Tópicos de estudo para o exame

Utilizar este documento de apoio como uma revisão dos assuntos a serem estudados. Resulta essencialmente de compilar os objetivos de aprendizagem que já constam nas apresentações das aulas TP.

Revisto em: 2024-06-14

Conteúdos:

A) O que é que está incluído no SDLC?

O SDLC e o trabalho do Analista

- Explicar o que é o ciclo de vida de desenvolvimento de sistemas (SDLC)
    - O Ciclo de Vida de Desenvolvimento de Sistemas (SDLC - System Development Life Cycle) é um processo estruturado que envolve várias fases distintas para desenvolver, implementar e manter sistemas de informação ou software. Ele proporciona uma abordagem sistemática para assegurar que os projetos de software sejam bem planejados e executados de forma eficiente e eficaz.

- Descrever as principais atividades/assuntos dentro de cada uma das fases do SDLC (há autores que incluem 4 fases no SDLC, outros que incluem 5 fases com a Manutenção)
    - Planeamento: A fase de planeamento é crucial para entender por que um sistema de informação deve ser construído e como a equipe do projeto o realizará. Define a transformação digital pretendida e estabelece a base para o desenvolvimento e implementação do sistema.

      Passos principais: 1. Arranque do Projeto

      - **Identificação do Valor do Sistema:**
        - Determina-se o valor que o sistema trará para a organização.
      - **Pedido de Novo Sistema (Caderno de Encargos):**
        - Resumo das necessidades de negócio.
        - Explicação de como o sistema proposto atenderá a essas necessidades e criará valor de negócio.
      - **Apresentação e Aprovação:**
        - Apresentação do pedido e análise de viabilidade ao comité de aprovação.
        - Decisão sobre a realização do projeto.

      2. Gestão do Projeto

      - **Criação do Plano de Trabalho:**
        - Desenvolvimento de um plano detalhado com cronogramas e recursos.
      - **Atribuição da Equipe:**
        - Seleção e designação dos membros da equipe.
      - **Monitoramento e Direção:**
        - Implementação de técnicas para monitorar e garantir o progresso do projeto durante todo o ciclo de vida do sistema.

    - Analise: A fase de análise responde a quem utilizará o sistema, o que o sistema deve fazer e onde e como será utilizado. A equipe do projeto investiga sistemas atuais, identifica oportunidades de melhoria e desenvolve um conceito para o novo sistema.

      Passos Principais:

      1. Estudo do Domínio/Área e Análise dos Sistemas Existentes

      - **Investigação dos Sistemas Atuais:**
        - Avaliação de como as pessoas trabalham atualmente e como o novo sistema pode ajudar.

      2. Levantamento de Requisitos

      - **Interação com Stakeholders:**

- Coleta e sistematização das necessidades e capacidades requeridas para o novo sistema.

3. Conceito para a Solução (Proposta do Sistema)

- **Desenvolvimento de uma Proposta:**
  - Criação de uma solução que atende às necessidades identificadas.

- Desenho: A fase de desenho decide como o sistema será construído em termos de hardware, software, infraestrutura de rede, interface, formulários, relatórios, programas específicos, bases de dados e arquivos necessários. Inclui a escolha de tecnologias e a estratégia para a equipe de desenvolvimento.

Passos Principais:

1. Estratégia de Desenvolvimento

- **Decisão sobre Desenvolvimento:**
  - Determinar se o desenvolvimento será interno ou contratado.

2. Conceção da Arquitetura do Sistema

- **Definição da Arquitetura:**
  - Escolha entre opções como cloud ou desktop.
  - Decisão sobre a distribuição dos componentes em vários servidores ou em um nó central.

3. Conceção do Modelo de Dados

- **Detalhamento das Estruturas de Dados:**
  - Especificação do modelo da base de dados, por exemplo, PostgreSQL.

4. Desenho das Entidades de Software e Seleção de Frameworks

- **Estruturação dos Programas:**
  - Aplicação de princípios e boas práticas para estruturar o código.
  - Integração de plataformas de software existentes para resolver problemas comuns, como soluções de pagamento online.

- Implementação: Na fase de implementação, o sistema é de facto construído (ou adquirido, no caso de pacotes pré-feitos), com a escrita do código, integração de sistemas, desenvolvimento das bases de dados, verificação do software (testes),… Inclui também a transição para o ambiente de produção.

Passos Principais:

1. Implementação de Sistemas (Construção e Garantia da Qualidade)

- **Desenvolvimento de Código:**
  - Escrita e compilação do código.
- **Testes:**
  - Realização de testes unitários, de integração e de sistema.
- **Integração:**
  - Integração de módulos, frameworks e componentes do sistema.
- **Desenvolvimento de Interfaces:**
  - Criação e implementação das interfaces de usuário.

2. Instalação e Transição

- **Colocação em Produção:**
  - Implantação do sistema no ambiente de produção.
  - Migração de dados e configuração final do sistema.

3. Plano de Suporte

- **Revisões Pós-Instalação:**
  - Monitoramento e avaliação do desempenho do sistema.
- **Gestão de Modificações:**
  - Realização de ajustes, correções e melhorias contínuas após a entrada em produção.

- Definir o temo "processo de software" (*software process*)
  - **Definição**: Um processo de software é um conjunto de atividades, métodos, práticas e transformações usadas para desenvolver e manter sistemas de software.

- Distinguir atividades de análise do domínio (de aplicação) de atividades de especificação do (produto de) software.
  - Análise de Domínio: Consiste em entender o contexto e os requisitos do ambiente onde o software será aplicado, incluindo a identificação de conceitos e operações relevantes.

  - Especificação de Software: Foca em definir detalhadamente as funcionalidades e restrições do sistema a ser desenvolvido, com base nos requisitos levantados na análise de domínio.

- Descrever o papel e as responsabilidades do Analista no SDLC
  - O analista de sistemas analisa a situação do negócio, identifica oportunidades de melhorias e projeta um sistema de informação para implementá-las. Ser analista de sistemas é um dos trabalhos mais desafiantes na eng.a de software.
  - O principal objetivo de um analista de sistemas não é criar um sistema "topo de gama" (na perspetiva da tecnologia), mas criar valor para a organização
  - Agents of change Identify ways to improve the organization Motivate & train others
  - Skills needed

- Technical: must understand the technology
- Business: must know the business processes
- Analytical: must be able to solve problems
- Communications: technical & non technical audiences
- Interpersonal: leadership & management
- Ethics: deal fairly and protect confidential information

- Distinguir as competências de "análise de sistemas" das de "programação de sistemas", em engenharia de software. Relacionar com os conceitos de "soft skills" e "hard skills" com o papel do analista.

  Análise de Sistemas:
  - Envolve a compreensão dos requisitos e especificação do que o sistema deve fazer.
  - **Soft Skills**: Comunicação, liderança, ética.
  - **Hard Skills**: Modelagem de processos, definição de requisitos.

  Programação de Sistemas:
  - Envolve a implementação técnica das especificações usando linguagens de programação.
  - **Hard Skills**: Codificação, testes, depuração.
  - **Soft Skills**: Colaboração em equipe, comunicação técnica.

Processo de software e o Unified Process/OpenUP

- Descrever a estrutura do UP/OpenUP (fases e objetivos; iterações) ● Descrever os objetivos e principais atividades de cada fase do UP/OpenUP ● O OpenUP pode ser considerado "método ágil"?

- Porque é que o UP se assume como "orientado por casos de utilização, focado na arquitetura, iterativo e incremental"?
- Identificar características distintivas dos processos sequenciais, como a abordagem *waterfall*.
- Identificar as práticas distintivas dos métodos ágeis (o que há de novo no modelo de processo, comparando com a abordagem "tradicional"?).
- Distinguir projetos (de desenvolvimento de software) sequenciais de projetos evolutivos.

Estrutura do UP/OpenUP (Fases e Objetivos; Iterações)

Fases:

1. **Iniciação**:
   - **Objetivo**: Definir o escopo e obter financiamento.
   - **Atividades**: Estudo de viabilidade, definição do escopo, obtenção de recursos.
2. **Elaboração**:
   - **Objetivo**: Analisar requisitos e definir a arquitetura.
   - **Atividades**: Modelagem dos requisitos, desenvolvimento de protótipos, definição da arquitetura base.

3. **Construção**:
    - **Objetivo**: Desenvolver e testar o sistema.
    - **Atividades**: Codificação, testes, integração de componentes.
4. **Transição**:
    - **Objetivo**: Entregar o sistema aos usuários finais.
    - **Atividades**: Testes de aceitação, treinamento dos usuários, implantação.

Iterações: Cada fase é dividida em iterações curtas que resultam em incrementos do produto, permitindo ajustes contínuos e feedback.

Objetivos e Principais Atividades de Cada Fase do UP/OpenUP

- **Iniciação**: Validar a ideia do projeto, definir metas, riscos, e escopo inicial.
- **Elaboração**: Refinar requisitos, validar a arquitetura e preparar para a construção.
- **Construção**: Desenvolver o sistema de acordo com a arquitetura e requisitos estabelecidos.
- **Transição**: Realizar testes finais, corrigir defeitos, treinar usuários e implantar o sistema.

O OpenUP Pode Ser Considerado "Método Ágil"?

Sim, porque adota iterações curtas, feedback contínuo, e flexibilidade para mudanças, características fundamentais dos métodos ágeis.

Por que o UP é Orientado por Casos de Utilização, Focado na Arquitetura, Iterativo e Incremental?

- **Orientado por Casos de Utilização**: Identifica e descreve os requisitos e interações do usuário com o sistema.
- **Focado na Arquitetura**: Garante uma estrutura robusta e flexível para o desenvolvimento.
- **Iterativo e Incremental**: Permite desenvolvimento em pequenos ciclos, com entregas frequentes e ajustes baseados no feedback.

Características Distintivas dos Processos Sequenciais, como a Abordagem Waterfall

- **Linearidade**: Fases distintas e sequenciais sem sobreposição.
- **Planejamento Rígido**: Cada fase deve ser completada antes de iniciar a próxima.
- **Pouca Flexibilidade**: Dificuldade em incorporar mudanças durante o processo.

Práticas Distintivas dos Métodos Ágeis

- **Iterações Curtas**: Desenvolvimento em ciclos rápidos com entregas frequentes.
- **Feedback Contínuo**: Ajustes baseados no feedback constante do cliente.
- **Flexibilidade**: Capacidade de responder rapidamente a mudanças nos requisitos.
- **Colaboração**: Forte ênfase na comunicação e colaboração entre equipes e clientes.

Projetos Sequenciais vs. Projetos Evolutivos

- **Sequenciais**: Seguem uma sequência rígida de fases; dificilmente incorporam mudanças após o início de cada fase.
- **Evolutivos**: Desenvolvem o produto através de iterações sucessivas; permitem ajustes contínuos e melhorias incrementais.

Modelação visual e a UML

- Justifique o uso de modelos na engenharia de sistemas
- Descreva a diferença entre modelos funcionais, modelos estáticos/estruturais e modelos de comportamento.
- Enumerar as vantagens dos modelos visuais.
- Explicar a organização da UML (classificação dos diagramas)
- Caraterizar o "ponto de vista" (perspetiva) de modelação de cada diagrama da UML usado nas aulas Práticas.
- Relacionar os diagramas UML com o momento em que são aplicados, ao longo do projeto de desenvolvimento.

Justificação do Uso de Modelos na Engenharia de Sistemas

Os modelos na engenharia de sistemas são essenciais porque oferecem representações abstratas e simplificadas de sistemas complexos. Eles ajudam a:

- **Compreensão e Comunicação**: Permitem aos stakeholders entenderem melhor o sistema, suas partes e interações.
- **Análise e Projeto**: Facilitam a análise de requisitos, a definição de arquitetura e o design do sistema antes da implementação.
- **Validação e Verificação**: Permitem validar o sistema antes da construção, identificando inconsistências e erros precocemente.
- **Documentação**: Servem como documentação visual do sistema, facilitando manutenção e evolução futuras.

Diferença entre Modelos Funcionais, Modelos Estáticos/Estruturais e Modelos de Comportamento

- **Modelos Funcionais**: Descrevem as funcionalidades e serviços oferecidos pelo sistema, focando nas interações com usuários e outros sistemas.
- **Modelos Estáticos/Estruturais**: Representam a estrutura estática do sistema, mostrando entidades, seus atributos e relacionamentos, como diagramas de classes na UML.
- **Modelos de Comportamento**: Descrevem como o sistema funciona ao longo do tempo, mostrando estados, transições de estados e interações entre objetos, como diagramas de sequência e diagramas de atividades na UML.

Vantagens dos Modelos Visuais

- **Comunicação Efetiva**: Facilitam a comunicação entre stakeholders técnicos e não técnicos.
- **Compreensão Melhorada**: Permitem uma compreensão mais rápida e profunda do sistema.
- **Detecção Precoce de Problemas**: Permitem identificar inconsistências e erros no design antes da implementação.
- **Documentação Clara e Concisa**: Servem como documentação visual do sistema, facilitando a manutenção e o entendimento futuro.

Organização da UML (Classificação dos Diagramas)

A UML organiza seus diagramas em duas grandes categorias:

- **Diagramas Estruturais**: Descrevem a estrutura estática do sistema.

    - Exemplos: Diagrama de Classes, Diagrama de Objetos, Diagrama de Componentes, Diagrama de Pacotes.
- **Diagramas Comportamentais**: Descrevem o comportamento dinâmico do sistema ao longo do tempo.
    - Exemplos: Diagrama de Casos de Uso, Diagrama de Sequência, Diagrama de Atividades, Diagrama de Estados.

Ponto de Vista de Modelação de Cada Diagrama da UML Usado nas Aulas Práticas

- **Diagrama de Classes**: Modela a estrutura estática do sistema, mostrando classes, atributos, métodos e seus relacionamentos.
- **Diagrama de Casos de Uso**: Foca nos requisitos funcionais do sistema, mostrando interações entre atores externos e casos de uso.
- **Diagrama de Sequência**: Descreve interações entre objetos em uma sequência temporal, mostrando como mensagens são trocadas.
- **Diagrama de Atividades**: Modela o fluxo de controle entre atividades no sistema, mostrando a lógica de execução.
- **Diagrama de Estados**: Descreve os estados que um objeto pode ter ao longo de seu ciclo de vida e as transições entre esses estados.

Relação dos Diagramas UML com o Momento em que São Aplicados ao Longo do Projeto de Desenvolvimento

- **Diagrama de Casos de Uso**: Aplicado na fase de requisitos, para capturar as funcionalidades do sistema.
- **Diagrama de Classes**: Aplicado na fase de design, para modelar a estrutura estática do sistema.
- **Diagrama de Sequência e Diagrama de Atividades**: Aplicados na fase de design detalhado e implementação, para descrever o comportamento dinâmico e as interações do sistema.
- **Diagrama de Estados**: Aplicado na fase de design para modelar o comportamento dos objetos ao longo do tempo, especialmente útil para sistemas com estados complexos.

B) Compreender as necessidades do negócio (atividades e resultados da Análise)

Práticas de engenharia de requisitos

- Distinguir entre requisitos funcionais e não funcionais
- Apresentar técnicas de recolha de requisitos e recomendá-las para diferentes tipos de projeto.
- Distinguir entre abordagens centradas em cenários (utilização) e abordagens centradas no produto para a determinação de requisitos
- Identificar, numa lista, requisitos funcionais e atributos de qualidade.
- Justifique que “a determinação de requisitos é mais que a recolha de requisitos”.
- Identifique requisitos bem e mal formulados (aplicando os critérios S.M.A.R.T.)
- Identifique requisitos bem e mal formulados (aplicando os critérios do ISO-IEEE 29148)
- Discutir as “verdades incontornáveis” apresentadas por Wiegers, sobre os requisitos de sistemas software [original, cópia disponível no material das TP].
- Identificar/exemplificar regras de negócio (distinguindo-as do conceito de requisitos).

- Qual a abordagem proposta no OpenUp para a documentação de requisitos de um produto de software (*outcomes* relacionados)?
- Comentar a afirmação "o processo de determinação de requisitos (*requirements elicitation*) é primeiramente um desafio de interação humana".

Distinguir entre requisitos funcionais e não funcionais:

- **Requisitos Funcionais:** Descrevem o que o sistema deve fazer, suas funcionalidades específicas.
- **Requisitos Não Funcionais:** Definem atributos do sistema, como desempenho, segurança, usabilidade, entre outros.

Apresentar técnicas de recolha de requisitos e recomendá-las para diferentes tipos de projeto:

- **Entrevistas:** Para entender necessidades dos usuários.
- **Workshops:** Para envolver stakeholders na definição de requisitos.
- **Observação:** Para capturar requisitos baseados em comportamentos observados.
- **Prototipagem:** Para validar requisitos de usabilidade e funcionalidade.

Distinguir entre abordagens centradas em cenários e centradas no produto:

- **Centradas em Cenários:** Focam em casos de uso específicos para capturar requisitos.
- **Centradas no Produto:** Enfatizam a definição de características e atributos do produto como um todo.

Identificar requisitos funcionais e atributos de qualidade:

- **Requisitos Funcionais:** Cadastro de usuários, geração de relatórios.
- **Atributos de Qualidade:** Desempenho, segurança, confiabilidade.

Justificar que "a determinação de requisitos é mais que a recolha de requisitos":

- Inclui análise, validação, verificação e documentação para garantir que os requisitos capturados sejam claros, completos e consistentes.

Identificar requisitos bem e mal formulados (S.M.A.R.T.):

- **Bem Formulados:** Específicos, Mensuráveis, Atingíveis, Relevantes, Temporais.
- **Mal Formulados:** Vagos, Ambíguos, Não Mensuráveis, Impraticáveis, Não Relevantes no tempo correto.

Identificar requisitos bem e mal formulados (ISO-IEEE 29148):

- **Bem Formulados:** Completos, Consistentes, Corretos, Não Ambíguos.
- **Mal Formulados:** Incompletos, Inconsistentes, Ambíguos, Incorretos.

Discutir as "verdades incontornáveis" de Wiegers sobre requisitos de sistemas software:

- Inclui afirmações sobre a natureza mutável dos requisitos, a necessidade de gerenciamento eficaz e a importância da comunicação clara entre stakeholders.

Identificar/exemplificar regras de negócio (distinguindo-as de requisitos):

- **Regras de Negócio:** Políticas ou diretrizes que definem como uma organização opera.
- **Requisitos:** Especificações precisas de funcionalidades ou atributos do sistema.

Abordagem proposta no OpenUp para documentação de requisitos:

- Enfatiza a criação de artefatos claros e concisos, como casos de uso e especificações detalhadas, para capturar e validar requisitos.

Comentar a afirmação "o processo de determinação de requisitos é primeiramente um desafio de interação humana":

- Destaca que entender e conciliar diferentes perspectivas de stakeholders é crucial para capturar requisitos precisos e adequados às necessidades do sistema.

Modelação funcional com casos de utilização

- Descrever o processo usado para identificar casos de utilização.
- Ler e criar diagramas de casos de utilização.
- Rever modelos de casos de utilização existentes para detetar problemas semânticos e sintáticos.
- Descrever os elementos essenciais de uma especificação de caso de uso.
- Explicar o uso complementar de diagramas de casos de utilização, diagramas de atividades e narrativas de casos de utilização.
- Explicar o sentido da expressão "desenvolvimento orientado por casos de utilização".
- Explicar os seis "Princípios para a adoção de casos de utilização" propostos por Ivar Jacobson (com relação ao "Use Cases 2.0")
- Explicar a relação entre requisitos e os casos de utilização
- Identificar as disciplinas e atividades relacionadas aos requisitos no OpenUP
- Relacionar o caso de utilização (entidade de modelação) com os cenários (formas de percorrer o caso de uso).
- Identifique o uso adequado de classes de associação.
- Como é que o empacotamento dos casos de uso (resultado da análise), pode contribuir para a identificação de potenciais módulos, na arquitetura?

Processo usado para identificar casos de utilização:

- Envolve entrevistar stakeholders para entender seus objetivos e requisitos, identificar os principais cenários de uso do sistema e documentar esses cenários como casos de uso.

Ler e criar diagramas de casos de utilização:

- Diagramas de Casos de Uso representam interações entre atores (usuários) e o sistema, mostrando os cenários de uso e suas sequências.

Rever modelos de casos de utilização existentes para detectar problemas semânticos e sintáticos:

- Problemas semânticos: inconsistências ou ambiguidades nos requisitos capturados.
- Problemas sintáticos: estrutura ou formato inadequado dos casos de uso conforme as normas UML.

Elementos essenciais de uma especificação de caso de uso:

- Nome do caso de uso, atores envolvidos, pré-condições, fluxo principal de eventos, fluxos alternativos e pós-condições.

Uso complementar de diagramas de casos de utilização, diagramas de atividades e narrativas de casos de utilização:

- Diagramas de Casos de Uso representam interações; Diagramas de Atividades detalham os processos internos dos casos de uso; Narrativas complementam com descrições textuais.

Sentido da expressão "desenvolvimento orientado por casos de utilização":

- Foco na definição de requisitos através de cenários de uso concretos, orientando o desenvolvimento de software pela compreensão detalhada dos casos de uso.

Princípios para a adoção de casos de utilização propostos por Ivar Jacobson (Use Cases 2.0):

1. **Valor** para os stakeholders.
2. **Foco** nos objetivos do sistema.
3. **Clareza** na comunicação dos requisitos.
4. **Evolução** contínua dos casos de uso.
5. **Colaboração** entre equipes.
6. **Motivação** para stakeholders.

Relação entre requisitos e casos de utilização:

- Casos de Uso capturam requisitos funcionais do sistema através de cenários específicos de interação.

Disciplinas e atividades relacionadas aos requisitos no OpenUP:

- Análise de Requisitos, Modelagem de Casos de Uso, Verificação e Validação de Requisitos.

Relação entre caso de utilização (entidade de modelação) e cenários (formas de percorrer o caso de uso):

- Caso de Uso representa a entidade de modelação; cenários são os diferentes caminhos ou sequências de ações dentro desse caso de uso.

Uso adequado de classes de associação:

- Associar atores a casos de uso para representar interações entre usuários e sistema.

Contribuição do empacotamento dos casos de uso para identificação de potenciais módulos na arquitetura:

- Agrupar casos de uso relacionados pode sugerir agrupamentos funcionais ou módulos no sistema, facilitando o design da arquitetura.

A modelação do contexto do problema: modelo do domínio/negócio

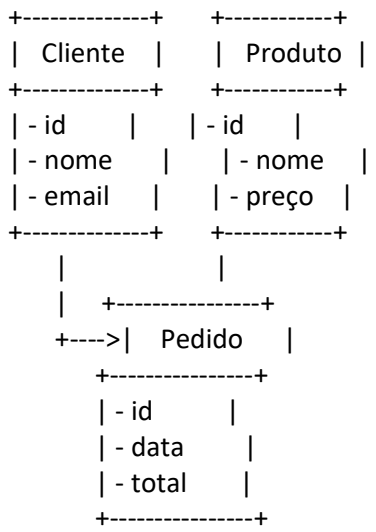
Caraterizar os conceitos do domínio de aplicação:

- Desenhe um diagrama de classes simples para capturar os conceitos de um domínio de problema.
- Apresente duas estratégias para descobrir sistematicamente os conceitos candidatos para incluir no modelo de domínio.
- Identificar construções específicas (associadas à implementação) que podem poluir o modelo de domínio (na etapa de análise).

Caraterizar os processos do negócio/organizacionais:

- Leia e desenhe diagramas de atividades para descrever os fluxos de trabalho da organização / negócios.
- Identifique o uso adequado de ações, fluxo de controle, fluxo de objetos, eventos e partições com relação a uma determinada descrição de um processo.
- Relacione os "conceitos da área do negócio" (classes no modelo de domínio) com fluxos de objetos nos modelos de atividade.

- **Diagrama de Classes Simples:**
  - Exemplo:



Estratégias para Descobrir Conceitos Candidatos:

1. **Entrevistas e Workshops:** Interagir com stakeholders para identificar termos frequentemente mencionados.
2. **Análise Documental:** Revisar documentos existentes, como relatórios de negócio e especificações, para identificar termos chave.

Construções que Podem Poluir o Modelo de Domínio:

- **Detalhes de Implementação:** Exemplos incluem classes técnicas de infraestrutura ou termos muito específicos de uma tecnologia particular que não refletem conceitos de negócio essenciais.

Caraterização dos Processos do Negócio/Organizacionais:

Leitura e Desenho de Diagramas de Atividades:

- **Fluxos de Trabalho da Organização/Negócio:** Diagramas de atividades descrevem sequências de atividades dentro de um processo, como o fluxo de um pedido de compra.

Uso Adequado de Elementos nos Diagramas de Atividades:

- **Ações:** Representam atividades realizadas no processo.
- **Fluxo de Controle:** Define a sequência das ações.
- **Fluxo de Objetos:** Indica como os objetos (dados) são passados entre as atividades.
- **Eventos:** Indicam o início ou término de atividades baseadas em condições.
- **Partições:** Agrupam atividades por responsabilidades ou papéis.

Relação entre Conceitos da Área do Negócio e Fluxos de Objetos:

- **Classes no Modelo de Domínio:** Representam entidades de negócio como Cliente, Produto, etc.
- **Fluxos de Objetos nos Modelos de Atividade:** Mostram como essas entidades (objetos) são criados, manipulados ou utilizados durante as atividades do processo.

C) Modelos no desenvolvimento

Orientação aos objetos no SDLC

- Discutir as diferenças e complementaridades entre os conceitos de orientação aos objetos na Análise (OOA), no Desenho (OOD) e na programação (OOP).
- Apresentar a distribuição de responsabilidades e a colaboração mecanismos de base na orientação aos objetos.

Diferenças e Complementaridades entre OOA, OOD e OOP:

1. **Análise Orientada a Objetos (OOA):**
    - **Objetivo:** Identificar e modelar conceitos do mundo real como objetos e suas interações.
    - **Atividades:** Identificação de classes, atributos, métodos e suas relações através de diagramas como Diagramas de Classes, Diagramas de Casos de Uso, entre outros.
    - **Foco:** Capturar requisitos funcionais e não funcionais do sistema.
2. **Desenho Orientado a Objetos (OOD):**
    - **Objetivo:** Transformar o modelo conceitual (OOA) em um design concreto do sistema.
    - **Atividades:** Refinar modelos de análise, definir arquitetura do sistema, detalhar classes e estruturas de dados, definir padrões de design.
    - **Foco:** Estruturação do sistema em componentes reutilizáveis e interconectados.
3. **Programação Orientada a Objetos (OOP):**
    - **Objetivo:** Implementar o design orientado a objetos em código executável.
    - **Atividades:** Codificação de classes, métodos, relações e interações baseadas nos modelos definidos em OOD.
    - **Foco:** Transformar modelos em software funcional, aplicando princípios como encapsulamento, herança e polimorfismo.

Complementaridades:

- **OOA e OOD:** OOA fornece a base conceitual e estrutural para o sistema, enquanto OOD traduz esses conceitos em uma estrutura de design detalhada e eficiente.
- **OOD e OOP:** OOD define como os objetos vão interagir e cooperar no sistema, enquanto OOP implementa essas interações em código executável, garantindo funcionalidade e eficiência.

Distribuição de Responsabilidades e Colaboração na Orientação aos Objetos

Distribuição de Responsabilidades:

- **Objetos:** São unidades que encapsulam dados e comportamentos relacionados.
- **Classes:** Especificam o design de objetos, definindo atributos e métodos.
- **Responsabilidade Única:** Princípio de design onde cada objeto ou classe deve ter uma única responsabilidade bem definida.

Colaboração Mecanismos de Base:

- **Associação:** Relação entre objetos onde um objeto utiliza serviços de outro.
- **Agregação e Composição:** Relações de todo-parte entre objetos.
- **Herança:** Mecanismo onde uma classe (subclasse) pode herdar características de outra classe (superclasse).
- **Polimorfismo:** Capacidade de objetos responderem de maneiras diferentes a mensagens iguais.

Benefícios da Orientação a Objetos:

- **Reutilização de Código:** Classes e objetos podem ser reutilizados em diferentes partes do sistema.
- **Manutenção Facilitada:** Mudanças em um objeto não afetam necessariamente outros objetos.
- **Encapsulamento:** Protege os detalhes internos dos objetos, promovendo modularidade e segurança.
- **Modelagem do Mundo Real:** Permite representar o sistema de forma mais próxima aos conceitos do mundo real.

Modelação estrutural

- Distinguir entre a análise de sistemas baseada numa abordagem algorítmica *top-down* e baseada nos conceitos do domínio do problema.
- Explicar a relação entre os diagramas de classe e de objetos.
- Rever um modelo de classes quanto a problemas de sintaxe e semânticos, considerando uma descrição do um problema de aplicação.
- Descreva os tipos e funções das diferentes associações no diagrama de classes.

Identifique o uso adequado da associação, composição e agregação para modelar a relação entre

Diferença entre Análise Baseada em Abordagem Algorítmica Top-Down e Baseada nos Conceitos do Domínio do Problema

Abordagem Algorítmica Top-Down:

- **Características:** Começa com uma visão geral do sistema e divide-o em partes menores e mais detalhadas.

- **Foco:** Divide o problema em etapas e subetapas detalhadas, enfatizando a decomposição funcional e a hierarquia de processos.
- **Aplicação:** Mais comum em sistemas onde a funcionalidade é claramente definida e a lógica do sistema pode ser decomposta em etapas sequenciais.

Análise Baseada nos Conceitos do Domínio do Problema:

- **Características:** Centra-se na compreensão dos conceitos fundamentais e entidades do domínio do problema.
- **Foco:** Identifica objetos, classes, relações e comportamentos com base nas necessidades e regras do domínio específico.
- **Aplicação:** Preferível quando o foco está na modelagem precisa e na representação fiel dos elementos do mundo real que o sistema está tentando abordar.

Relação entre Diagramas de Classe e de Objetos

- **Diagrama de Classe:** Representa a estrutura estática de um sistema, mostrando classes, atributos, métodos e suas relações.
- **Diagrama de Objetos:** Mostra uma instância particular de um ou mais objetos no sistema em um momento específico.
- **Relação:** Um diagrama de objetos pode ser derivado de um diagrama de classe, onde os objetos são instâncias das classes definidas no diagrama de classe.

Revisão de Modelo de Classes quanto a Problemas de Sintaxe e Semântica

- **Sintaxe:** Refere-se à correta estrutura e notação utilizada no diagrama de classes, como nomes de classes, atributos e métodos.
- **Semântica:** Refere-se à correção lógica e significado dos elementos modelados. Por exemplo, se as associações representam corretamente os relacionamentos entre as classes conforme definido nos requisitos do sistema.

Tipos e Funções das Associações no Diagrama de Classes

1. **Associação:**
    - **Função:** Relacionamento básico entre duas classes.
    - **Exemplo de Uso Adequado:** Relacionar um cliente com várias ordens de compra.
2. **Composição:**
    - **Função:** Tipo forte de associação onde uma classe (todo) possui outra classe (parte) como parte integrante.
    - **Exemplo de Uso Adequado:** Um carro tem um motor. A existência do carro depende do motor.
3. **Agregação:**
    - **Função:** Tipo de associação que indica uma relação de "todo-parte", mas menos restritiva que a composição.
    - **Exemplo de Uso Adequado:** Um departamento tem vários funcionários. Os funcionários podem existir independentemente do departamento.

Modelagem da Relação entre Classes usando Associação, Composição e Agregação

- **Associação:** Usada quando duas classes estão relacionadas de alguma forma, mas não há dependência forte entre elas.

- **Composição:** Usada quando uma classe (todo) possui outra classe (parte) como parte essencial, ou seja, a parte não pode existir sem o todo.
- **Agregação:** Usada quando uma classe (todo) possui outra classe (parte) como parte opcional, ou seja, a parte pode existir independentemente do todo.

Exemplos:

- **Associação:** Uma biblioteca possui vários livros. A relação entre biblioteca e livro é uma associação simples.
- **Composição:** Uma pessoa tem um coração. Sem a pessoa, o coração não tem razão de existir.
- **Agregação:** Um departamento possui vários funcionários. Os funcionários podem trabalhar em diferentes departamentos ou até mesmo fora deles.

Modelos de comportamento

- Explique o papel da modelação de comportamento no SDLC
- Explicar a complementaridade entre diagramas de sequência e de comunicação
- Relacionar a ideia essencial (na orientação aos objetos) de distribuição de responsabilidades com o diagrama de sequência (como é que ajuda?).
- Relacionar representações nos diagramas de sequência com código por objetos e vice-versa.
- Representar o ciclo de vida de uma entidade num diagrama de estados.
- Relacionar elementos presentes num D. Sequência com as entidades de um D. de Classes.
- Como é que o desenvolvimento de um modelo de colaboração entre objetos pode ser conduzido a partir dos casos de utilização? (*use case realizations*)

Papel da Modelação de Comportamento no SDLC

A modelação de comportamento no Software Development Life Cycle (SDLC) é crucial para entender como os diversos componentes do sistema interagem e se comportam em diferentes cenários. Ela permite:

- **Especificação de Requisitos:** Captura como os usuários interagem com o sistema e quais funcionalidades são necessárias.
- **Projeto Detalhado:** Define como os objetos e componentes do sistema colaboram para alcançar os requisitos.
- **Implementação:** Serve de base para traduzir o comportamento modelado em código executável.
- **Testes:** Facilita a criação de casos de teste para verificar se o sistema se comporta conforme o esperado.

Complementaridade entre Diagramas de Sequência e de Comunicação

- **Diagrama de Sequência:** Mostra a interação entre objetos ao longo do tempo, enfatizando a ordem das mensagens trocadas entre eles.
- **Diagrama de Comunicação:** Foca nos relacionamentos entre objetos, exibindo a estrutura estática das interações.

Complementaridade: Ambos os diagramas ajudam a entender como os objetos se comunicam, mas de perspectivas diferentes. O diagrama de sequência é mais detalhado temporalmente, enquanto o diagrama de comunicação é mais focado na estrutura das interações.

Distribuição de Responsabilidades e Diagrama de Sequência

- **Distribuição de Responsabilidades:** Na orientação a objetos, cada objeto é responsável por uma parte específica do comportamento do sistema.
- **Diagrama de Sequência:** Ajuda a visualizar como essas responsabilidades são distribuídas entre os objetos durante a execução de um cenário específico.

Como ajuda: O diagrama de sequência permite identificar quais objetos estão envolvidos em uma interação específica e como eles colaboram para realizar uma determinada função do sistema.

Relação entre Diagramas de Sequência e Diagramas de Classes

- **Diagrama de Sequência:** Mostra interações entre objetos em um cenário específico, destacando mensagens e a ordem das operações.
- **Diagrama de Classes:** Representa a estrutura estática do sistema, mostrando classes, atributos, métodos e associações entre objetos.

Relação: No diagrama de sequência, os objetos envolvidos são instâncias das classes definidas no diagrama de classes. Assim, os objetos no diagrama de sequência são representações das entidades (classes) no diagrama de classes.

Desenvolvimento de um Modelo de Colaboração a partir de Casos de Uso (Use Case Realizations)

- **Use Case Realizations:** São modelos detalhados que descrevem como os casos de uso são realizados através da interação entre objetos.
- **Processo:** A partir dos casos de uso, identifica-se quais objetos participam na realização de cada caso de uso e como eles colaboram entre si para alcançar os objetivos do caso de uso.

Como é conduzido: Durante a análise e design, os use case realizations são refinados usando diagramas de sequência e outros modelos de comportamento para especificar como os objetos colaboram para executar os passos do caso de uso.

Vistas de arquitetura

- Explicar as atividades associadas ao desenvolvimento de arquitetura de software, no openUP.
- Explicar a relação entre requisitos e a arquitetura (como é que aqueles influenciam esta). Exemplificar requisitos com impacto na arquitetura.
- Explique a prática de “arquitetura evolutiva” proposta no OpenUp.
- Identifique as camadas e partições numa arquitetura de software por camadas.
- Usar diagramas de sequência para descrever a cooperação entre módulos/elementos de uma arquitetura.
- Identifique os três tipos de estruturas principais, constituintes de uma arquitetura (segundo L. Bass *et al*). Relacionar essas categorias com os diagramas de UML mais relevantes para as documentar.
- Discutir razões técnicas e não técnicas que justificam o desenvolvimento de uma (boa) arquitetura para o sistema a construir.

Atividades associadas ao desenvolvimento de arquitetura de software no OpenUP

No OpenUP, as atividades associadas ao desenvolvimento de arquitetura de software incluem:

- **Definição de Requisitos Arquiteturais:** Identificação e análise dos requisitos que influenciam decisões arquiteturais.
- **Elaboração da Arquitetura Inicial:** Definição da estrutura inicial do sistema, incluindo a identificação de componentes principais e suas interações.
- **Refinamento da Arquitetura:** Iterações para detalhar e aprimorar a arquitetura à medida que mais requisitos são especificados e entendidos.
- **Validação da Arquitetura:** Verificação de que a arquitetura atende aos requisitos funcionais e não funcionais do sistema.

Relação entre Requisitos e Arquitetura

Os requisitos funcionais e não funcionais influenciam diretamente a arquitetura de software:

- **Requisitos Funcionais:** Especificam o comportamento do sistema, influenciando decisões sobre a estrutura interna, interfaces e componentes do sistema.
- **Requisitos Não Funcionais:** Definem as propriedades do sistema, como desempenho, segurança, escalabilidade, entre outros, que moldam escolhas arquiteturais.

Exemplo: Um requisito de alta disponibilidade pode exigir a adoção de uma arquitetura distribuída com redundância de servidores e balanceamento de carga.

Prática de "Arquitetura Evolutiva" no OpenUP

A "arquitetura evolutiva" no OpenUP enfatiza:

- **Incrementalidade e Iteratividade:** A arquitetura é desenvolvida de forma iterativa, adaptando-se conforme novos requisitos são compreendidos e incorporados.
- **Flexibilidade:** Permite ajustes e refinamentos contínuos da arquitetura à medida que o projeto avança e novos desafios são identificados.
- **Validação Contínua:** A arquitetura é validada continuamente para garantir que atenda aos requisitos emergentes e às necessidades do sistema.

Camadas e Partições numa Arquitetura de Software por Camadas

Em uma arquitetura por camadas, temos comumente:

- **Camadas:** São agrupamentos lógicos de funcionalidades ou serviços relacionados, como interface de usuário, lógica de negócio e persistência de dados.
- **Partições:** São divisões dentro de uma camada que separam diferentes aspectos ou responsabilidades funcionais, permitindo uma maior organização e modularidade.

Uso de Diagramas de Sequência para Descrever Cooperação entre Módulos/Elementos da Arquitetura

Os diagramas de sequência são úteis para representar a interação entre módulos ou elementos da arquitetura, mostrando a troca de mensagens e o fluxo de controle entre eles em um cenário específico.

Três Tipos de Estruturas Principais de uma Arquitetura (segundo L. Bass et al)

Segundo L. Bass et al., as três estruturas principais de uma arquitetura são:

- **Estrutura de Módulos:** Define como o sistema é dividido em partes ou módulos independentes com responsabilidades específicas.
- **Estrutura de Conexões:** Descreve como os módulos se comunicam entre si, incluindo protocolos de comunicação e interfaces.
- **Estrutura de Controle:** Define o fluxo de controle entre os módulos, especificando como decisões são tomadas e como o controle é passado entre componentes.

Diagramas UML Relevantes: Os diagramas de classes, diagramas de componentes e diagramas de implantação são utilizados para documentar essas estruturas na UML.

Razões Técnicas e Não Técnicas para o Desenvolvimento de uma Boa Arquitetura

- **Técnicas:**
    - **Modularidade e Reusabilidade:** Facilita a manutenção e evolução do sistema ao isolar mudanças e promover o reuso de componentes.
    - **Desempenho e Escalabilidade:** Permite o dimensionamento do sistema para suportar um aumento na carga ou complexidade.
    - **Segurança e Confiabilidade:** Garante que o sistema seja robusto contra falhas e vulnerabilidades.
- **Não Técnicas:**
    - **Alinhamento com Requisitos do Negócio:** Assegura que a arquitetura suporte os objetivos e necessidades do negócio.
    - **Facilitação da Comunicação:** Fornece um meio claro para discutir e comunicar o design do sistema entre stakeholders.
    - **Suporte a Decisões Futuras:** Antecipa mudanças e evoluções no sistema, facilitando ajustes sem grandes impactos.

Desenho do software (perspetiva do programador)

- Explicar como os casos de utilização podem ser usados para orientar as atividades de desenho (na perspetiva do livro do Larman).
- Explicar os princípios do baixo acoplamento e alta coesão em OO. Discutir implicações do *coupling* e *cohesion*.
- Comparar, num dado desenho por objetos, a ocorrência de maior/menor *coupling* e *cohesion*.
- Relacionar código por objetos com a sua representação em diagramas de classes da UML.
- Modelar a interação entre unidades de software (objetos) como diagramas de sequência.
- Construa um diagrama de classes e um diagrama de sequência considerando um código Java.
- Explicar as implicações no código da navegabilidade modelada no diagrama de classes.

Como os casos de utilização podem ser usados para orientar as atividades de desenho (na perspetiva do livro do Larman)

Segundo Craig Larman, os casos de uso são essenciais para guiar o desenho orientado a objetos, pois:

- **Capturam Requisitos Funcionais:** Os casos de uso identificam interações entre atores externos e o sistema, descrevendo o comportamento esperado.
- **Servem como Base para o Desenho:** Cada caso de uso pode ser mapeado em colaborações de objetos, identificando as classes envolvidas e suas responsabilidades.
- **Promovem Modularidade e Reuso:** Ao identificar cenários específicos, os casos de uso ajudam na definição de interfaces claras entre os componentes do sistema, promovendo a modularidade e o reuso de código.

Princípios do Baixo Acoplamento e Alta Coesão em OO

- **Baixo Acoplamento:** Refere-se ao grau de interdependência entre módulos ou classes. Baixo acoplamento significa que as classes têm pouca dependência entre si, o que facilita a manutenção e evolução do sistema.
- **Alta Coesão:** Refere-se ao grau em que os elementos dentro de um módulo ou classe estão relacionados entre si. Alta coesão indica que os elementos de uma classe estão fortemente relacionados e trabalham juntos para realizar uma única tarefa.

Comparação de Coupling e Cohesion em um Desenho por Objetos

Em um desenho por objetos, podemos comparar diferentes níveis de coupling (acoplamento) e cohesion (coesão):

- **Maior Coupling:** Classes fortemente acopladas têm muitas dependências entre si. Alterações em uma classe podem exigir alterações em várias outras classes, aumentando o risco de efeitos colaterais.
- **Menor Coupling:** Classes fracamente acopladas têm poucas dependências entre si. Isso facilita a manutenção e o teste, pois mudanças em uma classe têm menos impacto nas outras.
- **Maior Coesão:** Classes com alta coesão têm métodos e atributos que estão relacionados e trabalham juntos para realizar uma única responsabilidade. Isso facilita a compreensão e a manutenção do código.

Relação entre Código por Objetos e Diagramas de Classes da UML

- **Código por Objetos:** Implementa a estrutura e o comportamento de classes e objetos em uma linguagem de programação específica, como Java. Define como os objetos interagem e colaboram para cumprir as funcionalidades do sistema.
- **Diagramas de Classes da UML:** Representam a estrutura estática do sistema, mostrando as classes, seus atributos, métodos e relacionamentos. Servem como uma representação visual do design orientado a objetos.

Modelagem da Interação entre Unidades de Software (Objetos) como Diagramas de Sequência

- **Diagramas de Sequência:** Descrevem como grupos de objetos colaboram em uma sequência específica para realizar uma funcionalidade do sistema. Mostram as mensagens trocadas entre os objetos ao longo do tempo.

D) Práticas selecionadas na construção do software

Garantia de qualidade

- Identifique as atividades de validação e verificação incluídas no SDLC
- Descreva quais são as camadas da pirâmide de teste
- Descreva o assunto/objetivo dos testes de unidade, integração, sistema e de aceitação
- Explique o ciclo de vida do TDD
- Descreva as abordagens "debug-later" e "test-driven", de acordo com J. Grenning.
- Explique como é que as atividades de garantia de qualidade (QA) são inseridas no processo de desenvolvimento, numa abordagem clássica e nos métodos ágeis.
- O que é o "V-model"?
- Relacione os critérios de aceitação da história (*user-story*) com o teste *Agile*. Explique o sentido da afirmação "especificações executáveis".
- Justifique a necessidade de testes "*developer facing*" e "*custommer facing*".

Atividades de Validação e Verificação no SDLC

- **Validação:** Verifica se o software atende aos requisitos do usuário e se está sendo construído corretamente.
- **Verificação:** Garante que o software está sendo desenvolvido de acordo com os padrões e especificações definidas.

Camadas da Pirâmide de Testes

A pirâmide de testes sugere uma distribuição hierárquica dos testes, com maior foco nos níveis mais baixos:

- **Testes Unitários:** Testam unidades individuais de código, como métodos ou funções.
- **Testes de Integração:** Verificam a integração entre componentes ou módulos.
- **Testes de Sistema:** Validam o comportamento do sistema como um todo, conforme os requisitos.
- **Testes de Aceitação:** Garantem que o sistema atenda aos critérios de aceitação do usuário.

Objetivos dos Testes de Unidade, Integração, Sistema e de Aceitação

- **Testes de Unidade:** Validam unidades individuais de código para garantir que funcionem corretamente.
- **Testes de Integração:** Verificam a interação entre unidades ou módulos para garantir que funcionem juntos corretamente.
- **Testes de Sistema:** Avaliam o sistema como um todo para garantir que atenda aos requisitos funcionais e não funcionais.
- **Testes de Aceitação:** Confirmam que o sistema atende aos critérios de aceitação do usuário e está pronto para ser entregue.

Ciclo de Vida do TDD (Test-Driven Development)

- **Red:** Escrever um teste automatizado que falhe inicialmente.
- **Green:** Escrever o código mínimo necessário para passar no teste.
- **Refactor:** Refatorar o código para melhorar a estrutura, sem alterar o comportamento.

Abordagens "Debug-Later" e "Test-Driven" (segundo J. Grenning)

- **Debug-Later:** Desenvolver o código primeiro e, em seguida, realizar depuração para encontrar e corrigir erros.
- **Test-Driven Development (TDD):** Escrever testes automatizados antes de implementar o código, usando esses testes para guiar o desenvolvimento.

Inserção de Atividades de Garantia de Qualidade (QA) no Processo de Desenvolvimento

- **Abordagem Clássica:** QA é geralmente uma fase separada após o desenvolvimento, onde são realizados testes de aceitação e validação.
- **Métodos Ágeis:** QA é integrado ao longo do ciclo de vida do desenvolvimento, com testes contínuos e revisões de código para garantir a qualidade desde o início.

V-model

O "V-model" é um modelo de desenvolvimento que demonstra a relação entre cada fase do ciclo de vida do desenvolvimento de software e sua fase de teste correspondente. Ele enfatiza a verificação e validação em cada etapa:

- As fases de desenvolvimento são emparelhadas com fases de teste equivalentes, como análise/definição de requisitos, design, codificação, testes de unidade, testes de integração, etc.

Critérios de Aceitação da História (User Story) e Teste Agile

- **Critérios de Aceitação:** Condições que devem ser atendidas para que uma história de usuário seja considerada completa.
- **Teste Agile:** Testes automatizados escritos com base nos critérios de aceitação para verificar se a história foi implementada corretamente.

Especificações Executáveis

- **Sentido da Afirmação:** Especificações que podem ser executadas automaticamente como testes, garantindo que o software funcione conforme esperado.

Necessidade de Testes "Developer Facing" e "Customer Facing"

- **Developer Facing:** Testes destinados aos desenvolvedores para garantir que o código seja robusto, eficiente e de fácil manutenção.
- **Customer Facing:** Testes voltados para os usuários finais para garantir que o software atenda às suas necessidades e expectativas.

Integração contínua/Entrega Contínua

- Identificar os passos típicos de um ciclo de CI
- Distinguir entre C. Integration, C. Deployment e C. Delivery

- Relacionar o CI/CD com a natureza iterativa e incremental dos métodos ágeis de desenvolvimento, ou seja, como é que o CI/CD ajuda na concretização de metodologias incrementais? ● Explicar as tarefas e *outcomes* englobados numa "*build*" ● Explique o sentido da prática "*continuous testing*".

Passos Típicos de um Ciclo de CI (Integração Contínua)

1. **Commit:** Desenvolvedores fazem commit de seu código no repositório compartilhado.
2. **Build:** O sistema de CI automatizado realiza a compilação do código-fonte.
3. **Testes Automatizados:** Execução de testes automatizados para verificar a integridade do código.
4. **Análise Estática de Código:** Verificação de conformidade com padrões de codificação.
5. **Implantação em Ambientes de Teste:** Implantação automatizada em ambientes de teste para validar o comportamento.
6. **Feedback:** Relatório de resultados enviados aos desenvolvedores.

Diferença entre CI (Continuous Integration), CD (Continuous Deployment) e Continuous Delivery (Continuous Deployment)

- **Continuous Integration (CI):** Prática de integrar código de várias fontes em um repositório compartilhado frequentemente, com testes automatizados para validar cada integração.
- **Continuous Deployment (CD):** Processo automatizado de implantação de software em ambientes de produção após passar pelos testes automatizados e aprovações necessárias.
- **Continuous Delivery (CD):** Abordagem onde o software é sempre pronto para ser liberado a qualquer momento, mas a decisão de implantação em produção é feita manualmente.

Relação do CI/CD com Métodos Ágeis

- **Natureza Iterativa e Incremental:** Métodos ágeis enfatizam entregas frequentes e incrementais de software funcional. O CI/CD suporta essa abordagem ao automatizar processos de integração, teste e implantação, permitindo entregas rápidas e confiáveis em pequenos incrementos.

Tarefas e Outcomes de uma "Build"

- **Tarefas:** Compilação do código-fonte, execução de testes automatizados, análise estática de código, geração de artefatos de implantação.
- **Outcomes:** Identificação de erros de integração, falhas nos testes, violações de padrões de codificação, artefatos prontos para implantação.

Sentido da Prática "Continuous Testing"

- **Continuous Testing:** Prática de automatizar testes em todas as fases do ciclo de vida de desenvolvimento de software, desde a integração até a entrega. Isso garante que cada mudança no código seja testada rapidamente para identificar problemas o mais cedo possível, melhorando a qualidade do software e permitindo decisões informadas sobre a liberação.

E) Práticas dos métodos ágeis

Principais características dos métodos ágeis de desenvolvimento

- Discuta o argumento: “A abordagem em cascata tende a mascarar os riscos reais de um projeto até que seja tarde demais para fazer algo significativo sobre eles.”
- Explique o sentido da frase: “O desenvolvimento iterativo centra-se em ciclos curtos e orientados para a geração de valor”
- Discuta o argumento: “A abordagem ágil dispensa o planeamento do projeto”.
- Identifique vantagens de estruturar um projeto em iterações, produzindo incrementos funcionais com frequência.
- Caracterizar os princípios da gestão do *backlog* em projetos ágeis.
- Dado um “princípio” (do *Agile Manifest*), explicá-lo por palavras próprias, destacando a sua novidade (com relação às abordagens “clássicas”) e impacto / benefício.
- Apresentar situações em que, de facto, um método sequencial pode ser o mais adequado.
- Quais os objetivos das organizações com a adoção de metodologias de desenvolvimento “iterativas e incrementais”?

Mascarar os Riscos na Abordagem em Cascata

- Na abordagem em cascata, os requisitos são definidos no início e mudanças são difíceis de serem incorporadas em estágios avançados. Isso pode levar à descoberta de riscos significativos apenas quando o projeto está em fases avançadas, tornando tarde demais para mitigá-los eficazmente.

Desenvolvimento Iterativo e Geração de Valor

- O desenvolvimento iterativo foca em ciclos curtos e repetitivos de desenvolvimento, onde cada iteração produz um incremento funcional do software. Isso permite que o valor seja entregue ao cliente mais rapidamente e possibilita adaptações às mudanças de requisitos e feedbacks mais cedo no processo de desenvolvimento.

Dispensa de Planejamento no Agile?

- Embora os métodos ágeis valorizem mais a resposta a mudanças do que o seguimento rigoroso de um plano inicial, eles não dispensam o planejamento. O Agile promove um planejamento adaptativo e contínuo, ajustando o plano conforme novas informações e necessidades emergem ao longo do projeto.

Vantagens de Estruturar um Projeto em Iterações

- **Entrega Rápida de Valor:** Clientes recebem funcionalidades úteis mais cedo.
- **Feedback Contínuo:** Oportunidade de ajustar o produto com base no feedback do cliente.
- **Maior Flexibilidade:** Adaptar-se a mudanças nos requisitos de forma mais eficaz.
- **Redução de Riscos:** Identificação precoce de problemas e correção rápida.

Princípios da Gestão do Backlog em Projetos Ágeis

- **Priorização Constante:** Itens do backlog são priorizados continuamente com base no valor para o cliente.

- **Transparência:** O backlog é visível para todos os membros da equipe, promovendo entendimento e colaboração.
- **Adaptação:** O backlog é ajustado conforme novos requisitos e mudanças emergem.

Impacto dos Princípios do Agile Manifesto

- **Exemplo: "Responder a mudanças mais do que seguir um plano"**
  - **Novidade:** Prioriza a capacidade de adaptar-se às mudanças, em vez de seguir um plano inicial rígido.
  - **Impacto/Benefício:** Permite que equipes respondam rapidamente a novos requisitos e feedbacks, melhorando a relevância e a qualidade do produto final.

Situações em que um Método Sequencial é Adequado

- Projetos com requisitos bem definidos e estáveis desde o início.
- Projetos com prazos e orçamentos muito restritos.
- Projetos onde a tecnologia e os requisitos são conhecidos e estáveis.

Objetivos das Organizações com Metodologias Iterativas e Incrementais

- **Entrega Contínua de Valor:** Fornecer rapidamente produtos ou funcionalidades úteis ao cliente.
- **Adaptação às Mudanças:** Capacidade de responder eficazmente a novos requisitos e mudanças de mercado.
- **Maior Satisfação do Cliente:** Aproximar o produto final das expectativas e necessidades do cliente através de feedback contínuo.

Histórias (=*user stories*) e métodos ágeis

- Defina histórias (*user stories* - *US*) e dê exemplos.
- Como é que o conceito de US se relaciona [ou não] com o de requisito (da análise)?
- Compare histórias e casos de utilização em relação a pontos comuns e diferenças. Em que medida podem ser usados de forma complementar?
- Compare "Persona" com Ator com respeito a semelhanças e diferenças. Qual a utilidade das Personas no desenvolvimento das US?
- O que é a pontuação de uma história e como é que é determinada?
- Descreva o conceito de velocidade da equipa (como usado no PivotalTracker e SCRUM).
- Explique a abordagem proposta por Jacobson em "Use Cases 2.0" para combinar a técnica dos *Use cases* com a flexibilidade das *user stories*.
- Discuta se os casos de utilização e as histórias são abordagens redundantes ou complementares (quando seguir cada uma das abordagens? Em que condições? ...) ●

  Exemplifique a definição de critérios de aceitação para uma US.
- Como é que um *story map* organiza espacialmente o *backlog*, ou seja, como se usa o *story map* no contexto dos métodos ágeis?

Definição de Histórias (User Stories)

- **Definição:** Histórias (user stories) são descrições curtas e simples de uma funcionalidade desejada do ponto de vista do usuário. Elas são utilizadas nos métodos ágeis para capturar requisitos de forma leve e acessível, focando no valor entregue ao cliente.
- **Exemplos:**
    - **Exemplo 1:** Como um usuário, eu quero poder fazer login no sistema usando minha conta do Google, para facilitar o acesso.
    - **Exemplo 2:** Como um administrador do sistema, eu quero ter a capacidade de gerenciar perfis de usuário, para ajustar as permissões conforme necessário.

Relação entre Histórias e Requisitos de Análise

- **Relação:** Histórias são uma forma de requisitos de software, mas são mais centradas no valor entregue ao usuário do que em detalhes técnicos ou de implementação. Elas são complementares aos requisitos de análise, que são mais detalhados e técnicos.

Comparação entre Histórias e Casos de Utilização

- **Pontos Comuns:**
    - Ambos descrevem funcionalidades ou requisitos de software do ponto de vista do usuário.
    - Ambos são utilizados para capturar requisitos funcionais.
- **Diferenças:**
    - Histórias são mais sucintas e informais, enquanto casos de uso são mais estruturados e detalhados.
    - Casos de uso podem abranger uma sequência mais detalhada de interações, enquanto histórias são frequentemente descritas em uma única sentença.
- **Complementaridade:**
    - Podem ser usados de forma complementar: histórias podem ser usadas inicialmente para capturar os requisitos de alto nível e casos de uso podem ser elaborados posteriormente para detalhar interações mais específicas ou complexas.

Persona vs. Ator

- **Semelhanças:** Ambos representam perfis de usuários ou entidades externas que interagem com o sistema.
- **Diferenças:** Personas são representações mais detalhadas e humanizadas dos usuários, enquanto atores nos casos de uso são entidades mais genéricas que interagem com o sistema.
- **Utilidade das Personas nas Histórias:** Personas ajudam a manter o foco nas necessidades dos usuários durante o desenvolvimento das histórias, garantindo que as funcionalidades entregues sejam relevantes e úteis.

Pontuação de uma História

- **Definição:** A pontuação de uma história (story points) é uma unidade de medida relativa usada para estimar a complexidade, o esforço ou o tamanho de uma história.
- **Determinação:** É determinada pela equipe ágil durante a sessão de planejamento da sprint, usando técnicas como Planning Poker, onde cada membro atribui pontos com base na complexidade percebida da história.

Velocidade da Equipe (Team Velocity)

- **Definição:** Velocidade da equipe é a medida da quantidade de trabalho que uma equipe ágil pode completar em uma iteração (sprint).

- **Utilização no SCRUM e PivotalTracker:** No SCRUM e em ferramentas como o PivotalTracker, a velocidade da equipe ajuda a prever quantas histórias podem ser completadas em sprints futuros, facilitando o planejamento.

Abordagem de Jacobson em "Use Cases 2.0"

- **Combinação de Técnicas:** Jacobson propõe combinar a estrutura e formalidade dos casos de uso com a flexibilidade e foco nas necessidades do usuário das histórias.
- **Benefícios:** Isso permite capturar requisitos de forma estruturada (casos de uso) e também de maneira ágil e adaptável (histórias), atendendo a diferentes necessidades de documentação e especificação ao longo do ciclo de vida do projeto.

Histórias e Casos de Uso: Redundância ou Complementaridade?

- **Quando Seguir Cada Abordagem:**
  - Use histórias para capturar requisitos de alto nível de forma rápida e acessível.
  - Use casos de uso para detalhar interações mais complexas e sequências de eventos detalhadas.
  - Ambos podem ser usados de forma complementar, adaptando-se às necessidades específicas do projeto e da equipe.

Definição de Critérios de Aceitação para uma História

- **Definição:** Critérios de aceitação são condições que uma história deve atender para ser considerada completa e correta.
- **Exemplo:** Para a história "Como um usuário, eu quero poder fazer login no sistema usando minha conta do Google":
  - **Critérios de Aceitação:** Deve ser possível fazer login usando conta do Google; Deve ser fácil de configurar a conta do Google no sistema; Deve ser seguro e seguir práticas de segurança recomendadas.

Utilização do Story Map no Contexto Ágil

- **Organização Espacial do Backlog:** O story map organiza o backlog em um mapa visual que mostra as funcionalidades do sistema de acordo com sua prioridade e fluxo de uso pelo usuário.
- **Uso no Contexto Ágil:** Permite uma visão clara das funcionalidades a serem desenvolvidas, ajudando na priorização e no planejamento das entregas ao longo das iterações (sprints).

O *framework* SCRUM de gestão de equipas

- Explique o objetivo da “Daily Scrum meeting”
- Relacione os conceitos de *sprint* e iteração e discuta a sua duração esperada.
- Explique a método de pontuação das histórias (e critérios aplicados)
- Identificar os papéis numa equipa de Scrum e as principais "cerimónias”
- Relacione as práticas previstas no SCRUM e os princípios do “Agile Manifest”: em que medida estão alinhados?
- Pode-se considerar o SCRUM como a “*silver bullet*” (solução universal) para o sucesso dos projetos de desenvolvimento?
- Identifique alguns desafios/bloqueios à implementação eficaz do SCRUM.

Objetivo da "Daily Scrum Meeting"

- **Objetivo:** A reunião diária do SCRUM, ou Daily Scrum, tem como objetivo principal sincronizar a equipe ágil, garantindo que todos os membros estejam alinhados quanto ao progresso do trabalho, identifiquem impedimentos e ajustem o plano para atingir as metas da sprint.
- **Formato:** Realizada diariamente, geralmente no início do dia de trabalho, com duração curta (cerca de 15 minutos), onde cada membro da equipe responde três perguntas: 1) O que foi feito desde a última Daily Scrum? 2) O que será feito até a próxima? 3) Quais são os impedimentos?

Relação entre Sprint e Iteração

- **Sprint e Iteração:** No contexto do SCRUM, sprint e iteração são termos frequentemente utilizados de forma intercambiável para descrever o ciclo de desenvolvimento de software, onde um incremento de trabalho é produzido.
- **Duração Esperada:** A duração de uma sprint típica no SCRUM varia de 1 a 4 semanas, sendo 2 semanas uma duração comum. A iteração refere-se ao ciclo repetitivo de desenvolvimento dentro de uma sprint, focando na entrega contínua de valor ao cliente.

Método de Pontuação das Histórias

- **Pontuação das Histórias:** As histórias são pontuadas em termos de complexidade ou esforço relativo usando story points, uma unidade de medida relativa. A equipe usa técnicas como Planning Poker para atribuir pontos a cada história, considerando fatores como esforço, complexidade, incerteza e riscos envolvidos.
- **Critérios Aplicados:** Os critérios incluem o entendimento comum da história pela equipe, o esforço necessário para implementá-la e a clareza dos critérios de aceitação que definem quando a história está completa.

Papéis e Cerimônias no SCRUM

- **Papéis:** Os principais papéis no SCRUM são: Product Owner, Scrum Master e Equipe de Desenvolvimento.
    - **Product Owner:** Responsável por maximizar o valor do produto e gerenciar o backlog do produto.
    - **Scrum Master:** Responsável por garantir que a equipe adere aos princípios e práticas do SCRUM.
    - **Equipe de Desenvolvimento:** Responsável por desenvolver o produto.
- **Cerimônias:** As principais cerimônias incluem: Sprint Planning (planejamento da sprint), Daily Scrum (reunião diária), Sprint Review (revisão da sprint) e Sprint Retrospective (retrospectiva da sprint).

Alinhamento entre Práticas do SCRUM e Princípios do Agile Manifesto

- **Alinhamento:** As práticas do SCRUM estão alinhadas com os princípios do Manifesto Ágil, como priorizar indivíduos e interações sobre processos e ferramentas, entrega contínua de software funcional, colaboração com o cliente e responder a mudanças.
- **Exemplo:** O SCRUM enfatiza iterações curtas e feedback contínuo, o que está alinhado com o princípio de responder a mudanças ao invés de seguir um plano rígido.

SCRUM como "Silver Bullet"

- **Consideração:** O SCRUM não pode ser considerado uma solução universal ("silver bullet") para todos os projetos de desenvolvimento. Embora seja eficaz para muitos contextos, seu sucesso depende da cultura organizacional, do tipo de projeto e da maturidade da equipe.
- **Limitações:** Projetos complexos ou altamente regulamentados podem necessitar de adaptações no SCRUM ou mesmo de abordagens diferentes para garantir o sucesso.

Desafios à Implementação Efetiva do SCRUM

- **Desafios:** Alguns desafios comuns incluem resistência à mudança organizacional, falta de comprometimento da equipe, dificuldades na definição e gestão do backlog, além de dificuldades em integrar o SCRUM com processos existentes na organização.
- **Soluções:** Superar esses desafios requer educação e treinamento adequados, suporte contínuo da alta administração, adaptação do SCRUM conforme necessário e uma cultura organizacional que promova colaboração e transparência.