Resumo FDS - Avaliação 1

Todas as partes que estão em laranja, estão revisadas. 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)

01 -Introdução a Engenharia de Software:

Definição de Engenharia de Software:

- ➔ Investigação e proposta de soluções para desenvolver sistemas de software complexos e de grande porte de forma produtiva e com qualidade.
- ➔ Surgiu como resposta às demandas desafiadoras enfrentadas na produção desses sistemas.

Início da Engenharia de Software:

- ➔ O termo foi usado pela primeira vez na Conferência da OTAN em 1968, reconhecendo as dificuldades únicas enfrentadas na produção de sistemas complexos. Isso marcou o reconhecimento da necessidade de **uma abordagem disciplinada e sistemática para desenvolver software**.

Desafios Essenciais:

- ➔ A complexidade, conformidade, facilidade de mudanças e invisibilidade distinguem a engenharia de software de outras engenharias, apresentando desafios específicos que exigem abordagens adaptadas.

Áreas de Estudo em Engenharia de Software:

1. **Engenharia de Requisitos:**
    - Compreende atividades como especificação, análise, documentação e validação dos requisitos do sistema, garantindo que atendam às necessidades dos clientes.
2. **Projeto de Software:**
    - Define os principais componentes e interfaces do sistema, estabelecendo a arquitetura e estrutura geral.
3. **Construção de Software:**
    - Envolve a implementação do sistema, considerando algoritmos, estruturas de dados, ferramentas e padrões de codificação.
4. **Testes de Software:**
    - Verifica se o programa produz os resultados esperados, incluindo testes manuais e automatizados para garantir a qualidade.
5. **Manutenção de Software:**
    - Inclui atividades corretivas, preventivas, adaptativas e evolutivas, garantindo a eficácia contínua do software ao longo do tempo.
6. **Gerência de Configuração:**
    - Controla as versões do código-fonte e facilita a recuperação de versões antigas, garantindo integridade e rastreabilidade.
7. **Gerência de Projetos:**
    - Envolve negociação de contratos, gerenciamento de recursos humanos, riscos e outras atividades para garantir o sucesso do projeto.
8. **Processos de Desenvolvimento de Software:**
    - Define as atividades para a construção do sistema, incluindo modelos como Waterfall, Espiral e Ágil, cada um com abordagens diferentes.
9. **Modelagem de Software:**
    - Usa representações de alto nível para facilitar a compreensão, documentação e manutenção do sistema.
10. **Qualidade de Software:**
    - Considera atributos internos e externos que determinam a qualidade do software, como correção, robustez e modularidade.
11. **Prática Profissional:**
    - Engloba aspectos éticos, certificações, regulamentação e educação na área de engenharia de software.
12. **Aspectos Econômicos:**
    - Incluem retorno de investimento, monetização do software e escolha adequada de licenças, considerando questões financeiras e comerciais.

Tipos ABC de Sistemas:

- ➔ **Sistemas C (Casuais):** Pequenos e não críticos, desenvolvidos por um ou dois engenheiros, muitas vezes descartáveis e sem grande importância estratégica.
- ➔ **Sistemas B (Business):** Importantes para uma organização, beneficiam-se das práticas de engenharia de software e podem se tornar ativos valiosos quando bem desenvolvidos.
- ➔ **Sistemas A (Acute):** Críticos, com alto risco associado a falhas, podendo ter impacto significativo em termos de vidas humanas ou financeiras. Geralmente requerem certificações e especificações formais, estando fora do escopo de um curso padrão de engenharia de software.

02- Processos

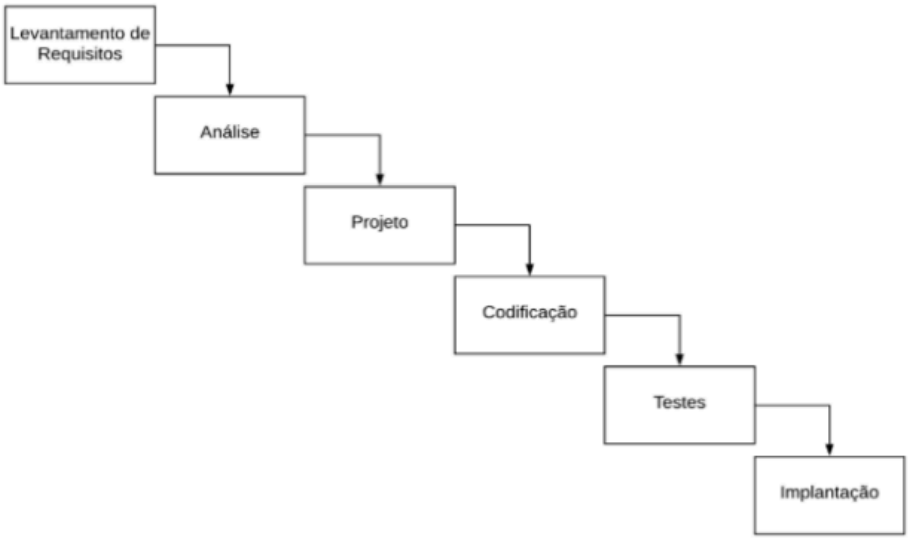
Importância dos Processos:

- ➔ Os processos são essenciais para fornecer uma estrutura clara para os desenvolvedores entenderem suas responsabilidades e os resultados esperados de seu trabalho. Eles servem como um guia para coordenar, motivar, organizar e avaliar o progresso e a qualidade do trabalho realizado.

Engenharia Tradicional:

- ➔ A engenharia tradicional, aplicada em áreas como construção civil, mecânica, elétrica, aviação, etc, é caracterizada por um planejamento detalhado e uma abordagem sequencial para o desenvolvimento de produtos e projetos.

Metodologia Waterfall:



Desafios do Waterfall:

➔ O modelo Waterfall, embora tenha sido amplamente utilizado na engenharia tradicional, **não se adaptou bem** ao desenvolvimento de software. Isso ocorre porque o software é uma entidade mais abstrata e mutável, com requisitos difíceis de prever e propensos a mudanças frequentes.

Fases do RUP (Rational Unified Process):

➔ **Inception:**
- análise de viabilidade

➔ **Elaboração:**
- requisitos + arquitetura

➔ **Construção:**
- projeto de baixo nível + implementação

➔ **Transição:**
- implantação (deployment)

➔ Cada fase era composta por várias iterações, cada iteração enfatizava algumas disciplinas e costumavam ter no mínimo 3 meses.

Desafios do RUP:

➔ Documentações detalhadas;

➔ Verbosas e pouco úteis;

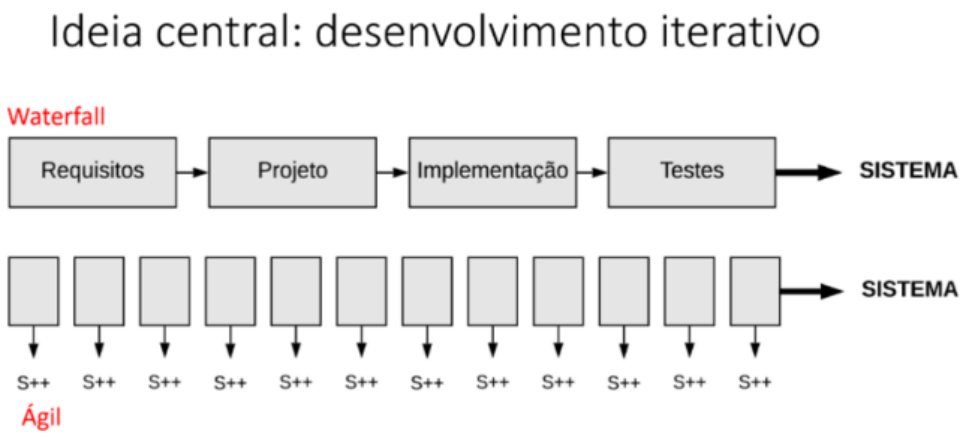
➔ Na prática, desconsiderada durante a implementação. Não funcionou com o software.

Transição para Métodos Ágeis:

➔ Devido à necessidade de uma abordagem mais flexível e responsiva às mudanças, houve uma transição gradual de métodos como o Waterfall para abordagens ágeis, como Scrum, XP e Kanban. Esses métodos valorizam a adaptação, a colaboração e a entrega contínua de software funcional.

Princípios do Manifesto Ágil (2001):

➔ O Manifesto Ágil estabelece valores como comunicação, feedback constante, simplicidade e colaboração. Ele promove o envolvimento contínuo do cliente ao longo do processo de desenvolvimento e prioriza a entrega iterativa e incremental de software de alta qualidade.



Métodos Ágeis:

➔ Os métodos ágeis, como Extreme Programming (XP), Scrum e Kanban, fornecem estruturas e práticas para equipes de desenvolvimento ágil. Eles incentivam uma abordagem colaborativa, iterativa e focada no cliente para o desenvolvimento de software.

Extreme Programming (XP):

➔ XP enfatiza valores como comunicação, simplicidade e feedback. Suas práticas incluem **pair programming**, **testes automatizados** e **refactoring**, visando melhorar a qualidade do código e a produtividade da equipe.

Metodologias ágeis

Scrum:

➔ Scrum é uma estrutura bem definida com papéis específicos (Product Owner, Scrum Master, Time) e eventos (Sprints, Revisão, Retrospectiva). Ele promove a transparência no progresso do trabalho, priorização de tarefas e entrega incremental de software.

➔ INVEST

- É um acrônimo utilizado na metodologia ágil, especialmente no framework Scrum, para descrever boas características de histórias de usuário. Aqui estão as características de uma boa história INVEST:

- 1. Independente (Independent): A história deve ser independente, o que significa que ela pode ser desenvolvida e entregue sem depender de outras histórias. Isso permite que as equipes trabalhem de forma mais eficiente e flexível.

- 2. Negociável (Negotiable): As histórias devem ser negociáveis, o que significa que os detalhes podem ser discutidos e ajustados entre os membros da equipe e os interessados (stakeholders). Isso promove a colaboração e a adaptação contínua durante o desenvolvimento do projeto.

- 3. Valiosa (Valuable): As histórias devem agregar valor ao usuário ou cliente final. Elas devem se concentrar em atender às necessidades do usuário e resolver problemas reais.

- 4. Estimável (Estimable): As histórias devem ser estimáveis, o que significa que a equipe pode determinar quanto esforço será necessário para implementá-las. Isso é essencial para o planejamento e a priorização do trabalho.

- 5. Pequena (Small): As histórias devem ser pequenas o suficiente para serem concluídas dentro de um único ciclo de iteração ou sprint. Isso permite que a equipe as entregue de forma rápida e iterativa, obtendo feedback valioso em um curto espaço de tempo.

- 6. Testável (Testable): As histórias devem ser testáveis, o que significa que é possível determinar se elas foram implementadas com sucesso. Critérios de aceitação claros e mensuráveis devem ser definidos para cada história, facilitando a verificação do trabalho realizado.

- Ao seguir essas características, as equipes podem criar histórias de usuário que são claras, gerenciáveis e focadas em atender às necessidades dos usuários finais, ao mesmo tempo em que mantêm a flexibilidade e a adaptabilidade durante o processo de desenvolvimento.

Kanban:

➔ Originado no sistema de produção da Toyota, Kanban é baseado em limitar o trabalho em progresso (WIP), visualizar o fluxo de trabalho e otimizar continuamente o processo. Ele fornece uma abordagem mais flexível e adaptável para o gerenciamento de projetos.

Quando não usar Métodos Ágeis:

➔ Existem situações em que métodos ágeis podem não ser apropriados, como em condições de mercado estáveis, projetos com requisitos claros e estáveis desde o início, ou quando os clientes não estão disponíveis para colaboração frequente. Em tais casos, abordagens mais tradicionais podem ser mais adequadas.

03 - Requisitos

Funcionais: Descrevem o que o sistema deve fazer, como abrir e fechar contas, fornecer extratos e realizar transações.

Não-funcionais: Estabelecem restrições sobre o sistema, como desempenho, segurança e usabilidade.

Exemplo de Requisitos:

Sistema de um Banco:

Requisitos Funcionais:

- Abrir e fechar contas bancárias.
- Fornecer extratos detalhados e saldo atualizado.
- Realizar transferências entre contas.
- Permitir saques de fundos.
- Entre outros serviços bancários essenciais.

Requisitos Não-funcionais:

- Garantir que o sistema informe o saldo em menos de 5 segundos.
- Manter o sistema disponível 99.99% do tempo.
- Capacidade de continuar operando mesmo se houver falhas no centro de dados.
- Criptografar dados sensíveis transmitidos entre o banco e suas agências.
- Proteger a privacidade dos clientes, não compartilhando seus dados com terceiros.
- Integrar-se perfeitamente com os sistemas do Banco Central.
- Capacidade de armazenamento para atender as necessidades de até 1 milhão de clientes.
- Oferecer uma interface usável para clientes com deficiência visual.

Elicitação de Requisitos:

Na engenharia de requisitos de software: Elicitação refere-se ao processo de identificar, descobrir e documentar os requisitos de um sistema de software por meio de entrevistas, workshops, prototipagem e outras técnicas de coleta de informações.

➔ Processo de descobrir e entender os principais requisitos do sistema, através de técnicas como **histórias de usuários** e **casos de uso**.

Histórias de Usuários:

➔ Utilizam cartões, conversas e confirmações para descrever requisitos funcionais de forma concisa e compreensível.

Casos de Uso:

➔ Detalham interações entre atores (usuários) e o sistema, incluindo fluxos normais e extensões para situações excepcionais.

Produto Mínimo Viável (MVP):

➔ Estratégia para validar uma ideia com o menor conjunto de funcionalidades viáveis, priorizando o aprendizado e feedback do usuário.

Testes A/B:

➔ Método para comparar duas implementações de requisitos, permitindo decisões baseadas em dados reais de uso e preferência do usuário.

Comentários Finais:

➔ Testes A/B são amplamente utilizados por grandes empresas da internet e requerem amostras significativas para resultados estatisticamente válidos.

04 - Controle de Versões com Git:

➔ Importância para facilitar o desenvolvimento colaborativo, armazenando versões do sistema e permitindo recuperar edições anteriores.
➔ Comparação entre sistemas centralizados (como SVN, CVS) e distribuídos (como Git).
➔ Vantagens dos sistemas distribuídos, como maior frequência de commits, velocidade e capacidade de trabalhar offline.

Git Básico:

- ➔ Utilização dos comandos essenciais do Git, como git add, git commit, git push e git pull.
- ➔ Resolução de conflitos de merge entre diferentes branches.

Integração Contínua:

- ➔ Significado e importância da integração contínua, onde desenvolvedores compartilham seus commits assim que são saudáveis.
- ➔ Estratégias de branching para garantir a integração de código e a subsequente implantação em produção.
- ➔ Práticas recomendadas para manter a saúde do produto, incluindo automação de build, testes automatizados e uso de servidores de CI.

Estratégias de Branching:

- ➔ Explanação sobre padrões básicos de branching, como Mainline e Healthy Branch.
- ➔ Detalhamento dos padrões de integração, como Mainline Integration.
- ➔ Utilização de Release Branch para preparar uma versão do produto para release, mantendo a estabilidade.

Uso em Projetos:

- ➔ Aplicação do Healthy Mainline como método principal de integração de código.
- ➔ Utilização do Release Branch para estabilizar e preparar o código para produção.
- ➔ Implementação de pipelines no Github Actions para assegurar a integridade da mainline e realizar publicações na Azure.

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.