NOTAS DE AULA DE ENGENHARIA DE SOFTWARE

Anielle Severo Lisbôa de Andrade

Disponível em:

SUMÁRIO

1. Engenharia de Requisitos

Seção 1: Introdução

1.1 Definição

- Descrições dos **serviços** do sistema + **restrições** operacionais. Engenharia de Requisitos (ER) = processo de **descobrir, analisar, documentar e verificar** esses serviços e restrições, Sommerville [2].

1.2 Importância

- Disciplina **crítica** para o sucesso do projeto. Falhas em ER causam atrasos, custos e insatisfação

1.3 Objetivo

- Apresentar as seções (Processo, Tipos, Tendências, Conclusão).

Seção 2: O Processo da Engenharia de Requisitos

- Processo **iterativo** e não puramente sequencial. Atividades se sobrepõem, Sommerville [2].

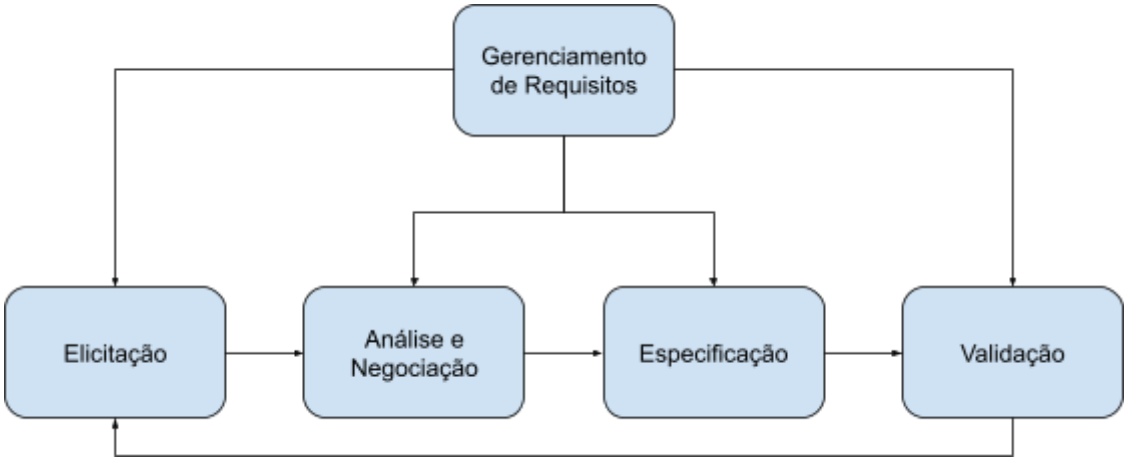


Figura 1. O Processo Iterativo da Engenharia de Requisitos
Fonte: Adaptado de Sommerville [2].

2.1 Elicitação de Requisitos

- Como a atividade de descoberta e coleta das necessidades dos stakeholders.
- Técnicas mais comuns: entrevistas, questionários, workshops e a análise de documentos

2.2 Análise e Negociação

- Verificar consistência e viabilidade. Resolver conflitos (trade-offs)..
- A negociação é um processo para chegar a um consenso entre os stakeholders.

2.3 Especificação e Documentação

- Documentar no **SRS** (*Software Requirements Specification*).
- A importância da clareza, concisão e não ambiguidade na documentação.

2.4 Validação de Requisitos

- A validação é a revisão do documento para garantir que os requisitos representem as necessidades do cliente. "Estamos construindo o produto certo?" (Revisões, protótipos).

2.5 Gerenciamento de Requisitos

- Inclui o controle de mudanças. Documentar.
- Envolve a manutenção da rastreabilidade (capacidade de traçar as relações entre requisitos, componentes de design e código)

Seção 3: Tipos de Requisitos

3.1 Requisitos de Usuário e Requisitos de Sistema

- **Usuário:** Alto nível, linguagem natural, para clientes e gerentes.
- **Sistema:** Detalhado, preciso, para desenvolvedores

3.2 Requisitos Funcionais, Não Funcionais e de Domínio

- **Funcionais:** O QUE o sistema faz. Serviços, comportamentos, reações a entradas.
- **Não Funcionais (RNF):** COMO o sistema faz. Restrições e propriedades emergentes (desempenho, segurança, confiabilidade). Podem tornar o sistema inútil se não atendidos.
- **Subdivisão dos RNF:**
    - **De Produto:** Usabilidade, eficiência, portabilidade.
    - **Organizacionais:** Padrões de processo, linguagens de programação.
    - **Externos:** Leis (privacidade), ética, interoperabilidade.
- **De Domínio:** Derivados da área de aplicação (fórmulas, regras de negócio específicas).

Seção 4: Discussão e Tendências Atuais

4.1 Impacto dos Métodos Ágeis na Engenharia de Requisitos

- A abordagem tradicional, focada na produção de um Documento de Requisitos de Software (SRS) detalhado e completo antes do início do projeto, entra em conflito com os valores ágeis de flexibilidade e resposta à mudança.
- Histórias de usuário são a principal forma de representar requisitos em metodologias ágeis, backlogs e elicitação contínua.

4.2 Automação e o Uso de Inteligência Artificial (IA)

- **Paradigma:** IA Generativa permitindo desenvolvimento a partir de **"requisitos naturais"** (linguagem, imagens, etc.), Robinson et al.[1].

- **Oportunidades:** IA como novo intermediário; escalar técnicas de IHC (Interação Humano-Computador).
- **Desafios:** Necessidade de **guiar o usuário** , capturar **conhecimento tácito** , risco de **"colapso do modelo"** (falta de diversidade nas soluções) e necessidade de criar **Datasets**.

4.3 Novos Desafios: Requisitos para Sistemas de IA

- Dificuldade em especificar requisitos funcionais para modelos de Machine Learning. Foco em RNFs: **justiça (fairness), explicabilidade, robustez**.

Seção 5: Conclusão

- **Síntese:** ER é o **alicerce** indispensável para software de qualidade.
- **Recapitulação:** O texto cobriu o processo sistemático, as classificações e as tendências evolutivas (agilidade, IA).
- **Mensagem Final:** ER é um campo **dinâmico** e continua sendo um fator **determinante** para o sucesso de projetos.

Referências

[1] ROBINSON, Diana, et al. Requirements are all you need: The final frontier for end-user software engineering. *ACM Transactions on Software Engineering and Methodology*, 2025, 34.5: 1-22.
[2] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.

2. Reuso de Software

Seção 1: Introdução

1.1 Definição

- Reuso como a prática de utilizar ativos de software existentes (como código, design ou documentação) na criação de novos sistemas, Ezran et al. [1].
- Uma definição de Arquitetura de Software, Taylor [3].
    - Aumento de produtividade, melhora na qualidade e vantagem competitiva.

1.2 Motivação

- A principal razão para o reuso: melhorar a produtividade, reduzir o tempo e o custo de desenvolvimento e aumentar a qualidade do software, Ezran et al. [1].

1.3 Objetivo

- Irá explorar os tipos de reuso, seus benefícios e desafios e o seu papel na engenharia de software moderna.

Seção 2: Fundamentos e Estratégias de Reuso

2.1 Reuso Ad-Hoc vs. Reuso Sistemático

- **Reuso Ad-Hoc vs. Sistemático**: O reuso ad hoc é informal e não planejado, gerando ganhos limitados. O reuso sistemático é uma estratégia de negócio que exige investimento em processos e arquitetura para maximizar o retorno, Ezran et al. [1].
    - Reuso oportunístico
    - Reuso planejado
    - Paradigma produtor-consumidor

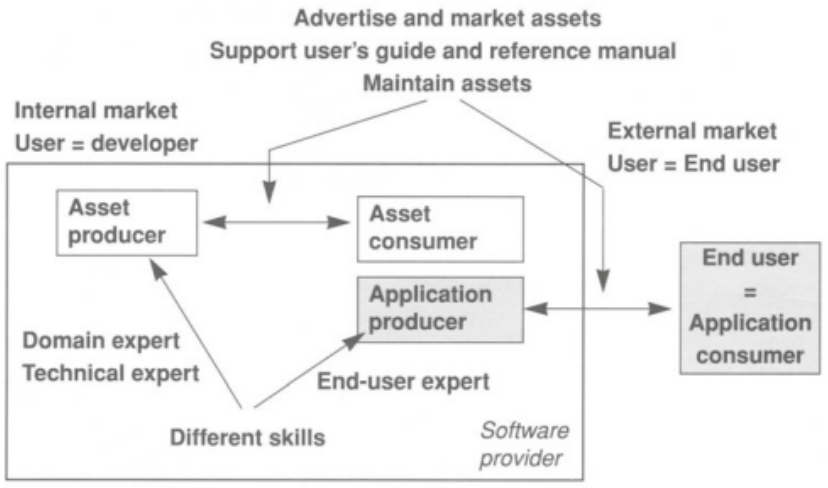


Figura 1: O paradigma produtor-consumidor.
Fonte: Adaptado de EZRAN et al. [1]

2.2 Ativos de Reuso e Linhas de Produto

Um ativo reutilizável pode ser qualquer artefato do ciclo de vida do software, não se limitando ao código. Eles variam em granularidade, de uma função simples a uma arquitetura completa, Ezran et al. [1].

- **Estratégias de reuso**
    - **Vertical**: Reutiliza funcionalidades dentro de um domínio de negócio específico (ex: setor financeiro).
    - **Horizontal**: Reutiliza funcionalidades que transcendem domínios de negócio (ex: componentes de GUI).

Técnicas de Implementação:

- **Herança e Composição**: Mecanismos de Orientação a Objetos para reuso de código. A herança cria especializações, enquanto a composição constrói objetos a partir de outros.
- **Linhas de Produto de Software (LPS)**: A abordagem mais robusta para reuso sistemático, onde uma arquitetura base e ativos comuns são compartilhados para criar eficientemente múltiplas variantes de produtos.

Seção 3: Benefícios e Desafios

3.1 Benefícios do Reuso

- Aumento da **produtividade** e redução do tempo de desenvolvimento, Ezran et al. [1].
- Melhora da **qualidade** e **confiabilidade** do software, já que os ativos reutilizados já foram testados.
- Redução de **custos** a longo prazo
- **Consistência e Vantagem Competitiva**.

3.2 Desafios da Implementação do Reuso

- **Custo inicial, curva de aprendizagem:** O investimento na criação de ativos reutilizáveis pode ser alto, Ezran et al. [1].
- **Gerenciamento de Ativos:** A criação de um repositório de ativos (asset repository)
- **Barreiras Culturais e Organizacionais:** A falta de incentivo para que desenvolvedores criem ou usem componentes existentes.
- **Compatibilidade e Integração**: A reutilização de ativos de software pode introduzir desafios de compatibilidade, especialmente em ambientes heterogêneos. COTS.

Seção 4: Discussão e Tendências Atuais: O Impacto da Inteligência Artificial

4.1 IA como Catalisador do Reuso de Software

- A IA, especialmente LLMs, acelera o reuso ao automatizar a **geração de código**, refatoração e documentação de ativos. A IA também auxilia na análise de arquiteturas para identificar padrões de design e otimizações, Taivalsaari et al. [2].

4.2 A Mudança de Paradigma: Reuso e Dependências na Era da IA

- O debate se concentra na transição do **reuso tradicional** (de artefatos existentes) para uma abordagem **"AI Native"** (gerar código do zero). Isso desafia o reuso baseado em dependências, mudando o foco para a **reutilização de conhecimento e requisitos** para guiar a IA.

4.3 Desafios e Tendências Futuras

- Os desafios incluem a dificuldade da IA em capturar o "porquê" das decisões de design e o risco de herdar **vieses e imprecisões** dos dados de treinamento. A tendência é que a **IA e o reuso coexistam**, com a IA sendo uma ferramenta que capacita arquitetos a aplicar princípios de design de forma mais eficiente.

Seção 5: Conclusão

- **Síntese**: O reuso de software é uma estratégia de negócio que evolui de uma prática ad-hoc para uma disciplina sistemática.
- **Recapitulação**: Seções
- **Mensagem Final**: A IA não torna o reuso obsoleto; ela o transforma. O futuro do reuso de software reside na coexistência entre a IA, que facilita e acelera o processo, e os arquitetos, que mantêm a visão e o controle estratégico sobre as decisões de design e a qualidade final do sistema.

Referências

[1] EZRAN, Michel; MORISIO, Maurizio; TULLY, Colin. Practical Software Reuse. Berlin: Springer, 2013.
[2] TAIVALSAARI, Antero; MIKKONEN, Tommi; PAUTASSO, Cesare. On the Future of Software Reuse in the Era of AI Native Software Engineering. *arXiv preprint arXiv:2508.19834*, 2025.
[2] TAYLOR, Richard N. Software architecture: foundations, theory, and practice. Hoboken: John Wiley & Sons, 2010.

3. Arquitetura de Software

Seção 1: Introdução

1.1 Definição

- É a estrutura fundamental de um sistema, que abrange seus componentes, suas relações e o ambiente em que se insere [3].
- É a ferramenta que permite transformar os requisitos em um produto funcional, confiável e com valor para o cliente, garantindo que a visão do projeto seja mantida ao longo de todo o seu ciclo de vida.

1.2 Importância

- O fato de que as decisões arquiteturais são as mais difíceis de mudar e afetam diretamente a qualidade, o desempenho e a evolução do sistema a longo prazo [3].
- O objetivo central é:
    - **Analisar** a eficácia do projeto antes da implementação
    - **Reduzir riscos** associados à construção do software
    - **Facilitar a comunicação** entre todas as partes interessadas
    - **Promover a reutilização** de conhecimentos

1.3 Objetivo

- O artigo irá detalhar os atributos de qualidade, os padrões arquiteturais e a importância de uma arquitetura bem definida.

Seção 2: Fundamentos e Atributos de Qualidade

2.1 Os Atributos de Qualidade

- São as principais métricas de sucesso de uma arquitetura.
- Exemplos: Desempenho (capacidade de resposta), Segurança (proteção contra ameaças), Manutenibilidade (facilidade de alteração) e Escalabilidade (capacidade de lidar com mais usuários), Taylor, R. [3]
- Segundo Pressman [2], adiciona “funcionalidade, usabilidade, confiabilidade, desempenho e suporte” FURPS.

2.2 O Modelo 4+1 de Vistas

- A arquitetura não é uma única representação. O modelo de Kruchten, P. [1] é um framework para documentar a arquitetura em diferentes vistas, como a Lógica, de Processos, de Desenvolvimento e Física.

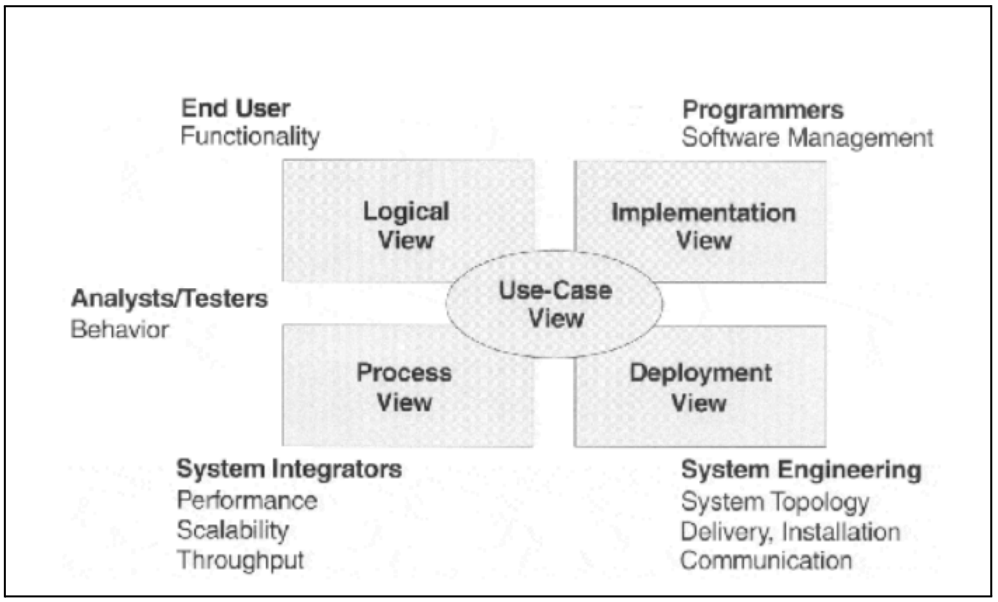


Figura 1: O Modelo 4+1 de Vistas de Arquitetura.
Fonte: Adaptado de KRUCHTEN, P. [1]

Seção 3: Padrões Arquiteturais e de Projeto

3.1 Estilos Arquiteturais

- Soluções de design em alto nível para problemas recorrentes. Focam na estrutura geral do sistema, Taylor, R. [3].
- Dois padrões principais:
    - **Estilo em Camadas:** Organiza o sistema em camadas hierárquicas.
    - **Pipe-and-Filter:** Fluxo de dados sequencial entre componentes.

3.2 Padrões Arquiteturais e de projeto

- **Padrões de Projeto:** Soluções de design de nível mais baixo, focadas em problemas específicos dentro de um módulo, Taylor, R. [3].
    - **Padrão de Três Camadas (Three-Tier):** Padrão que separa a aplicação em **Apresentação**, **Lógica de Negócios** e **Dados**.

Diferença-chave:

- **Estilos Arquiteturais** definem o design macro (o sistema como um todo).
- **Padrões de projeto** resolvem problemas em nível de componente.

Seção 4: Discussão e Tendências Atuais

A Arquitetura de Software evoluiu para se alinhar com as práticas modernas de desenvolvimento.

- **No contexto Ágil:**
    - Não é um grande projeto inicial, mas um **design evolutivo e contínuo**.

    - A arquitetura deve ser **flexível** para permitir mudanças rápidas a cada iteração, sem perder a integridade.
- **Arquitetura de Microsserviços [3]:**
    - É um padrão que consiste em construir a aplicação como um conjunto de **serviços pequenos e independentes**.
    - **Vantagens:** Escalabilidade independente, resiliência (falha de um serviço não afeta o todo) e autonomia de equipes.
    - **Desvantagens:** Maior complexidade na comunicação e gestão do sistema.

Seção 5: Conclusão

Síntese:
Recapitulação:
Mensagem Final: Uma boa arquitetura é a base para um software de qualidade; é importante em um ambiente em constante mudança.

Referências

[1] KRUCHTEN, P. Introdução ao RUP Rational Unified Process. Rio de Janeiro: Ciência Moderna, 2004.
[2] PRESSMAN, Roger S.. Engenharia de Software. 6a ed., São Paulo, McGraw-Hill, 2006
[3] TAYLOR, Richard N. Software architecture: foundations, theory, and practice. Hoboken: John Wiley & Sons, 2010.

4. Verificação, Validação e Teste de Software

Seção 1: Introdução

1.1 Definição

- Processo de garantir que o software atenda às expectativas e funcione corretamente, para qualidade de soft.

1.2 Conceitos Essenciais

- Distinção entre **Verificação** ("Estamos construindo o produto certo?") e **Validação** ("Estamos construindo o produto da maneira certa?")
- Teste de Software. Sommerville [3]

1.3 Objetivo

- Detalhar os fundamentos, os tipos de teste e o papel da área em metodologias modernas, como as ágeis.

Seção 2: Fundamentos de Verificação e Validação

2.1 Métodos de Verificação (Estáticos)

- Incluí métodos estáticos. Estamos construindo o produto corretamente? Inspeções e revisões em todos os processos desde os requisitos até o código-fonte. Técnicas de veri.:
    - Inspeções de software, erros, omissões e anomalias. Vanta. Muitos defeitos de uma única vez.
    - Análise Estática Automatizada: escaneiam o texto-fonte de um prog. para detec. possíveis defeitos e anomalias, como variáveis não iniciadas, código inacessível, inconsistências de interface, comple. A detecção de erros do compil.

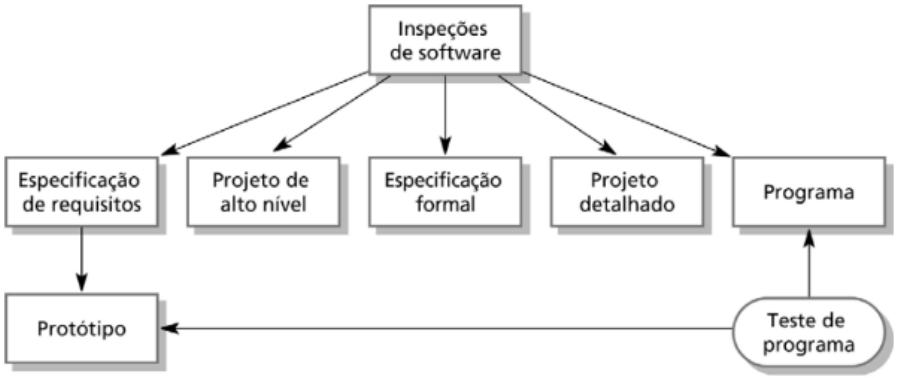


Figura 1: Verificação e Validação dinâmica e estática. Fonte: Sommerville [3]

2.2 Papel da Validação (Dinâmica)

- Validação como a execução do software para garantir que ele atenda aos requisitos do usuário.
- Validação diretamente à atividade de **Teste de Software**.

Seção 3: Tipos e Estratégias de Teste de Software

Existem duas estratégias principais de teste, que são complementares para uma avaliação completa do software, Bartié [1].

3.1 Testes de Caixa Branca

- **Foco:** A lógica interna e a estrutura do código. O objetivo é testar caminhos de execução, loops e condições lógicas
- **Conhecimento:** Requer acesso e conhecimento do código-fonte e da arquitetura do software.
- **Métodos:** Cobertura de código, cobertura de caminhos, e cobertura de desvios condicionais.

3.2 Testes de Caixa Preta

- **Foco:** A funcionalidade e o comportamento do sistema a partir dos requisitos.
- **Conhecimento:** Não exige conhecimento do código. O testador baseia-se nas regras de negócio e nas especificações funcionais.
- **Métodos:** Particionamento de equivalência e análise de valores-limite.

A combinação de ambas as estratégias garante a qualidade tanto da implementação interna quanto da funcionalidade externa do software.

Ainda existem outros tipos de teste

- **Unidade:** focada em componentes individuais.
- **Integração:** Avalia a interação entre componentes.
- **Sistema:** Valida o sistema como um todo.
- **Aceitação:** Confirma se o sistema atende às necessidades do cliente.
- **Regressão:** Garante que as novas mudanças não prejudicaram o que já estava funcionando.

Seção 4: Discussão sobre o papel da V&V no Contexto de Projetos

A V&V, em um contexto moderno, é uma disciplina estratégica e integrada ao desenvolvimento, focada na qualidade contínua.

4.1 V&V em Metodologias Ágeis

- Em ambientes ágeis, a V&V é uma atividade contínua e integrada, realizada em ciclos curtos e iterativos.
- O foco é validar a qualidade do software a cada novo incremento, e não apenas no final do projeto.
- O teste de regressão se torna essencial, com alta automação, para garantir que as novas funcionalidades não introduzam falhas em partes já existentes.

4.2 V&V e a Inteligência Artificial Generativa

Como o ChatGPT e o Microsoft Copilot, Haldar et al.,[2]

- **Aceleração da Criação de Artefatos:** Ferramentas como o Copilot e o ChatGPT podem gerar rapidamente artefatos preparatórios de teste, como casos de teste, *use cases* e scripts de teste.
- **Análise de Requisitos e Traceabilidade:** A IA pode auxiliar na criação de matrizes de rastreabilidade (RTMs), que conectam requisitos a casos de teste.
- **Identificação de Gaps (lacunas):** A IA pode ajudar a identificar lacunas em testes não funcionais (como usabilidade, segurança e performance) e em casos de borda (*edge cases*), que podem passar despercebidos em testes manuais.
- **Otimização do Processo:** Embora a IA ofereça vantagens em velocidade e cobertura, a supervisão humana ainda é crucial para garantir a precisão, relevância e qualidade dos artefatos gerados

Seção 5: Conclusão

- Síntese:
- Recapitulação
- Mensagem Final: V&V é uma disciplina essencial para a entrega de software de qualidade, com foco em métodos automatizados e contínuos. A importância do tema para a Engenharia de Software como um todo.

Referências

[1] BARTIE, A. Garantia da Qualidade de Software. Rio de Janeiro: Elsevier, 2002.
[2] HALDAR, Susmita; PIERCE, Mary; CAPRETZ, Luiz Fernando. Exploring the Integration of Generative AI Tools in Software Testing Education: A Case Study on ChatGPT and Copilot for Preparatory Testing Artifacts in Postgraduate Learning. *IEEE Access*, 2025.
[2] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.

5. Manutenção, Refatoração e Evolução de Software

Seção 1: Introdução

1.1 Definição

- M. de Soft. É o processo de modificar um sistema após a sua entrega para corrigir defeitos, melhorar o desempenho ou adaptar-se a um novo ambiente, Sommerville [4].
- A e. de software é a necessidade inevitável de mudança para que um sistema permaneça útil, integrando o desenvolvimento e a manutenção em um ciclo de vida contínuo. Mud. contínua e da Compl. crescente, Pressman [3].
- A refatoração é o processo de alterar a estrutura interna de um software sem modificar seu comportamento externo, com o objetivo de aprimorar seu design e legibilidade, Fowler [2].

1.2 Importância

- Estatísticas indicam que a maior parcela do esforço de manutenção não está no reparo de defeitos (17%), mas na adaptação a novos ambientes (18%) e na adição ou modificação de funcionalidades (65%), Sommerville [4].

1.3 Objetivo

- Irá detalhar os tipos de manutenção, a relação entre manutenção e o papel da refatoração e as tendências atuais, como a IA.

Seção 2: Tipos de Manutenção de Software

A manutenção de software, que consome a maior parte dos recursos, é dividida em quatro tipos principais, Sommerville [4]:

2.1 M. Corretiva

- Foca na correção de defeitos para garantir a confiabilidade.

2.2 M. Adaptativa

- Adapta o sistema a mudanças de ambiente para assegurar sua longevidade.

2.3 M. Perfeccionista

- Melhora a estrutura interna do código, prevenindo sua degradação.

2.4 M. Evolutiva

- A mais significativa, adiciona ou modifica funcionalidades. Regida pelas **Leis de Lehman**, que destacam a necessidade de mudança contínua e o crescimento da complexidade.
    - Lei da Mudança Contínua
    - Lei da Complexidade Crescente

Seção 3: A Refatoração como Estratégia de Evolução

Diferencia a refatoração de outros tipos de manutenção e a posiciona como uma prática moderna.

3.1 Definição de Refatoração

- A refatoração é o processo de reestruturar o código interno de um software, em pequenos passos, sem alterar o seu comportamento externo e inserir novos bugs, para melhorar a sua legibilidade, manutenibilidade e qualidade, Fowler [2].

3.2 Conexão com a Manutenção

- A refatoração é um tipo de manutenção perfeccionista e é fundamental para a manutenção evolutiva, Fowler [2].
    - Refatoração preparatória - refatorar antes de adicionar uma nova funcionalidade.
    - Contínua - refatora imediatamente

Seção 4: Discussão e Tendências atuais

A manu. e evo., são vistas como etapas reativas, com processos contínuos essenciais. Refatoração, garantir a sustentabilidade do código permite de forma proativa adaptar e evoluir.

4.1 Tendências Atuais

- Auxiliar na análise do código, identificar “code smells”, refatorações automáticas.
- Um estudo recente de Divyansh et al. [1] investigou o ChatGPT na melhoria da qualidade, em 4 métricas: complexidade ciclomática, complexidade cognitiva, “code smells” no código e débito técnico.
    - Resultados: redução da complexidade (mais fácil de entender), diminuição dos code smells (mais críticos e bloqueadores), diminuição do débito técnico (redução 24h).

Seção 5: Conclusão

- **Síntese**
- **Recapitulação**
- **Mensagem Final**: A Manutenção, a Refatoração e a Evolução não são atividades secundárias, mas sim o cerne do ciclo de vida do software, e uma gestão eficaz destas fases é crucial para a sustentabilidade de um sistema a longo prazo.

Referências

[1] Divyansh, S., et al. (2025). Evaluating the Effectiveness of ChatGPT in Improving Code Quality. *2025 IEEE 4th International Conference on Computing and Machine Intelligence (ICMI)*.
[2] FOWLER, Martin. Refatoração: aperfeiçoando o projeto de código existente. Porto Alegre, RS: Bookman, 2004.
[3] PRESSMAN, Roger S.. Engenharia de Software. 6a ed., São Paulo, McGraw-Hill, 2006.
[4] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.

6. Melhoria de Processo de Software

Seção 1: Introdução

1.1 Definição

- Melhoria de Processo de Software (MPS) como um esforço contínuo para aprimorar os processos de desenvolvimento e manutenção, visando aumentar a produtividade, a qualidade e a previsibilidade [4].

1.2 Importância

- O fato de que um processo maduro e bem definido é a base para a entrega de software de alta qualidade e para a redução de riscos [4].

1.3 Objetivo

- O artigo irá explorar os modelos de maturidade de processo, as principais abordagens de melhoria e os benefícios para a organização.

Seção 2: Modelos de Maturidade de Processo

2.1 Modelo CMMI (Capability Maturity Model Integration)

- O CMMI é um modelo de referência para a melhoria de processos, que ajuda as organizações a avaliar sua maturidade e a planejar melhorias.
- Os *níveis de maturidade* do modelo (Inicial, Gerenciado, Definido, Quantitativamente Gerenciado, Otimizando) [1].

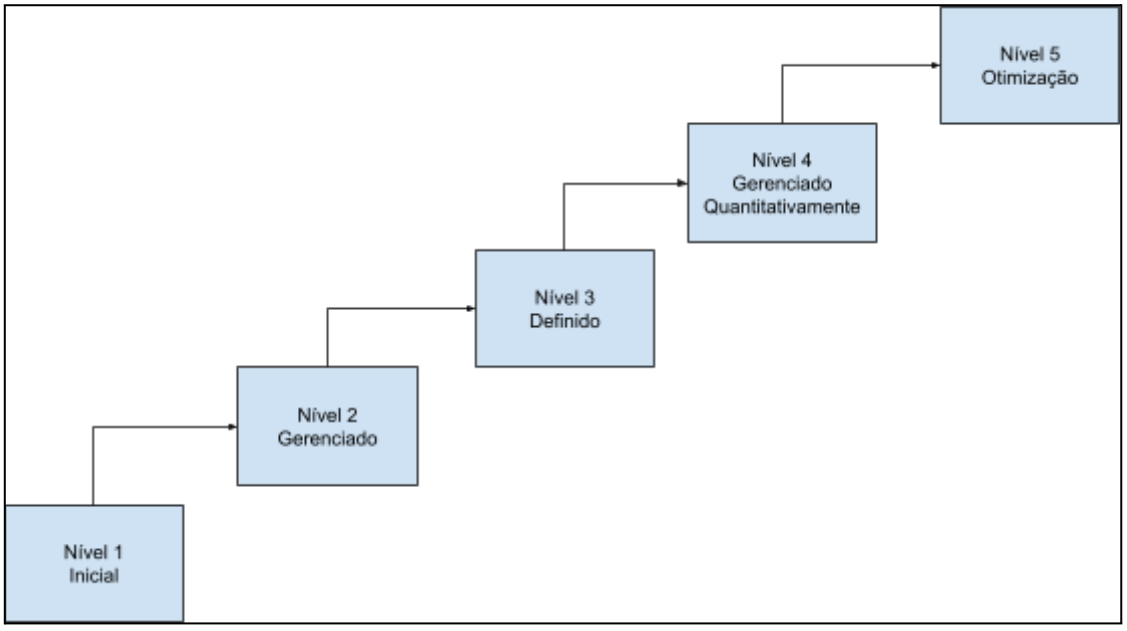


Figura 1: Modelo CMMI por níveis da maturidade.
Fonte: Adaptado de Chrissis et al.[1]

2.2 O Ciclo de Melhoria de Processo

- A melhoria de processo como um ciclo cíclico e contínuo, Sommerville [4].
- Medicação de Processo, Análise de Proc., Mudança de Proc.

Seção 3: Gerenciamento e Métricas

3.1 A Estratégia de Melhoria

- A necessidade de uma estratégia que inclua a definição de objetivos, a medição do processo e o controle de mudanças [3].

3.2 Métricas de Processo e Produto

- A importância de medir para melhorar.
- Métricas de processo (ex.: tempo de ciclo, densidade de defeitos) e métricas de produto (ex.: número de bugs por linha de código) [3].

Seção 4: Discussão e Tendências Atuais

4.1 Melhoria de Processo no Contexto Ágil

- A transição de um processo formal e burocrático (como o CMMI) para uma melhoria mais orgânica e focada em feedback contínuo [3].
- Reuniões de *Retrospectiva* no Scrum como um exemplo de uma prática ágil para melhoria de processo contínua [3].

4.2 O Papel da Automação na Melhoria de Processo

- Como a automação de testes, a integração e a entrega contínua (CI/CD) são ferramentas que implementam a melhoria do processo [2].

Seção 5: Conclusão

- Melhoria de Processo de Software é uma disciplina que exige compromisso contínuo, e que sua aplicação, seja por modelos formais ou por abordagens ágeis, é essencial para a competitividade e a qualidade do software.

Referências

[1] CHRISSIS, M. B.; KONRAD, M.; SHRUM, S. CMMI: guidelines for process integration and product improvement. 2.ed. Upper Saddle River: Person Addison-Wesley, 2006.
[2] DUVALL, P. M.; MATYAS, S.; GLOVER, A. Continuous Integration: improving software quality and reducing risk. Upper Saddle River: Addison-Wesley, 2007.
[3] PRESSMAN, Roger S.. Engenharia de Software. 6a ed., São Paulo, McGraw-Hill, 2006.
[4] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.

7. Linhas de Produto de Software

Seção 1: Introdução

1.1 Definição

- A Linha de Produto de Software (LPS) é uma estratégia de reuso em escala, em que uma família de sistemas de software é desenvolvida a partir de um conjunto comum de ativos reutilizáveis, Ezran et al. [2].

1.2 Motivação

- A principal razão para adotar a LPS: a redução do tempo de desenvolvimento, a diminuição de custos e o aumento da qualidade ao produzir sistemas semelhantes de forma sistemática, Ezran et al. [2].

1.3 Objetivo

- O artigo irá explorar os conceitos de *core asset*, variabilidade e os benefícios e desafios de uma LPS, Machine Learning.

Seção 2: Conceitos Fundamentais de Linhas de Produto de Software

2.1 O Conceito de *Core Asset*

- O *core asset* é o conjunto de ativos reutilizáveis (arquitetura, componentes, código, modelos, etc.) que formam a base para todos os produtos da linha, Ezran et al. [2].
    - Arquitetura de Referência, Componentes reutilizáveis, Modelos de Análise e Requisitos e Planos de Teste.

2.2 Gerenciamento da Variabilidade

- A **variabilidade** é a capacidade de uma LPS de produzir diferentes produtos a partir de um conjunto de ativos comuns, Buchmann et al. [1].
- O papel do Gerenciamento de Requisitos, Pressman [5], e do Gerenciamento de Configuração, Molinari [4], para lidar com as variações entre os produtos.

Seção 3: Benefícios e Desafios da Implementação de uma LPS

3.1 Benefícios

- Aumento da produtividade e da velocidade de mercado (time-to-market) [2].
- Melhoria da qualidade e confiabilidade dos produtos [2].
- Redução de custos de desenvolvimento e manutenção a longo prazo [2].

3.2 Desafios

- Custo inicial: O investimento na criação dos *core assets* é alto [2].
- Gerenciamento complexo: Lidar com a variabilidade e a evolução dos ativos reutilizáveis exige um gerenciamento rigoroso [2].
- Questões organizacionais: O sucesso de uma LPS depende de uma mudança na cultura da organização [2].

Seção 4: Discussão e Tendências Atuais: A Intersecção entre LPS e Aprendizado de Máquina

- A crescente complexidade de sistemas de software modernos, que agora incorporam modelos de ML e Modelos de Linguagem de Grande Escala (LLMs), tem impulsionado a busca por metodologias que permitam gerenciar essa complexidade de forma eficaz, Gomez-Vazquez et al. [3].

4.1 Desafios na Combinação de Modelos de ML

- **Modelos de ML (e LLMs):** Grandes, caros para treinar/executar; **Técnicas de combinação:** MoE (Mixture of Experts), fusão de modelos; **Complexidade da combinação:** Escolha dos modelos ("especialistas"); Estratégias de fusão; Associação de tarefas aos especialistas (MoE); Infinidade de configurações.

4.2 A LPS como Abordagem para a Engenharia de Modelos de ML

- **Propósito:** Gerenciar complexidade e democratizar criação de modelos combinados; **Paralelo LPS-ML:** LPS é ideal para gerenciar **variabilidade** (presente tanto em SW tradicional quanto em modelos de ML; **Feature Models (LPS):** Mecanismo central para especificar a "linha de produto" de ML; Define os **pontos de variabilidade** na combinação de ML; Tipo de modelos a combinar; Estratégia de combinação; Associação de tarefas aos especialistas; **Benefícios:** Processo sistemático, claro e repetível para especificação/treinamento de arquiteturas de ML.

Seção 5: Conclusão

- Síntese: A LPS é uma estratégia de reuso avançada e poderosa que, se bem gerenciada, pode trazer benefícios significativos para uma organização, mas exige um planejamento cuidadoso e um compromisso de longo prazo.
- Recapitlação
- Mensagem Final:

Referências

[1] BACHMANN, Felix; CLEMENTS, Paul C. Variability in Software Product Lines. Software Engineering Institute, Pittsburgh. set. 2005. 61p. Relatório Técnico.

[2] EZRAN, Michel; MORISIO, Maurizio; TULLY, Colin. Practical Software Reuse. Berlin: Springer, 2013.

[3] GOMEZ-VAZQUEZ, Marcos; CABOT, Jordi. Exploring the use of software product lines for the combination of machine learning models. In: *Proceedings of the 28th ACM International Systems and Software Product Line Conference*. 2024. p. 26-29.

[4] MOLINARI, L. Gerência de Configuração: técnicas e práticas no desenvolvimento do software. Florianópolis: Visual Books, 2007.
[5] PRESSMAN, Roger S.. Engenharia de Software. 6a ed., São Paulo, McGraw-Hill, 2006.

8. Inteligência Artificial aplicada na Engenharia de Software

Seção 1: Introdução

1.1 Definição

- A **Inteligência Artificial (IA) na Engenharia de Software** é o uso de técnicas de IA para auxiliar, automatizar e aprimorar o processo de desenvolvimento, que é cada vez mais complexo, Ahmed et al. [1].
- A ascensão da IA, e mais recentemente, dos grandes modelos de linguagem (LLMs), representa a inovação mais disruptiva no campo desde a popularização da Internet, Pezzè et al. [2].

1.2 Impacto

- O impacto profundo da IA na área está redefinindo o cenário e auxiliando em desafios clássicos e novos. Ferramentas como o GitHub Copilot e o Amazon CodeWhisperer, segundo Ahmed et al. [1], são apenas a "ponta do iceberg" de um intenso esforço de pesquisa e empreendedorismo.

1.3 Objetivo

- Explorará as principais áreas de aplicação da IA na engenharia de software e os desafios técnicos e organizacionais que a comunidade deve enfrentar para usá-la com sucesso. Apresentar as sessões de forma resumida.

Seção 2: Áreas de Aplicação da IA na Engenharia de Software

2.1 Engenharia de Requisitos e Projeto

- O papel da IA na classificação, extração e validação de requisitos. Mencionar sua aplicação incipiente no design de software.
- Como a IA é usada para classificar, extrair e validar requisitos? Como modelos de *deep learning* podem ser empregados na análise de requisitos, Pezzè et al. [2].

2.2 Codificação e Teste

- Aplicação da IA para acelerar tarefas complexas de codificação, como a geração de código, o *code completion* e o resumo de código, Ahmed et al. [1].
- Sobre o uso da IA para gerar casos de teste, *test oracles* e para detectar e localizar bugs, o que tem sido objeto de grande atenção na área.

2.3 Manutenção e Operações de TI (AIOps)

- O uso da IA para aprimorar operações de TI, com tarefas como detecção de anomalias e análise de causa raiz [1, 2].

Seção 3: Desafios e Questões em Aberto

3.1 Desafios Técnicos

- A necessidade de conjuntos de dados grandes, de alta qualidade e imparciais para treinar os modelos.
- A dificuldade de avaliar objetivamente os produtos e processos baseados em IA, devido à falta de *métricas* e de *test oracles* robustos.

3.2 Desafios Organizacionais e Éticos

- A questão da confiança e da *explicabilidade da IA (XAI)*, já que a natureza de "caixa-preta" de muitos modelos leva à desconfiança, Pezzè et al. [2].
- Os desafios de privacidade, violações de licença e segurança do código gerado por IA, que pode conter vulnerabilidades, Ahmed et al. [1].

Seção 4: Discussão e o Futuro da Profissão

4.1 IA e a Melhoria Contínua

- IA é uma ferramenta poderosa para automatizar a coleta de métricas de processo, contribuindo para a Melhoria de Processo, um tema central na Engenharia de Software.

4.2 IA e a Engenharia Experimental

- A eficácia de uma ferramenta baseada em IA deve ser validada por meio de experimentos e estudos empíricos.

4.3 O Futuro da Profissão

- IA não irá substituir o engenheiro de software. Em vez disso, ela automatizará tarefas repetitivas, permitindo que os profissionais se concentrem em atividades de maior valor, como resolução de problemas criativos e arquitetura de software.

Seção 5: Conclusão

- **Síntese:** IA é um catalisador para a inovação e o crescimento sustentável da área. No entanto, seu uso requer um plano cuidadoso, que leve em conta a transparência e as questões éticas para garantir que ela aprimore a profissão sem comprometer seus valores fundamentais.
- **Recapitulação**
- **Mensagem Final**:

Referências

[1] AHMED, I., Aleti, A., Cai, H., Chatzigeorgiou, A., He, P., Hu, X., ... & Xia, X. (2025). Artificial Intelligence for Software Engineering: The Journey so far and the Road ahead. ACM Transactions on Software Engineering and Methodology, 34(5), 1-27.

[2] PEZZÈ, M., Abrahão, S., Penzenstadler, B., Poshyvanyk, D., Roychoudhury, A., & Yue, T. (2025). A 2030 Roadmap for Software Engineering. ACM Transactions on Software Engineering and Methodology, 34(5), 1-55.

9. Engenharia de Software Experimental

Seção 1: Introdução

1.1 Definição

- Engenharia de Software Experimental (ESE) é a disciplina que aplica métodos científicos para testar hipóteses, validar práticas e construir um corpo de conhecimento baseado em evidências [4].

1.2 A Importância da ESE

- O fato de que a ESE é fundamental para a maturidade da Engenharia de Software, permitindo que as decisões sejam tomadas com base em dados e não apenas em intuição [4].

1.3 Objetivo

- Este trabalho irá explorar os fundamentos da ESE, os tipos de estudos empíricos e o seu papel na prática de software.

Seção 2: Fundamentos da Engenharia de Software Experimental

2.1 O Processo de Experimentação

- As etapas básicas de um estudo experimental, como Escopo, Planejamento, Operação, Análise e Interpretação, Apresentação e Empacotamento, de acordo com Wohlin et al. [4].

2.2 Tipos de Estratégias Empíricas

- **Experimentos:** os experimentos são estudos controlados, nos quais uma ou mais variáveis são manipuladas para testar uma hipótese.
- **Estudos de Caso** (*Case Studies*)**:** são estudos aprofundados de um ou mais projetos de software, sem um controle rigoroso de variáveis.
- **Estudos de Levantamento** (*Surveys*)**:** como a coleta de dados de uma grande população de desenvolvedores ou empresas, usando questionários ou entrevistas.
  A figura 1 ordena os estudos com base em como eles normalmente podem ser conduzidos para permitir uma maneira controlada de transferir os resultados de pesquisa para a prática.

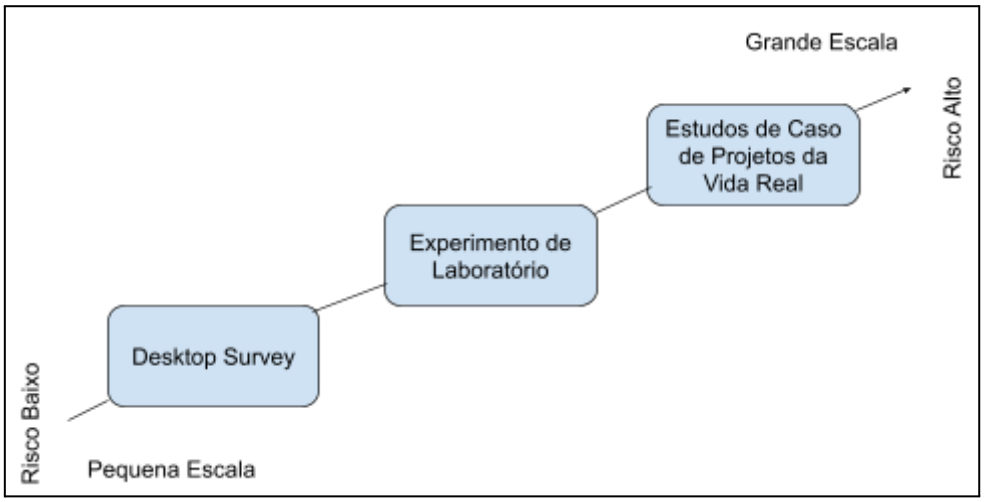


Figura 1: Surveys, Experimentos e Estudos de Caso.

Fonte: Wohlin et al. [4].

Seção 3: Validade e Medições em Estudos Empíricos

De acordo com Cook e Campbell [1]:

- Validade de Conclusão: conclusões estatisticamente corretas sobre a relação entre variáveis.
- Validade Interna: uma relação de causa e efeito é verdadeira, e não por fatores externos não controlados.
- Validade do Construto: grau em que as variáveis medidas representam os construtos teóricos que se pretende estudar.
- Validade Externa: generalizar os resultados do estudo para outros contextos, ambientes e populações.
- Qualidade de Medições de acordo com Wohlin et al. [4].
    - **Medidas diretas** são mensuráveis por si só, como o tempo e o tamanho de um software.
    - **Medidas indiretas** são derivadas de outras medições, como produtividade e densidade de defeitos.
    - **Medidas objetivas** não dependem de julgamento humano e podem ser repetidas com o mesmo resultado, como linhas de código.
    - **Medidas subjetivas** dependem de algum tipo de julgamento humano, como a usabilidade

Seção 4: O Papel da EBSE e as Revisões Sistemáticas

4.1 O Movimento *Evidence-Based Software Engineering* (EBSE)

- O EBSE é a prática de tomar decisões de software com base nas melhores evidências disponíveis.
- ESE como a fonte primária dessas evidências [3].

4.2 Revisões Sistemáticas (*Systematic Reviews*)

- As revisões sistemáticas são utilizadas como um método para sintetizar as evidências existentes sobre um tema específico [3].
    - Planejamento, Busca por Estudos Primários, Seleção de Estudos, Avaliação da Qualidade dos Estudos, Síntese e Análise dos Dados.

4.3 O Uso da Inteligência Artificial Generativa em EBSE

- O volume crescente de publicações científicas e a maior acessibilidade a bibliotecas digitais tornam as revisões de literatura e estudos de mapeamento tarefas demoradas e trabalhosas, Esposito et al.[2].
- A IA Generativa (IA Generativa) tem potencial para otimizar e agilizar o processo de pesquisa.
- Essa nova tecnologia pode simplificar e acelerar a geração e análise de textos, que são atividades centrais em estudos sistemáticos

4.3.1 Aplicações da IA Generativa nas Etapas de uma Revisão Sistemática

- **Criação da Estratégia de Busca:** LLMs (Large Language Models) podem gerar termos PICO e strings, Busca e Documentação, Seleção de Estudos, Avaliação da Qualidade, Extração de Dados, Síntese e Análise. Desafios e Considerações Éticas (Replicabilidade, Explicabilidade (Explainability), Função da IA)

Seção 5: Conclusão

- **Síntese**
- **Recapitulação**: Resumir os tópicos abordados.
- **Mensagem Final**: Engenharia de Software Experimental é essencial para a maturidade da disciplina, pois fornece um caminho para a validação de práticas e para a construção de um conhecimento científico sólido.

Referências

[1] COOK, T.D., Campbell, D.T.: Quasi-experimentation – Design and Analysis Issues for Field Settings. Houghton Mifflin Company, Boston (1979).
[2] ESPOSITO, Matteo, et al. Generative AI in Evidence-Based Software Engineering: A White Paper. *arXiv preprint arXiv:2407.17440*, 2024.
[3] KITCHENHAM Barbara Ann, David Budgen, and Pearl Brereton. 2015. Evidence-Based Software Engineering and Systematic Reviews. Chapman & Hall/CRC.
[4] WOHLIN, Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated.

10. Padrões Arquiteturais e de Projetos

Seção 1: Introdução

1.1 Definição

- Os padrões de software são como soluções comprovadas para problemas recorrentes de design e arquitetura [2, 3].

1.2 Importância

- O fato de que os padrões fornecem um vocabulário comum, reduzem o risco e o retrabalho e promovem a reutilização de conhecimento, não apenas de código.

1.3 Diferença-Chave

- A distinção fundamental entre *Padrões Arquiteturais* (macro-nível, estrutura do sistema) e *Padrões de Projeto* (micro-nível, design de classes e objetos) [2] e [3].

1.4 Objetivo

- Irá explorar e exemplificar esses dois tipos de padrões, mostrando como eles se complementam no processo de design de software.

Seção 2: Padrões Arquiteturais

2.1 Conceito

- Os padrões arquiteturais destacam seu papel na organização da estrutura e na definição dos componentes de um sistema [4].

2.2 Exemplos Práticos

- Cliente-Servidor: esse padrão fundamental, onde o cliente solicita um serviço e o servidor o fornece (ex: aplicações web).
- Microsserviços: padrão moderno, em que um sistema complexo é quebrado em serviços pequenos e independentes [4].

Seção 3: Padrões de Projeto (*Design Patterns*)

3.1 Conceito

- Os padrões de projeto são como soluções para problemas específicos de design de software em um contexto de programação orientada a objetos [1].

3.2 A "Gang of Four" (GoF)

- Os padrões de projeto foram popularizados pela obra seminal [1]. Conforme ilustrado na tabela 1, é uma representação clássica da obra de **Gamma et al.**, a "Gang of Four".

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 1: O espaço dos padrões de projeto de acordo com Gamma et al. [1]

3.3 Categorias e Exemplos

- Padrões de Criação (Creational Pattern): Focados na criação de objetos (ex.: Singleton).
- Padrões Estruturais (Structural Pattern): Focados na composição de classes e objetos (ex: Adapter).
- Padrões Comportamentais (Behavioral Pattern): Focados na comunicação entre objetos (ex.: Observer).

Seção 4: Discussão e a Complementaridade entre os Padrões

4.1 A Complementaridade dos Padrões

- Uma analogia para explicar a relação entre eles (ex: um padrão arquitetural é o plano urbanístico da cidade, enquanto um padrão de projeto é o projeto detalhado de um edifício dentro da cidade).

4.2 Padrões e a Reutilização de Conhecimento

- Retome o conceito de reuso de software, mas agora focado na reutilização de conhecimento e soluções, não apenas de código.

Seção 5: Conclusão

Os padrões são uma ferramenta fundamental da Engenharia de Software, promovendo a criação de sistemas mais manuteníveis, flexíveis e robustos, ao fornecer um catálogo de soluções testadas e um vocabulário comum para a comunicação entre desenvolvedores.

Referências

[1] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.
[2] MARCO Tulio Valente. Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade, Editora: Independente, 2020.
[3] PRESSMAN, Roger S.. Engenharia de Software. 6a ed., São Paulo, McGraw-Hill, 2006.
[4] TAYLOR, Richard N. Software architecture: foundations, theory, and practice. Hoboken: John Wiley & Sons, 2010.

11. Gestão de Projetos de Software

Seção 1: Introdução

1.1 Definição

- GPS é a aplicação de conhecimento para entregar produtos digitais [8].

1.2 Contexto

- Evoluiu para se adaptar às mudanças de software [4,8].

1.3 Modelos

- Modelos de ciclo de vida do projeto: inicial e predominante, tradicional como cascata, planejamento rigoroso, definição completa dos requisitos. Sistemas complexos [9].
- Migração para abordagens ágeis.
    - Scrum, Kanban [2]

1.4 Objetivo

- Este texto tem como objetivo discutir os pilares fundamentais do gerenciamento de projetos de software. A Seção 2 abordará as atividades, o planejamento e os riscos inerentes aos projetos. A Seção 3 discutirá a transição para o paradigma ágil. A Seção 4 analisará as tendências atuais, com foco na Inteligência Artificial. Por fim, a Seção 5 apresentará as considerações finais sobre o tema.

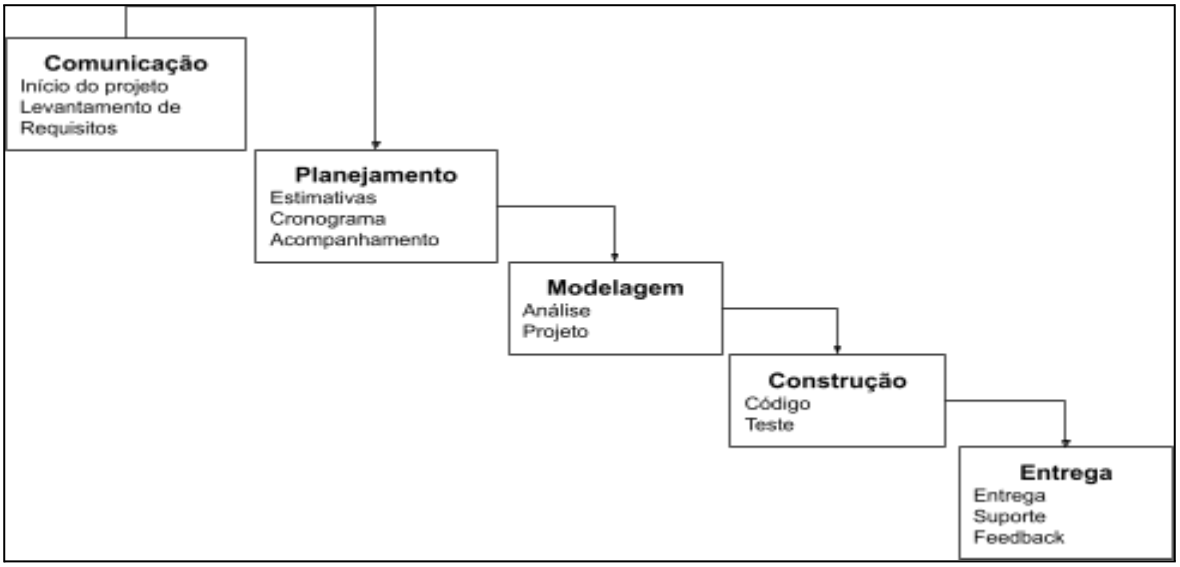
Seção 2: Abordagens de gerenciamento de projetos

2.1 Modelo Tradicional

- Fases lineares e sequenciais (ex: Cascata). Vantagem: previsibilidade. Desvantagem: rigidez. [8]

2.2 Desenho do modelo cascata

- Figura 1 - Modelo Cascata de acordo com Pressman [8].



2.3 Ágil

- Foco em colaboração e adaptabilidade. Entrega incremental e frequente [9]. Adoção de metodologias como Scrum/Kanban [3].

Seção 3: Gerenciamento da Qualidade e Configuração

3.1 Importância da Garantia da Qualidade (QA)

- Conceito: Conjunto de atividades para garantir que o software cumpra requisitos. Conexão com a gestão: prevenção de defeitos, redução de retrabalho, impacto no cronograma e custo [2].

3.2 O Papel do Gerenciamento de Configuração de Software (GCS)

- Conceito: Processo de controle e rastreamento de mudanças (código, documentação).
- Conexão com a Gestão: Manter a integridade dos artefatos e deslizamento de escopo [6].

Seção 4: Discussão e Tendências Atuais

4.1 A Complementaridade do PMBOK e do SWEBOK

- **PMBOK** (Project Management Body of Knowledge) e **SWEBOK** (Software Engineering Body of Knowledge) são complementares. Um é sobre "como gerenciar", o outro é sobre "como construir". [5]

4.2 A Convergência com as Práticas de DevOps

- Como a gestão ágil se integra com a automação e a entrega contínua (CI/CD), tornando o processo de desenvolvimento e entrega mais rápido e confiável [1].

4.3 A Influência de Tecnologias Emergentes

- Inteligência Artificial e a análise de dados estão sendo usadas para otimizar processos de gestão [7].

Seção 5: Conclusão

- A escolha da abordagem de gestão depende do contexto do projeto.
- Gestão de Projetos de Software moderna é uma disciplina **híbrida**, **analítica** e **centrada nas pessoas**, que integra o rigor do planejamento com a flexibilidade das metodologias ágeis.

Referências

[1] BANICA, Logica, et al. Is DevOps another project management methodology?. *Informatica Economica*, 2017, 21.3.

[2] BARTIE, A. Garantia da Qualidade de Software. Rio de Janeiro: Elsevier, 2002.

[3] COHN, M. Desenvolvimento de Software com Scrum: aplicando métodos ágeis com sucesso. Porto Alegre: Bookman, 2011.

[4] HELDMAN, K. Gerência de Projetos: guia para o exame oficial do PMI. 5.ed. Rio de Janeiro: Elsevier, 2009.

[5] IEEE, C. S. Guide to the Software Engineering Body of Knowledge. Disponível em: https://goo.gl/lddan1. Acesso em: 21 de agosto de 2019.

[6] MOLINARI, L. Gerência de Configuração: técnicas e práticas no desenvolvimento do software. Florianópolis: Visual Books, 2007.

[7] ONG, Stephen; UDDIN, Shahadat. Data science and artificial intelligence in project management: the past, present and future. *The Journal of Modern Project Management*, 2020, 7.4.

[8] PRESSMAN, Roger S.. *Engenharia de Software*. 6a ed., São Paulo, McGraw-Hill, 2006.

[9] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.

12. Gestão de Projetos de Software (2)

Seção 1: Introdução

1.1 Definição

- O Gerenciamento de Projetos (GP) de Software é uma disciplina essencial da Engenharia de Software, Sommerville [3].

1.2 Motivação

- Um mau gerenciamento é a principal causa de falhas em projetos, resultando em, Sommerville [3].:
    - Atrasos no cronograma.
    - Custos acima do orçamento.
    - Não atendimento aos requisitos do cliente.
- O objetivo do GP é assegurar que o software seja entregue atendendo às restrições de custo e prazo, agregando valor à organização.
- Desafios Únicos do GP de Software (Por que é difícil?), Produto Intangível, Processos Não Padronizados, Projetos "Únicos"
- A Evolução dos Paradigmas de Gestão (**Ponto de Partida:** Métodos tradicionais (preditivos/cascata), Sommerville [3]. **A Resposta:** O surgimento dos Métodos Ágeis, COHN [1]. **A Fronteira Atual:** A ascensão da Inteligência Artificial e da Ciência de Dados, ONG [2].

1.3 Objetivo do texto

- A Seção 2 abordará as atividades, o planejamento e os riscos inerentes aos projetos. A Seção 3 discutirá a transição para o paradigma ágil. A Seção 4 analisará as tendências atuais, com foco na Inteligência Artificial. Por fim, a Seção 5 apresentará as considerações finais sobre o tema

Seção 2: Atividades de Gerenciamento de Projetos

- **As Atividades Essenciais do Gerente,** Sommerville []**:**
    - **Definir o Projeto:** Elaborar propostas, planejar atividades e estimar custos.
    - **Gerenciar a Equipe:** Selecionar e avaliar o pessoal, lidando com restrições de orçamento e disponibilidade.
    - **Acompanhar e Comunicar:** Monitorar o progresso continuamente e gerar relatórios para as partes interessadas

2.1 Planejamento de Projetos

- **Processo Iterativo:** O plano de projeto não é estático; é revisado e atualizado durante todo o ciclo de vida do projeto, Sommerville [3].
- **Documentação (Plano de Projeto):** Estrutura o trabalho, definindo atividades, recursos, cronograma e riscos.

- **Controle:** Definir **Marcos** (pontos de controle para a gerência) e **Produtos** (entregas para o cliente)

2.2 Gerenciamento de Riscos

- **Objetivo:** Prever e se preparar para problemas que ameacem o **projeto** (cronograma), o **produto** (qualidade) ou o **negócio**.
    - Processo Cíclico de 4 Etapas, conforme Figura 1:

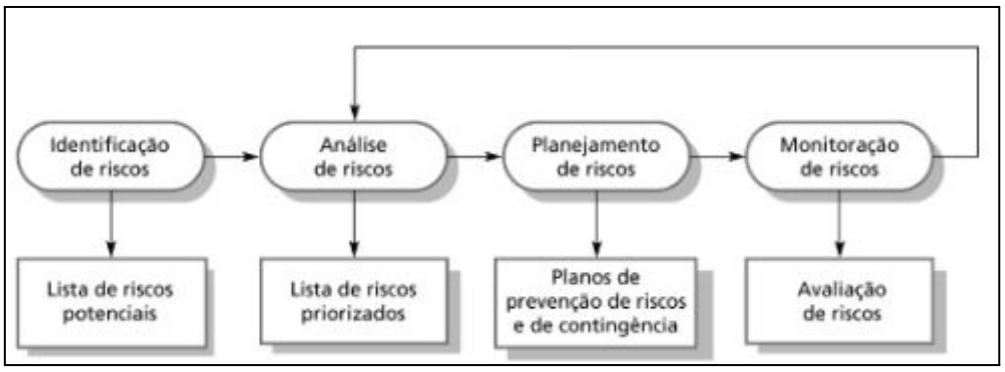


Figura 1 - O Processo de Gerenciamento de Riscos
(Adaptado de Sommerville, 2007)

        - **Identificação:** Listar todos os possíveis riscos (pessoal, tecnologia, etc.).
        - **Análise:** Classificar riscos por **probabilidade** e **impacto** (ex: baixo, médio, alto).
        - **Planejamento:** Criar estratégias para os riscos mais críticos (planos de **prevenção**, **minimização** e **contingência**).
        - **Monitoramento:** Acompanhar os riscos continuamente ao longo do projeto

Seção 3: Métodos Ágeis

- **Origem do Ágil:** Reação aos métodos tradicionais (cascata). Foco em **flexibilidade**, **colaboração** e **adaptação a mudanças,** Cohn [1].
- Scrum (Framework Principal):
    - **Ciclos:** Trabalho feito em **Sprints** (iterações curtas de 2-4 semanas).
    - **Papéis: Product Owner** (o que/por que - prioriza o backlog), **Scrum Master** (como - facilita o processo), **Equipe Dev** (constrói).
    - **Eventos:** O ciclo é Planejamento > Scrum Diário > Revisão > Retrospectiva.
- **Vantagens do Ágil:**
    - **Velocidade e Flexibilidade:** Entrega rápida de valor e fácil adaptação a novas necessidades.
    - **Menor Risco e Maior Qualidade:** Feedback contínuo reduz riscos e testes integrados melhoram a qualidade do produto

Seção 4: Tendências Atuais: A Inteligência Artificial aplicada à Gerência de Projetos

- Embora a Inteligência Artificial (IA) exista desde os anos 50, sua popularidade na GP cresceu recentemente devido à expansão da tecnologia e ao enorme volume de dados disponíveis (*Big Data*), Ong et al. [2]

- **Objetivo Principal:** O uso de IA e Ciência de Dados na gestão de projetos visa simplificar processos e gerar maior eficiência na entrega.
- **Aplicações Práticas:**
    - **Automação** de tarefas, **otimização** de custo/tempo e **análise preditiva** de cronogramas com *machine learning*.
- **Futuro da Área:**
    - Análise de "Big Data" para otimizar planejamento, risco e alocação de recursos.
    - Tendência de **substituição de tarefas humanas**, com a IA potencialmente excedendo as capacidades de gestão em algumas décadas.

Seção 5: Conclusão

- **Ideia Central:** Reafirmar que a Gestão de Projetos (GP) de Software é uma disciplina **essencial**, **complexa** e em **constante evolução**.
- **Síntese da Evolução:**
    - Recapitular a jornada: da base **tradicional** (planejamento, risco) , passando pela revolução **Ágil** (Scrum, flexibilidade) , até o futuro com **IA** (automação, predição).

Perfil do Gerente Moderno:

- Concluir que o profissional de sucesso hoje deve ser **híbrido** e **adaptável**, combinando fundamentos clássicos, agilidade e visão tecnológica. O **aprendizado contínuo** é a chave.

Referências

[1] COHN, M. Desenvolvimento de Software com Scrum: aplicando métodos ágeis com sucesso. Porto Alegre: Bookman, 2011.
[2] ONG, Stephen; UDDIN, Shahadat. Data science and artificial intelligence in project management: the past, present, and future. *The Journal of Modern Project Management*, 2020, 7.4.
[3] SOMMERVILLE, Ian. Engenharia de Software. 8a ed., São Paulo, Addison-Wesley, 2007.