **eficiência**.

→ Implementação:

- ◆ Algumas vezes, testes são escritos antes do código (TDD - Test Driven Development).

→ Responsabilidade:

- ◆ Desenvolvedores escrevem testes para o código que desenvolvem.

→ Funções Adicionais:

- ◆ Detectar regressões e servir como documentação.

Tipos de Teste

→ Teste de Unidade:

- ◆ Testam pequenas unidades de código (normalmente classes ou métodos).

→ Teste de Integração:

- ◆ Testam a **interação** entre diferentes partes do sistema.

→ Teste de Sistema:

- ◆ Testam o sistema **como um todo**, incluindo interações externas.

→ Pirâmide de Testes:

- ◆ Estrutura que organiza os testes em camadas (unidade na base, integração no meio, e sistema no topo).

Processo de Teste de Software

→ Planejamento do Teste:

- ◆ Definir o que será testado e como.

→ Monitoramento e Controle do Teste:

- ◆ **Acompanhar** a execução dos testes e **garantir** que estão conforme o planejado.

→ Análise do Teste:

- ◆ **Examinar os resultados** dos testes para **identificar defeitos**.

→ Modelagem do Teste:

- ◆ Projetar e **priorizar casos de teste**, identificar dados e ambientes necessários.

→ Execução do Teste:

- ◆ Realizar os testes planejados.

Técnicas de Modelagem de Teste

→ Caixa-preta:

- ◆ **Definição:**
 - Técnicas de teste de caixa-preta focam nas funcionalidades do software, sem considerar a estrutura interna do código. A ideia é **testar as entradas e saídas do sistema de acordo com os requisitos especificados**.
- ◆ **Particionamento de Equivalência:**
 - Dividir dados de entrada em grupos que devem ser tratados da mesma forma.
- ◆ **Análise do Valor Limite:**
 - Testar limites extremos de entrada.

- ◆ **Pairwise Testing:**

- Testar todas as combinações pares de variáveis.

- ◆ **Personas:**

- Testar com base em perfis de usuário.

→ **Caixa-branca:**

- ◆ **Definição:**

- Técnicas de teste de caixa-branca analisam a estrutura interna do código, verificando o funcionamento detalhado do sistema. Isso envolve examinar a lógica, os fluxos de controle e a cobertura do código.

- ◆ **Cobertura de Instruções:**

- Testar cada linha de código.

- ◆ **Cobertura de Decisão:**

- Testar cada decisão (if, else).

→ **Resumo:**

- ◆ As técnicas de teste de caixa-preta e caixa-branca são complementares.
- ◆ Enquanto a caixa-preta foca em verificar se o sistema atende aos requisitos funcionais sem considerar como ele foi implementado, a caixa-branca se concentra em garantir que a implementação interna do sistema esteja correta.
- ◆ Uma abordagem eficaz de testes de software geralmente envolve a combinação de ambas as técnicas para garantir uma cobertura abrangente e uma alta qualidade do software.

→ **Baseadas na Experiência:**

- ◆ **Teste Exploratório:**

- Testar sem casos de teste pré-definidos.

- ◆ **Heurísticas:**

- Usar regras gerais e experiência para guiar os testes.

Testes de Sistema

→ **Testes ponta-a-ponta (end-to-end):**

- ◆ Testar o sistema inteiro através de sua interface externa.

→ **Exemplo:**

- ◆ Usar Selenium para testar um sistema web, automatizando interações como preenchimento de formulários e cliques em botões.

BDD – Behavior Driven Design

→ **Extensão do TDD:**

- ◆ Escrever testes antes do código, mas focando em comportamento.

→ **DSL Simples:**

- ◆ Converter linguagem natural estruturada em testes executáveis.

→ **Critérios de Aceitação:**

- ◆ Fornecer critérios claros para aceitação de funcionalidades.

Testes de Unidade

→ **Objetivo:**

- ◆ Testar pequenas unidades de código de forma isolada.

- ◆ **Exemplo:**

- Testar uma classe Stack.

→ **Framework de Testes:**

- ◆ Usar xUnit para organizar e executar testes.

→ **Princípios FIRST:**

- ◆ São diretrizes fundamentais que orientam a criação e manutenção de testes de unidades de alta qualidade.

- ◆ **Princípios:**

- **Rápidos (Fast):**

- Testes de unidade devem ser rápidos de executar.

- **Independentes (Independent):**

- Testes de unidade devem ser independentes uns dos outros.

- **Determinísticos (Repeatable):**

- Testes de unidade devem ser repetíveis, produzindo os mesmos resultados em qualquer ambiente.

- **Auto-verificáveis (Self-checking):**
 - Testes de unidade devem ser auto-verificáveis, determinando automaticamente se passaram ou falharam.
- **Oportunos (Timely):**
 - Testes de unidade devem ser escritos de forma oportuna, preferencialmente antes do código que testam.

Mocks

→ Função:

- ◆ Emular objetos reais de forma simplificada para testes rápidos e unitários.

→ Exemplo:

- ◆ Usar Mockito para criar mocks e programar seu comportamento.

Desenvolvimento Dirigido por Testes (TDD)

→ Prática:

- ◆ Escrever testes antes do código.

→ Benefícios:

- ◆ Assegura que os testes sejam escritos.
- ◆ Incentiva código testável.
- ◆ Melhora o design e a usabilidade do código.

→ Ciclo TDD:

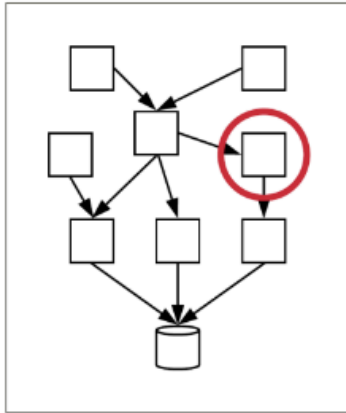
- ◆ Vermelho (escrever teste), Verde (codar para fazer teste passar), Refatorar (melhorar código).

Testes de Integração

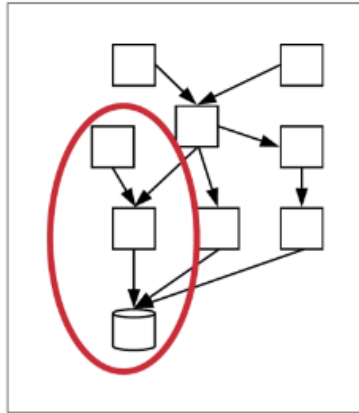
→ **Objetivo:**

- ◆ Testar a interação entre diferentes partes do sistema, incluindo dependências externas como bancos de dados.

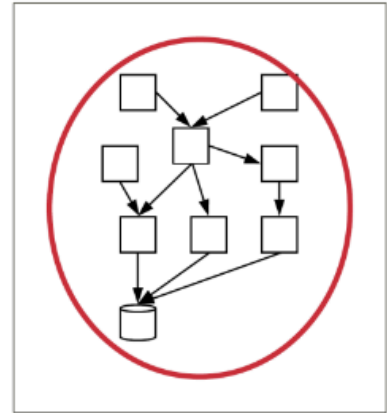
→ **Relembrando:**



Unidade



Integração



Sistema

Cobertura de Testes

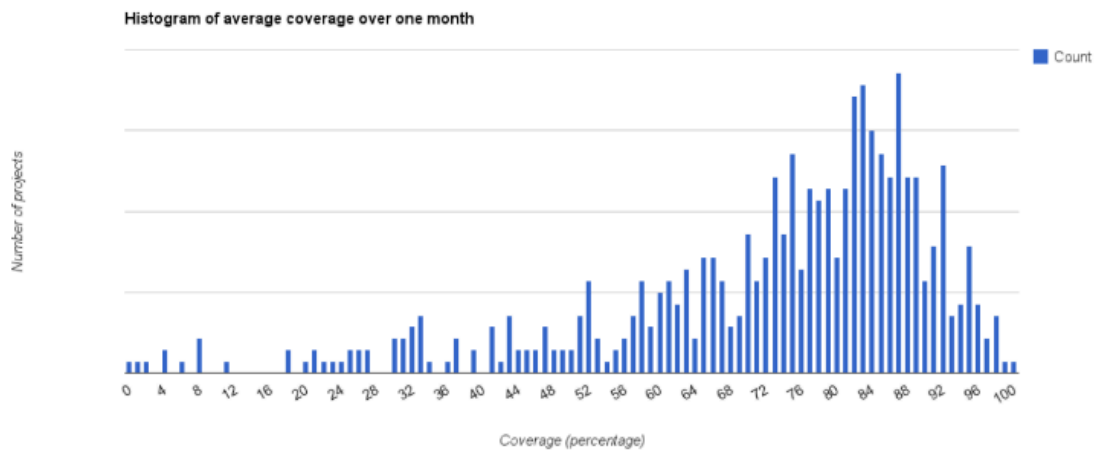
→ **Definição:**

- ◆ Cobertura de testes = (número de comandos executados pelos testes) / (total de comandos do programa).

→ **Cobertura Ideal:**

- ◆ Varia, mas geralmente é recomendada entre 60% e 90%.

→ Exemplo Google:



The median is 78%. the 75th percentile 85% and 90th percentile 90%.

Outros Tipos de Testes

- Aceitação (manual)
- Alfa e beta (manual)
- Requisitos não-funcionais

Software Defect Reports

→ Propósito:

- ◆ Corrigir erros identificados pelos testes.

→ Ciclo de Vida do Defeito:

- ◆ Analisar -> Reportar -> Acompanhar -> Retesta -> Fecha

Analisar um Defeito

- Identificar a causa raiz, verificar se é possível reproduzir, isolar o defeito e encontrar caminhos alternativos.

Reportar um Defeito

- Garantir que não é duplicata, comunicar com desenvolvedor, entrar informações no sistema e garantir correção.

Acompanhar um Defeito

- Implementar processo de acompanhamento, considerar Defect Review Board com lead tester, desenvolvedores e gerência.

Retestar e Fechar um Defeito

- Possíveis saídas: Problema corrigido, contínua ou substituído por novo. Se corrigido, fechar defeito.

08 - DevOps

- Antigamente, a implantação de software era sempre traumática devido a uma divisão clara entre as equipes de desenvolvimento e operações, com pouca comunicação entre elas. As operações (Ops) envolviam administradores de sistema, suporte, sysadmin, pessoal de IT, etc.

Ideia Central do DevOps:

- DevOps visa aproximar as equipes de Desenvolvimento (Dev) e Operações (Ops) para trabalharem juntas desde o início do projeto. Isso promove uma colaboração contínua, reduzindo conflitos e facilitando a passagem de bastão entre as equipes.

Objetivo Principal:

- O objetivo principal do DevOps é acabar com o jogo-de-empurra onde desenvolvedores culpam os administradores de sistema pelo mal funcionamento dos servidores e vice-versa.

DevOps:

- Não é um cargo específico, mas sim um conjunto de princípios e práticas que constituem uma cultura ou movimento.
- O termo surgiu por volta de 2009 e foca na colaboração e automação para entregar software de maneira eficiente e confiável.

Princípios de DevOps:

1. Aproximar Devs e Ops desde o início do projeto.

- Tradicionalmente, as equipes de desenvolvimento e operações trabalhavam de forma isolada, o que muitas vezes resultava em conflitos e atrasos. O DevOps propõe que essas equipes trabalhem juntas desde o começo do projeto para garantir que todos estejam alinhados em termos de requisitos, expectativas e objetivos. Isso melhora a comunicação, a colaboração e ajuda a identificar e resolver problemas mais cedo no ciclo de vida do desenvolvimento.

2. Adotar princípios ágeis durante a fase de implantação.

- Os princípios ágeis, como entregas frequentes e incrementais, feedback contínuo e adaptação a mudanças, não devem se limitar apenas à fase de desenvolvimento, mas também à implantação. Isso significa que a fase de implantação deve ser iterativa, incremental e flexível, permitindo ajustes rápidos e respondendo às necessidades do negócio de maneira ágil.

3. Transformar implantações em um evento rotineiro e sem problemas.

- Em muitas organizações, a implantação de software é vista como um evento arriscado e estressante. O DevOps busca mudar essa percepção transformando as implantações em um processo rotineiro e previsível. Isso é alcançado através da automação, testes rigorosos e práticas de integração contínua, que garantem que cada mudança seja implementada de forma suave e sem interrupções significativas.

4. Realizar **implantações diárias de sistemas (ou partes deles)**.

- Em vez de esperar meses para uma grande liberação, o DevOps incentiva implantação frequentes, até diárias, de partes do sistema. Isso permite que novas funcionalidades, correções de bugs e melhorias cheguem aos usuários de forma rápida. Implantar pequenas mudanças frequentemente reduz o risco associado a grandes lançamentos e facilita a identificação e resolução de problemas.

5. **Automatizar o processo de implantação para eficiência e consistência.**

- A automação é um componente central do DevOps. Automatizar o processo de implantação ajuda a garantir que ele seja realizado de maneira consistente e eficiente, minimizando erros humanos. Ferramentas de automação podem gerenciar tarefas como configuração de ambientes, execução de testes, implantação de código e monitoramento de desempenho, permitindo que a equipe se concentre em tarefas mais estratégicas.

Práticas de DevOps:

→ **Controle de Versões (GIT):**

- ◆ Essencial para o desenvolvimento colaborativo, permitindo armazenar e recuperar diferentes versões do software.
- ◆ Existem sistemas de controle de versão centralizados (como SVN, CVS) e distribuídos (como Git).

◆ **Vantagens dos Sistemas de Controle de Versão Distribuídos (DVCS):**

- Comitar alterações com mais frequência e rapidez.
- Trabalhar offline.
- Suporte a arquiteturas alternativas como P2P e hierárquicas.

◆ **Multirepos vs Monorepos:**

- Multirepos: Diversos repositórios separados para diferentes projetos (comum).
- Monorepos: Um único repositório que contém múltiplos projetos, favorecendo grandes empresas.

◆ Vantagens dos Monorepos:

- Uma única fonte de "verdade".
- Incentivo ao reúso de código.
- Mudanças atômicas e facilidade em refatorações de larga escala.

◆ Desvantagens dos Monorepos:

- Podem exigir ferramentas customizadas.

→ Integração Contínua (CI):

- ◆ Prática que envolve integrar o código frequentemente, recomendando-se pelo menos uma vez por dia.
- ◆ Boas práticas dentro da Integração Contínua são: Builds automatizados, Testes automatizados e pair programming.

◆ Práticas:

- Frequência de Integração:

- Integração do código várias vezes ao dia.

- Automação de Build e Testes:

- Compilação e execução de testes automatizados após cada integração.

- Retroalimentação Rápida:

- Feedback imediato sobre a qualidade do código.

- Garantia de Qualidade e Estabilidade:

- Evita acumulação de problemas de integração e identifica bugs precocemente.

- Automação de Implantação:

- Possibilita a entrega contínua de novas funcionalidades aos usuários.

◆ Padrões e Estratégias de Branching:

- Mainline:

- Um único branch que representa o estado atual do produto.

- **Healthy Branch:**
 - Realização de checagens automáticas a cada commit/push para garantir a integridade do branch.
- **Mainline Integration:**
 - Desenvolvedores integram seu trabalho a partir do mainline, realizando merge conforme necessário.

◆ **Feature Branching:**

- Branches criados para desenvolvimento de novas funcionalidades, que são integrados à mainline após a conclusão.

◆ **Caminho para Produção:**

- **Release Branch:**
 - Para estabilizar versões prontas para lançamento.
- **Maturity Branch:**
 - Marca versões do código com um determinado nível de qualidade.
- **Environment Branch:**
 - Configurações específicas para novos ambientes.
- **Hotfix Branch:**
 - Captura correções urgentes de bugs em produção.
- **Release Train:**
 - Releases regulares em intervalos definidos.
- **Release-Ready Mainline:**
 - Possibilita releases diretos da mainline.

◆ **Estratégias de Branch:**

- **Git-flow:**
 - Utiliza branches permanentes (Main e Develop) e branches temporários de apoio (Feature, Release, Hotfix).
- **GitHubFlow:**
 - Simplificação do Git-flow, sem branches develop, release e hotfix, utilizando suporte a Pull Requests.
- **Trunk-based Development (TBD):**
 - Desenvolve-se diretamente no branch principal (trunk) para evitar conflitos de merge.

→ Deployment Contínuo (CD):

- ◆ Extensão da integração contínua onde commits/merges frequentes são imediatamente implantados em produção, facilitando experimentação e feedback rápido.
- ◆ **História da Engenharia de Software: Encurtamento de Ciclos**
 - **Waterfall:**
 - Ciclos de meses ou anos para desenvolvimento e entrada em produção.
 - **Ágil:**
 - Ciclos de semanas para desenvolvimento e meses para produção.
 - **Ágil + CD:**
 - Ciclos de semanas para desenvolvimento com implantações imediatas.
- ◆ **Feature Flags:**
 - Mecanismos que permitem ativar ou desativar funcionalidades específicas de um software sem precisar alterar o código-fonte, facilitando testes, implementações graduais e a gestão de lançamentos.

Deployment Contínuo

→ Automação do Processo de Lançamento

- ◆ Implementa mudanças na produção automaticamente.
- Condições para Lançamento
 - ◆ O lançamento ocorre assim que os testes forem aprovados.

Deployment Delivery

- Prontidão para Lançamento Manual
 - ◆ Foco em garantir que o software esteja sempre pronto para ser lançado;
- Método de Lançamento
 - ◆ O lançamento é realizado de forma manual.

Resumo da Palestra de Microserviços:

Aqui está um resumo da palestra "Microservices" de Martin Fowler na GOTO 2014 em tópicos:

→ Definição de Microservices:

- ◆ Arquitetura composta por serviços pequenos e independentes.
- ◆ Cada serviço executa um processo único e se comunica via APIs.

→ Comparação com Arquiteturas Monolíticas:

- ◆ Monólitos são sistemas únicos e integrados.
- ◆ Microservices oferecem melhor escalabilidade e manutenção.

→ Benefícios dos Microservices:

- ◆ Escalabilidade: Serviços podem ser escalados individualmente.
- ◆ Flexibilidade de desenvolvimento: Equipes independentes podem trabalhar em serviços diferentes.

→ Desafios dos Microservices:

- ◆ Comunicação entre serviços: Necessidade de gerenciar a comunicação e a latência.
- ◆ Gerenciamento de dados: Distribuição de dados entre diferentes serviços.

→ Melhores Práticas:

- ◆ Implementação de DevOps: Integração contínua e entrega contínua (CI/CD).
- ◆ Monitoramento e Log: Necessidade de monitorar e registrar serviços individuais.

→ Exemplos e Casos de Uso:

- ◆ Discussão de empresas que adotaram microservices com sucesso.

→ Conclusão:

- ◆ Microservices são uma solução poderosa para sistemas complexos, mas requerem uma abordagem disciplinada para gestão e implementação.

Para mais detalhes, assista à palestra [aqui](#).

Questões:

Respondidas baseadas no que EU to achando certo, se tu acha que eu to errada comenta.

1.

Qual é a diferença entre verificação e validação dos testes de software?

R.

A verificação é o processo de assegurar que os critérios técnicos estejam sendo cumpridos de forma correta, já a validação é baseada nos parâmetros do cliente, assegurando que ele as suas necessidades e expectativas estejam sendo adequadamente cumpridas.

2.

Quais são os princípios do DevOps e como eles influenciam a colaboração entre as equipes de desenvolvimento e operações?

R.

1- Aproximação entre desenvolvedores e operadores 2- Aplicação de metodologias ágeis, 3- implantação ser um evento rotineiro e constante no sistema, 4- realizar implantação diárias, 5- automatizar o processo de implantação.

A partir desses 5 princípios a equipe ao estar mais integrada e em maior coordenação possui uma grande chance de aumentar a qualidade de produção da aplicação além de agilizar a sua produção.

3.

Explique a importância do controle de versões no desenvolvimento de software, destacando as vantagens dos sistemas de controle de versão distribuídos.

R.

O controle de versões é útil para que haja um ponto de encontro entre todos os desenvolvedores dentro de um único local. A utilização do sistema de controle distribuído se destaca principalmente por causa do seu fluxo de trabalho, que ao

ser utilizado aumenta significativamente a eficiência e qualidade de entrega dentro de um projeto, onde ele permite o aumento da frequência de commits, além de possibilitar o trabalho off-line para todos os desenvolvedores responsáveis pelo projeto.

4.

Quais são as práticas comuns de integração contínua e como elas contribuem para a eficiência do desenvolvimento de software?

R.

1- Frequência de integração, 2- Testes automatizados, 3- Feedback Rápido/ Retroalimentação Rápida, 4- Garantia de qualidade e estabilidade, 5- Automação da implementação. A partir do momento em que as práticas de CI são aplicadas dentro de um projeto, sua eficiência aumenta de forma exponencial, uma vez que os processos, que normalmente são mais inclinados a resultar em problemas e complicações, são limpos e tratados de forma adequada, não sendo mais obstáculos para o projeto em desenvolvimento.

5.

Descreva a diferença entre o modelo de desenvolvimento Git-flow e o GitHubFlow.

R.

O Git-Flow é um modelo de desenvolvimento com melhor adequação para projetos mais complexos, sendo eles maiores e mais complexos, já que ele disponibiliza diferentes tipos de branches diferentes para múltiplos ambientes de produção, testes e etc. Em contrapartida, o GitHubFlow é um modelo de desenvolvimento mais simples preferível para projetos mais dinâmicos e ágeis, onde as alterações são integradas diretamente da branch de principal e implantadas na branch de produção após a testagem e validação, sem a necessidade de ambientes intermediários.

6.

Como as estratégias de branching, como o Git-flow, influenciam o gerenciamento do ciclo de vida do software?

R.

O gerenciamento é influenciado a partir do momento em que o Git-Flow organiza as estratégias de desenvolvimento do projeto para uma organização mais concisa e com previsibilidade de erros maior do que a de um projeto solto. Tais organizações permitem um controle mais eficiente sobre as mudanças,

desafios e imprevisibilidades do projeto, facilitando também a comunicação mais clara e eficiente entre a equipe, além de promoverem uma maior organização do fluxo de trabalho para o projeto. Todos esses fatores influenciam diretamente na eficiência de um projeto, aumentando-a.

7.

Qual é o objetivo do deployment contínuo e como ele se relaciona com a prática de integração contínua?

R. É automatizar o processo de entrega dos trechos de código, testados e validados, de uma aplicação, por meio de metodologias como o build de testes automatizados e a integração automática na linha de produção. Por meio da união da integração contínua e o deployment contínuo é possível uma redução de riscos e fornecer feedbacks rápidos e eficientes sobre a qualidade e desempenho do software.

8.

Quais são os benefícios das feature flags no desenvolvimento de software e como elas podem ser utilizadas?

R.

As feature flags são ferramentas que permitem ativar ou desativar funcionalidades específicas de um software sem realizar o redeploy, oferecendo benefícios como deploys mais seguros, facilidade no rollback de funcionalidades problemáticas e suporte a A/B testing e lançamentos graduais. Elas podem ser utilizadas para criar uma experiência personalizada para o usuário baseada nas suas preferências, pois apenas algumas flags são, ou não, ativadas. Ademais, as flags podem ser utilizadas como uma ferramenta de controle de acesso, gerenciamento centralizado, monitoramento e análise, entre outros.

9.

Explique a diferença entre defeito, bug e falha no contexto de testes de software.

R.

Defeito é um trecho de código que não foi projetado corretamente dentro de uma aplicação. O bug é a consequência desse erro dentro do código. A falha de contexto é a elaboração de uma feature, por exemplo, de forma possivelmente

correta, tecnicamente falando, mas que não foi adequadamente interpretada pelo desenvolvedor.

10.

Como o Test Driven Development (TDD) influencia o ciclo de desenvolvimento de software e a qualidade do código produzido?

R.

O TDD auxilia na produção pois assegura que os testes sejam escritos, estimula a criação de códigos que sejam testáveis além de aumentar a qualidade do design e usabilidade do código.

11.

Quais são os princípios SOLID e como cada um deles contribui para o design de software robusto?

R.

12. O que é a arquitetura em camadas e quais são seus benefícios e limitações?

13. Explique a arquitetura Model-View-Controller (MVC) e como ela se aplica ao desenvolvimento web.

14. Qual é a diferença entre microserviços e uma arquitetura monolítica? Quais são as vantagens e desvantagens de cada abordagem?

15. O que são padrões arquiteturais e como eles ajudam na organização de um sistema de software?

16. Descreva o princípio de Demeter e como ele impacta a modularidade e a manutenção de um sistema.

17. Quais são os diferentes tipos de diagramas UML e como eles são utilizados no desenvolvimento de software?

18. O que é a arquitetura orientada a mensagens e quais são suas principais características?

19. Como a arquitetura publish/subscribe funciona e em quais cenários ela é mais eficaz?

20. Explique o conceito de “primeira lei da arquitetura de software” segundo Neal Ford e Mark Richards.

21. Qual é a importância da arquitetura de software e como ela influencia o sucesso de um projeto?

22. O que são anti-padrões arquiteturais e como eles podem afetar negativamente um sistema?

23. Descreva a técnica de teste de caixa-preta e como ela difere da técnica de teste de caixa-branca.
24. Quais são as etapas do processo de teste de software e o que envolve cada uma delas?
25. Como a pirâmide de testes organiza os diferentes tipos de testes e quais são suas camadas principais?
26. Explique o conceito de testes exploratórios e como eles podem ser úteis em um projeto de software.
27. Quais são os princípios FIRST para testes de unidade e como eles ajudam a garantir a qualidade dos testes?
28. O que são mocks em testes de software e como eles são utilizados?
29. Descreva o ciclo de vida de um defeito desde a detecção até a correção.
30. Quais são os diferentes tipos de testes de sistema e como eles se diferenciam dos testes de unidade e de integração?
31. Como a abordagem de Test Driven Development (TDD) impacta a qualidade e a manutenibilidade do código?
32. O que são critérios de aceitação em BDD e como eles são utilizados para validar funcionalidades?
33. Explique o conceito de cobertura de testes e qual é a cobertura ideal para garantir a qualidade do software.
34. O que é o desenvolvimento baseado em trunk (trunk-based development) e quais são seus benefícios?
35. Descreva a importância da automação no processo de DevOps e como ela contribui para a eficiência das operações.
36. Quais são as diferenças entre um monorepo e multirepos e quais são as vantagens e desvantagens de cada abordagem?
37. Como os padrões de branching como Mainline Integration e Feature Branching ajudam no gerenciamento do desenvolvimento de software?
38. O que é um release branch e como ele é utilizado no processo de desenvolvimento e implantação?
39. Explique como a arquitetura microkernel difere da arquitetura monolítica e quais são suas vantagens.
40. Quais são os benefícios da arquitetura distribuída e em quais cenários ela é mais adequada?
41. O que é o princípio de Liskov Substitution e como ele se aplica ao design de software?
42. Como a arquitetura de sistemas interativos (Interactive Systems) melhora a experiência do usuário?
43. O que é o modelo “pipes and filters” e como ele organiza o processamento de dados?
44. Quais são as características de uma boa arquitetura de software segundo Ralph Johnson?

45. Explique a diferença entre testes de aceitação alfa e beta e como eles contribuem para a qualidade do software.
46. Como a técnica de particionamento de equivalência é utilizada em testes de software?
47. O que é análise de valor limite em testes de software e como ela ajuda a identificar erros?
48. Descreva o processo de reportar um defeito e quais informações são essenciais para a correção eficaz.
49. Quais são as etapas de acompanhamento de um defeito e por que elas são importantes?
50. Como a automação de testes pode ajudar a detectar regressões e melhorar a documentação do software?