Resumo Turbinado para Prova - Engenharia e Requisitos de Software (Aulas 1, 2, 3, 4, 7)

Aula 1: O Básico da Engenharia de Software (ES)

- **O que é ES?** É aplicar princípios de engenharia para criar e manter software de forma econômica, confiável e eficiente. Não é só programar, é o processo *todo*.
- **Por que ES?** Para lidar com a "Crise do Software" (projetos atrasados, estourando orçamento, baixa qualidade, difíceis de manter).
- **ES vs Ciência da Computação:** ES foca na *prática* de construir software; CC foca mais na *teoria* e fundamentos.
- **Bom Software tem:** Manutenibilidade (fácil de corrigir/melhorar), Confiabilidade (funciona como esperado), Eficiência (usa bem os recursos), Usabilidade (fácil de usar), etc.
- **Processo de Software (Visão Geral):**
    1. **Especificação:** Definir o que o sistema deve fazer.
    2. **Projeto e Implementação:** Desenhar e codificar o software.
    3. **Validação:** Garantir que o software atende ao que foi pedido (testes!).
    4. **Evolução:** Modificar o software para novas necessidades ou correções.
- **Ética:** Profissionais de ES têm responsabilidades (confidencialidade, competência, etc.).

Aula 2: Processos de Software (Como fazer o software)

- **Modelo de Processo:** Um "mapa" de como conduzir o desenvolvimento.
- **Modelos Principais:**
    - **Cascata (Waterfall):** Sequencial (uma fase após a outra). Simples, mas ruim para requisitos que mudam. Funciona melhor quando tudo é bem definido desde o início.
    - **Prototipagem Evolucionária:** Cria um protótipo inicial e vai refinando com feedback do cliente. Bom para requisitos incertos. O protótipo *vira* o sistema final.
    - **Prototipagem de Requisitos (Throwaway):** Cria um protótipo só para entender/validar requisitos, depois joga fora e começa "do zero" com um modelo mais robusto (como cascata ou incremental).
    - **Incremental:** Entrega o software em partes funcionais (incrementos). Cliente vê valor mais cedo. Mais flexível a mudanças. (Ex: RUP pode ser visto como incremental/iterativo).
    - **Espiral:** Foca na análise e gestão de *riscos* em cada ciclo. Bom para projetos grandes e arriscados. Complexo de gerenciar.
    - **Ágil:** Prioriza indivíduos e interações, software funcional, colaboração com cliente e resposta a mudanças *acima* de processos rígidos e documentação excessiva (Manifesto Ágil!). Exemplos: Scrum, XP.
- **Qual escolher?** Depende do projeto (tamanho, complexidade, clareza dos requisitos, criticidade, time).

Aula 3: Engenharia de Requisitos (O que o sistema DEVE fazer - Parte 1)

- **Requisito:** Uma descrição do que o sistema deve fazer (serviço) ou uma restrição sobre ele. É a base de TUDO. Errar aqui custa caro!
- **Tipos de Requisitos:**
    - **Funcionais (RF):** O que o sistema *faz*. Ex: "O sistema deve permitir cadastrar cliente".
    - **Não-Funcionais (RNF):** Como o sistema faz, qualidades, restrições. Ex: "O sistema deve responder em até 2 segundos" (desempenho), "Deve rodar em Linux" (restrição), "Deve ser fácil de usar" (usabilidade), "Apenas admins podem excluir dados" (segurança).
    - **De Domínio:** Vêm do "mundo" onde o sistema vai operar. Ex: Leis trabalhistas, fórmulas físicas, regras de negócio específicas.
- **Níveis:**
    - **Requisitos de Usuário:** Alto nível, linguagem do cliente.
    - **Requisitos de Sistema:** Detalhados, técnicos, para os desenvolvedores. Base para o projeto.
- **Processo de Eng. Requisitos:** Elicitação -> Análise -> Especificação -> Validação -> Gerenciamento (é iterativo!).
- **Stakeholders:** Qualquer pessoa/grupo afetado pelo sistema (clientes, usuários, gerentes, desenvolvedores, etc.). Identificá-los é crucial.
- **Elicitação (Levantamento):** Descobrir os requisitos. Técnicas:
    - Entrevistas (abertas ou fechadas).
    - Questionários.
    - Observação (ver o usuário trabalhando - Etnografia).
    - Workshops (reunir stakeholders - JAD).
    - Análise de Documentos existentes.
    - Prototipagem (para visualizar e obter feedback).
    - Cenários (descrever interações específicas).
    - **Casos de Uso (Use Cases):** Descrevem interações entre ator e sistema para atingir um objetivo. (Mais detalhes na Aula 7).

Aula 4: Engenharia de Requisitos (O que o sistema DEVE fazer - Parte 2)

- **Análise de Requisitos:** Entender, organizar, classificar, negociar e priorizar os requisitos levantados. Resolver conflitos. Modelagem inicial pode ajudar.
- **Especificação de Requisitos:** Documentar os requisitos de forma clara e precisa. O documento principal é a **SRS (Software Requirements Specification)** ou ERS (Especificação de Requisitos de Software).
- **Boa SRS é:** Correta, Completa, Consistente (sem contradições), Clara (sem ambiguidade), Verificável (dá pra testar?), Priorizada, Rastreável (de onde veio? para onde vai no projeto?), Modificável.
- **Estrutura da SRS (Exemplo):** Introdução, Descrição Geral (visão do produto, funções, usuários, restrições), Requisitos Específicos (Funcionais, Não-Funcionais, Interfaces Externas, Banco de Dados), Apêndices, Glossário.
- **Validação de Requisitos:** Checar se a SRS descreve o sistema que o cliente *realmente* quer e se está boa (completa, consistente, etc.). **É diferente de validar o *software* (testes)!**

- **Técnicas de Validação:** Revisões/Inspeções (ler e discutir), Prototipagem (mostrar algo pro cliente), Geração de Casos de Teste (se dá pra testar, está mais claro), Checklists.
- **Gerenciamento de Requisitos:** Lidar com as *mudanças* nos requisitos ao longo do projeto (que *vão* acontecer!). Inclui controle de versão, análise de impacto e **rastreabilidade** (conectar requisitos a outros artefatos - design, código, testes). Matriz de Rastreabilidade ajuda nisso.

Aula 7: Modelagem com UML (Foco em Casos de Uso e Classes)

- **UML (Unified Modeling Language):** Linguagem gráfica padrão para *visualizar, especificar, construir e documentar* sistemas. Ajuda a entender e comunicar.
- **Diagrama de Casos de Uso (Use Case Diagram):**
    - **Para quê?** Mostrar as *funcionalidades* do sistema do ponto de vista de quem interage com ele. O "o quê" o sistema faz.
    - **Quem usa?** Analistas, clientes, desenvolvedores (para entender o escopo).
    - **Elementos:**
        - **Ator (Actor):** Papel que interage com o sistema (pessoa, outro sistema). Bonequinho palito.
        - **Caso de Uso (Use Case):** Uma funcionalidade/objetivo que o ator realiza com o sistema. Oval.
        - **Relacionamentos:**
            - *Associação:* Linha simples entre Ator e Caso de Uso (quem faz o quê).
            - *<<Include>>:* Um caso de uso *obrigatoriamente* inclui outro (reuso). Seta tracejada apontando para o incluído.
            - *<<Extend>>:* Um caso de uso *pode* estender o comportamento de outro (opcional, condicional). Seta tracejada apontando para o estendido.
            - *Generalização:* Entre Atores (um ator é tipo específico de outro) ou Casos de Uso (raro). Seta com ponta triangular vazada.
    - **Especificação de Caso de Uso (Textual):** Detalha *um* caso de uso (Nome, Atores, Pré-condições, Pós-condições, Fluxo Principal, Fluxos Alternativos/Exceção). Essencial!
- **Diagrama de Classes (Class Diagram):**
    - **Para quê?** Mostrar a *estrutura estática* do sistema: as classes, seus atributos, métodos e como se relacionam. O "esqueleto" do código.
    - **Quem usa?** Analistas, designers, desenvolvedores.
    - **Elementos:**
        - **Classe:** Retângulo com Nome, Atributos (dados) e Operações (métodos/comportamentos).
        - **Atributo:** `visibilidade nome : tipo = valor_inicial` (ex: `- nome : String`)
        - **Operação:** `visibilidade nome(parametros) : tipo_retorno` (ex: `+ calcularSalario() : float`)
        - **Visibilidade:** **+** (público), **-** (privado), # (protegido).

        - **Relacionamentos:**
            - *Associação:* Conexão geral entre classes. Linha simples. Pode ter nome, papel e multiplicidade.
            - *Agregação:* Relação "tem um" ou "todo-parte", onde a parte *pode* existir sem o todo. Losango vazado no lado do "todo".
            - *Composição:* Relação "parte de", onde a parte *não pode* existir sem o todo (forte dependência). Losango preenchido no lado do "todo".
            - *Herança/Generalização:* Relação "é um tipo de". Uma classe (subclasse) herda atributos/operações de outra (superclasse). Seta com ponta triangular vazada apontando para a superclasse.
        - **Multiplicidade:** Indica quantos objetos de uma classe podem se relacionar com *um* objeto da outra (nos relacionamentos). Ex: `1` (exatamente um), `0..1` (zero ou um), `*` (zero ou mais), `1..*` (um ou mais).

Parte 2: Resumo Consolidado (Estilo Pressman - Aulas 1, 2, 3, 4, 7)

Este resumo visa cobrir os pontos essenciais dos tópicos das aulas indicadas, com a profundidade e a terminologia encontradas no livro do Pressman, para que você tenha uma visão completa para a prova.

Engenharia de Software: Fundamentos e Processo (Aula 1 e 2)

- **Definição e Camadas:** Engenharia de Software (ES) é a aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção de software.[1] Pressman a vê como uma tecnologia em camadas:
    - **Foco na Qualidade (Base):** Fundamento que permeia todo o processo.
    - **Processo:** A cola que une as camadas; define a estrutura e o fluxo das atividades.
    - **Métodos:** Indicam o "como fazer" técnico (análise, design, codificação, teste).
    - **Ferramentas:** Suporte automatizado ou semi-automatizado para métodos e processo (CASE tools).
    -
- **Processo de Software:** É a estrutura para as atividades, ações e tarefas necessárias para construir software de alta qualidade.
    - **Estrutura Genérica:**
        - **Comunicação:** Intensa colaboração com stakeholders para iniciar o projeto e levantar requisitos.
        - **Planejamento:** Criação de um "mapa" – estimativas, cronograma, riscos.
        - **Modelagem:** Criação de modelos para entender melhor os requisitos (Análise) e definir a solução (Design). *UML é chave aqui.*
        - **Construção:** Geração de código e testes (unitários, integração).

        - **Implantação (Deployment):** Entrega do software (ou incremento) ao cliente, avaliação e feedback.
    - **Atividades Guarda-Chuva:** Ocorrem ao longo de todo o processo: Gerenciamento de Riscos, Garantia da Qualidade (SQA), Gerenciamento da Configuração de Software (SCM), Medição, Gerenciamento de Projeto, etc.
- **Modelos de Processo Prescritivos:** Tentam impor ordem e estrutura.
    - **Cascata (Waterfall):** Sequencial clássico. Bom para requisitos estáveis, ruim para mudanças.
    - **Incremental:** Entrega operacional em "incrementos". Requisitos são definidos e evoluem a cada incremento. Permite feedback mais cedo.
    - **Espiral:** Iterativo, focado em análise de riscos a cada ciclo. Combina desenvolvimento iterativo com aspectos do cascata. Bom para projetos grandes e arriscados.
    - *(Outros mencionados por Pressman: V-Model, Component-Based Development)*
- **Modelos de Processo Ágeis:** Enfatizam agilidade, resposta a mudanças e entrega rápida.
    - **Manifesto Ágil:** Valoriza Indivíduos e Interações, Software em Funcionamento, Colaboração com Cliente, Responder a Mudanças.
    - **Princípios Ágeis:** Satisfação do cliente, aceitar mudanças, entregas frequentes, colaboração, etc.
    - **Exemplos:**
        - **Scrum:** Framework com papéis (PO, SM, Dev Team), cerimônias (Sprint Planning, Daily, Review, Retrospective) e artefatos (Product Backlog, Sprint Backlog, Increment). Foco em gestão e auto-organização.
        - **Extreme Programming (XP):** Foco em práticas técnicas de engenharia (Pair Programming, TDD, Refactoring, Integração Contínua, Small Releases).

Engenharia de Requisitos (Aulas 3 e 4)

- **Objetivo:** Entender o que o cliente quer, analisar necessidades, avaliar viabilidade, negociar solução, especificar de forma não ambígua, validar e gerenciar os requisitos. É a base para a qualidade.
- **Tarefas Principais:**
    - **Concepção (Inception):** Entendimento básico do problema e escopo.
    - **Elicitação:** Levantar requisitos dos stakeholders. Técnicas: Entrevistas, Facilitated Application Specification Techniques (FAST - similar a JAD/Workshops), Quality Function Deployment (QFD), Cenários/Casos de Uso, Prototipagem.
    - **Elaboração (Análise):** Expandir o entendimento, refinar requisitos, modelagem inicial (foco no domínio do problema), negociar prioridades e resolver conflitos.
    - **Negociação:** Conciliar conflitos entre requisitos ou entre stakeholders e desenvolvedores (custo, prazo, etc.).
    - **Especificação:** Criar um documento formal (SRS - Software Requirements Specification) ou modelos equivalentes (ex: backlog ágil detalhado) que

descrevam o que o sistema deve fazer. Pode incluir modelos UML (Casos de Uso, Classes de Análise), protótipos, especificações formais.
    - **Validação:** Assegurar que os requisitos especificados estão corretos, completos, consistentes e realmente representam o que o cliente precisa. Técnicas: Revisões Técnicas Formais, Prototipagem, Desenvolvimento de Casos de Teste.
    - **Gerenciamento de Requisitos:** Lidar com mudanças inevitáveis de forma controlada, mantendo a rastreabilidade (conexão entre requisitos e outros artefatos do projeto – design, código, testes).
- **Tipos de Requisitos:**
    - **Funcionais (RF):** Descrevem as funções/serviços que o software deve executar.
    - **Não-Funcionais (RNF):** Descrevem atributos de qualidade (desempenho, segurança, usabilidade, confiabilidade) ou restrições (tecnológicas, legais, de interface).
    - **Restrições (Constraints):** Limitações impostas ao desenvolvimento ou à solução (ex: deve usar SGBD Oracle, deve rodar em Windows).
- **SRS (Especificação de Requisitos de Software):** Documento (ou conjunto de modelos/artefatos) que serve de contrato. Deve ser: Correto, Completo, Consistente, Não-Ambíguo, Verificável, Priorizado, Rastreável, Modificável. Sua estrutura varia, mas geralmente inclui introdução, descrição geral, requisitos específicos (RF, RNF, Interfaces) e apêndices.

Modelagem com UML (Foco Aula 7: Casos de Uso e Classes)

- **Modelagem de Análise:** Foco em entender os requisitos e o domínio do problema. Responde "O quê?".
    - **Diagrama de Casos de Uso (Use Case Diagram):**
        - **Propósito:** Descrever a funcionalidade do sistema sob a perspectiva do usuário (ator), mostrando as interações para alcançar objetivos. Essencial na elicitação e especificação.
        - **Elementos:** Atores (papéis externos), Casos de Uso (funcionalidades), Relacionamentos (Associação ator-caso de uso; para reuso obrigatório; para comportamento opcional/condicional; Generalização entre atores ou casos de uso).
        - **Especificação Textual:** Complementa o diagrama detalhando o fluxo de eventos (principal, alternativos, exceções), pré/pós-condições. Crucial para o entendimento completo.
- **Modelagem de Projeto (Introdução via Diagrama de Classes):** Foco em definir a solução técnica. Responde "Como?".
    - **Diagrama de Classes (Class Diagram):**
        - **Propósito:** Descrever a estrutura estática do sistema em termos de classes, atributos, operações e relacionamentos. Base para a implementação orientada a objetos.
        - **Elementos:**

        - **Classe:** Nome, Atributos (dados, com), Operações (métodos, com).
        - **Visibilidade:** (public), (private), (protected), (package/default).
        - **Relacionamentos Estruturais:**
            - **Associação:** Relação geral entre classes (pode ter navegabilidade, nome, papéis, multiplicidade).
            - **Agregação:** Relação "todo-parte" onde a parte *pode* existir independentemente do todo (losango vazio). Ex: Departamento e Professor.
            - **Composição:** Relação "todo-parte" forte onde a parte *não* existe sem o todo (dependência de ciclo de vida, losango preenchido). Ex: Pedido e ItemPedido.
            - **Herança/Generalização:** Relação "é um tipo de" (subclasse herda da superclasse, seta com triângulo vazio).
        - **(Outros Relacionamentos Comuns):** Dependência (, tracejada), Realização (interface-classe, tracejada com triângulo vazio).
    - **Multiplicidade:** Quantos objetos de uma classe se associam a *um* objeto da outra (, , , ,).

Possíveis Perguntas "Difíceis" ou Práticas (Baseadas nas Aulas 1, 2, 3, 4, 7)

Sobre Processos de Software (Aula 2):

1. **Escolha de Modelo:** "Uma empresa quer desenvolver um sistema para controlar um equipamento médico de alto risco. Os requisitos são muito bem definidos e a segurança é a prioridade máxima. Qual modelo de processo seria mais adequado e por quê? Justifique considerando as características do projeto."
    - *Como pensar:* Qual modelo lida melhor com requisitos estáveis e foco em qualidade/segurança?
    - **Resposta Curta Correta:** Modelo Cascata, pois os requisitos são estáveis e a segurança/qualidade exige verificação rigorosa em cada fase bem definida.
2. **Comparação Ágil vs. Tradicional:** "Seu time vai desenvolver um app de rede social, onde as funcionalidades podem mudar com frequência com base no feedback dos primeiros usuários. Seria melhor usar o Modelo Cascata ou uma abordagem Ágil? Por quê? Quais os benefícios e desvantagens de cada um *neste cenário*?"
    - *Como pensar:* Compare flexibilidade, resposta a mudanças, entrega de valor.

    - **Resposta Curta Correta:** Abordagem Ágil (ex: Scrum), pois permite adaptação a mudanças frequentes e feedback rápido do usuário, essencial para um app social onde requisitos evoluem.
3. **Riscos do Modelo:** "Quais são os principais riscos de usar o modelo de Prototipagem Evolucionária? Como eles poderiam ser mitigados?"

    - *Como pensar:* Riscos incluem: sistema final pode ter estrutura ruim, dificuldade em gerenciar, requisitos podem nunca estabilizar.
    - **Resposta Curta Correta:** Riscos: arquitetura final ruim ('otimizada' para o protótipo), 'scope creep' (requisitos mudando sem parar). Mitigação: planejar refatorações, definir critérios de aceite claros e usar time-boxing.

Sobre Engenharia de Requisitos (Aulas 3 e 4):

4. **Identificação e Classificação:** "Dado o cenário: 'Um sistema para uma locadora de filmes online [...] Todos os dados de pagamento devem ser criptografados usando AES-256.' Identifique 2 Requisitos Funcionais, 2 Não-Funcionais e 1 de Domínio (se houver)."

    - *Como pensar:* RF (o que faz), RNF (como faz/qualidade/restrição), Domínio (regras do 'mundo real').
    - **Resposta Curta Correta:** RF: Buscar filme, Baixar filme (premium). RNF: Exibir capa < 1s, Disponibilidade 24/7 (com exceção), Criptografia AES-256 (segurança). Domínio: (Nenhum explícito, mas poderia ser algo como 'Respeitar classificação etária').
5. **Qualidade do Requisito:** "O requisito 'O sistema deve ter uma interface bonita' é um bom requisito? Justifique usando as características de um bom requisito (da Aula 4). Como você poderia reescrevê-lo para ser melhor?"

    - *Como pensar:* É subjetivo, ambíguo, não verificável? Como torná-lo objetivo e testável?
    - **Resposta Curta Correta:** Não é bom (ambíguo, subjetivo, não verificável). Melhorar: definir critérios objetivos (ex: seguir guia de estilo X da empresa, ou métricas de usabilidade como 'tarefa Y < 5 min para 80% dos usuários').
6. **Técnica de Elicitação:** "Você precisa definir os requisitos para um sistema que será usado por um grupo muito diverso de pessoas (idade, habilidade com tecnologia, etc.) que estão geograficamente dispersas. Qual(is) técnica(s) de elicitação seria(m) mais eficaz(es) e por quê?"

    - *Como pensar:* Quais técnicas alcançam pessoas remotamente e coletam informações de perfis variados?
    - **Resposta Curta Correta:** Questionários online (alcance amplo/disperso), Entrevistas por vídeo (aprofundar com amostras diversas), Workshops virtuais. Observação/presencial inviável pela dispersão.
7. **Validação vs. Verificação:** "Qual a diferença fundamental entre Validação de Requisitos e Validação/Verificação de Software (Testes)? Por que a Validação de Requisitos é feita *antes* do desenvolvimento?"

- *Como pensar:* Validação Reqs = É o produto certo? Testes = O produto foi feito certo? Por que checar antes?
- **Resposta Curta Correta:** Validação Reqs: Checa se os requisitos definem o sistema que o cliente quer ("construir a coisa certa?"). Testes: Checa se o software atende aos requisitos ("construir a coisa certo?"). Validar requisitos antes evita o alto custo de corrigir erros após a codificação.

8. **Gerenciamento de Mudanças:** "Durante um projeto, o cliente pede uma mudança significativa em um requisito funcional já aprovado. Qual o processo ideal para lidar com essa solicitação? Por que simplesmente 'fazer a mudança' pode ser ruim?"

    - *Como pensar:* Qual o fluxo seguro para avaliar e incorporar mudanças? Quais os riscos de pular etapas?
    - **Resposta Curta Correta:** Processo: Registrar -> Analisar Impacto (custo, prazo, riscos) -> Aprovar/Rejeitar -> Atualizar Documentos -> Comunicar. Fazer direto causa estouro de orçamento/prazo, inconsistências e desalinhamento.

Sobre Modelagem UML (Aula 7):

9. **Diagrama de Casos de Uso - Relacionamentos:** "Explique com suas palavras a diferença entre e . Dê um exemplo para cada um em um contexto de Loja Virtual."

    - *Como pensar:* Include = Obrigatório/Reuso. Extend = Opcional/Condicional.
    - **Resposta Curta Correta:** Include: funcionalidade obrigatória/reusada por outra (ex: 'Validar Login' incluído em 'Finalizar Compra'). Extend: funcionalidade opcional/adicional a outra (ex: 'Adicionar Cupom' estende 'Finalizar Compra').

10. **Diagrama de Classes - Relacionamentos:** "Qual a diferença prática entre Agregação e Composição? Quando você usaria uma em vez da outra ao modelar , e ?"

    - *Como pensar:* Qual relação implica dependência de vida (parte morre com o todo)?
    - **Resposta Curta Correta:** Composição: parte NÃO existe sem o todo (dependência de vida, ex: Pedido-ItemPedido). Agregação: parte PODE existir sem o todo (ex: Departamento-Professor). Para Universidade-Departamento: Composição. Para Departamento-Professor: Agregação.

11. **Diagrama de Classes - Modelagem Prática:** "Desenhe um Diagrama de Classes simples para representar um e suas . Um carro tem exatamente 4 rodas, e uma roda pertence a apenas um carro. Qual tipo de relacionamento você usaria e por quê? Inclua um atributo no Carro e na Roda."

    - *Como pensar:* Qual relacionamento representa uma parte que não existe sem o todo? Qual a quantidade?

    - **Resposta Curta Correta:** Composição (Roda não existe sem o Carro). Multiplicidade: Carro "1" -- "4" Roda (losango preenchido no lado do Carro). Classes: Carro{-marca:String}, Roda{-aro:Integer}.
12. **Especificação de Caso de Uso:** "Por que a Especificação de Caso de Uso textual é importante, mesmo já tendo o diagrama? O que aconteceria se ela não fosse feita?"

    - *Como pensar:* O que falta no diagrama que o texto fornece? Qual o risco de não ter esses detalhes?
    - **Resposta Curta Correta:** Detalha o 'como' (fluxos principal/alternativos, regras, pré/pós-condições) que o diagrama omite. Sem ela, a implementação seria ambígua, incompleta e propensa a erros de interpretação.