

## Unidade 1

### Introdução à Engenharia de Software

#### Aula 1

##### Introdução à engenharia de software

### Introdução à engenharia de software

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

##### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá os fundamentos essenciais da engenharia de software, abordando princípios essenciais, a evolução e os desafios no cenário atual. A compreensão profunda desses temas é vital para sua prática, capacitando-o a enfrentar os desafios da evolução do software e aplicar modelos de processos eficientes. Prepare-se para uma jornada enriquecedora, no qual cada conhecimento compartilhado contribuirá diretamente para a excelência em sua atuação profissional. Vamos começar a desvendar os segredos da engenharia de software!

### Ponto de Partida

Nesta aula, mergulharemos nos fundamentos da Engenharia de Software, uma disciplina crucial para o desenvolvimento de sistemas computacionais eficientes e confiáveis. Esta área da tecnologia da informação é sustentada por uma série de princípios que não apenas orientam os profissionais no processo de desenvolvimento, mas também garantem a manutenção e a escalabilidade do software. Vamos explorar alguns desses princípios, compreendendo como eles são aplicados na prática e qual o impacto deles no ciclo de vida do desenvolvimento de software.

Além disso, abordaremos a evolução histórica da Engenharia de Software, destacando o período conhecido como "crise do software" nas décadas de 1960 e 1970, quando a indústria de software enfrentou desafios significativos em termos de entrega de projetos dentro do prazo, do orçamento e com a qualidade esperada. Esse período foi marcado por uma demanda crescente por software e uma falta de metodologias eficientes para gerenciá-lo, o que levou ao surgimento de vários modelos de processos de desenvolvimento de software. Esses modelos, como a cascata, incremental e espiral, surgiram como respostas a esses desafios, cada um com suas próprias características, vantagens e desvantagens. Vamos analisar esses modelos, entendendo como cada um deles contribui para a solução dos problemas enfrentados pela indústria de software e como eles são aplicados atualmente no desenvolvimento de software.

Para fixar o conteúdo, vamos fazer a seguinte atividade: você deverá realizar uma análise aprofundada dos modelos de processos de desenvolvimento de software – Cascata, Espiral e Incremental – mediante uma abordagem comparativa, envolvendo pesquisa, estudo de caso e elaboração de um relatório.

Comece com uma pesquisa detalhada sobre cada um dos modelos mencionados. Foque nas características-chave, vantagens e desvantagens de cada modelo. Após a pesquisa, escolha um projeto de software, real ou hipotético, e analise como ele seria desenvolvido sob cada um dos modelos. Considere aspectos como as etapas do projeto, a participação do cliente, a gestão de riscos e a capacidade de adaptação a mudanças.

Em seguida, elabore um relatório que compila suas descobertas. Este relatório deve incluir uma seção de discussão, em que você avalia qual modelo seria mais adequado para o projeto escolhido, com justificativas para sua escolha. Baseie suas conclusões na aplicabilidade de cada modelo em diferentes contextos de projetos de software.

Bons estudos!

## Vamos Começar!

## Princípios da engenharia de software

Existem várias definições de Engenharia de Software, a maioria destacando a importância de processos, métodos e ferramentas para garantir a qualidade do software desenvolvido. Se a aplicação desses elementos não resultar em um produto de alta qualidade, todo o esforço pode ser considerado inútil. Segundo Pressman (2021), a Engenharia de Software é definida como um processo que inclui uma série de métodos e ferramentas que permitem aos profissionais criar software de excelente qualidade. É crucial reconhecer que a Engenharia de Software é um campo dinâmico, em que a relevância e as exigências dos programas de computador têm evoluído significativamente ao longo do tempo. A demanda por qualidade e rapidez nas entregas aumentou, levando ao desenvolvimento de novas abordagens na criação de software. Portanto, compreender essa engenharia é essencial para implementar práticas que atendam às necessidades das organizações nas quais os profissionais de TI operam.

Dominar um conceito permite sua adaptação às necessidades individuais ou da organização, preservando sua essência fundamental. Neste contexto, uma compreensão aprofundada de Engenharia de Software exige a exploração dos termos que a constituem. "Engenharia" se refere ao design e à produção de um artefato, em que requisitos e especificações são vitais. Diferentemente dos artefatos físicos, os programas de computador não seguem um processo de manufatura convencional. Este campo envolve desafios únicos, necessitando de soluções específicas distintas dos métodos industriais tradicionais.

Avançando para a definição de software, podemos considerá-lo como:

- Instruções executáveis que realizam funções específicas.
- Estruturas de dados que permitem a manipulação de informações pelos programas.
- Documentação relacionada ao sistema.

Esses componentes formam a base da Engenharia de Software, que, segundo a IEEE *Computer Society*, é a prática de aplicar abordagens sistemáticas, disciplinadas e quantificáveis para o desenvolvimento, a operação e a manutenção de software, e inclui o estudo desses métodos (IEEE, 2004).

Estabelecendo a base teórica da Engenharia de Software, é essencial reconhecer os princípios que definem e sustentam suas práticas. Estes princípios, delineados pelo (*Software Engineering Body of Knowledge*) – SWEBOK – uma publicação notável do IEEE, são fundamentais para estruturar os processos e padronizar as soluções dentro deste campo da computação.

O IEEE identifica a “organização hierárquica” como um princípio crucial. Este conceito sugere a apresentação de componentes de uma solução ou a organização de elementos de um problema em um formato hierárquico, detalhado progressivamente em cada nível.

A “formalidade” segue como outro princípio importante, enfatizando uma abordagem rigorosa e padronizada na resolução de problemas. Em seguida, vem a “completeza”, que ressalta a importância de abordar todos os elementos de um problema e garantir que a solução proposta os contemple integralmente.

O princípio de “dividir para conquistar”, segundo o IEEE, orienta a divisão de problemas complexos em partes menores e gerenciáveis, fundamentando a criação de soluções modulares. Este princípio é complementado pela “ocultação”, que defende que cada módulo deve ter acesso apenas às informações necessárias para sua operação, ocultando o que é desnecessário.

O princípio da “localização” sugere agrupar itens logicamente relacionados em um sistema, enquanto a “integridade conceitual” propõe que os engenheiros de software sigam uma filosofia e arquitetura de projeto consistentes.

Por fim, o SWEBOK destaca a “abstração”, que se concentra em isolar aspectos essenciais de um problema específico, permitindo ao profissional focar no que é mais relevante, enquanto adia questões secundárias ou relacionadas para um momento posterior. Esta abordagem prática

enfatiza a habilidade do engenheiro de software em concentrar-se no essencial para solucionar um problema em particular.

## Evolução e crise de software

A essência da Engenharia de Software é focada em seu elemento central: o software. Conforme definido por Pressmann e Maxim (2016), o software de computador é tanto o produto criado quanto o suporte proporcionado pelos profissionais da área. Este produto não se limita apenas aos programas executáveis em um computador, mas também inclui os dados e as informações geradas durante a execução desses programas, assim como a documentação necessária para referência e manutenção futura.

Este conceito ampliado de software destaca que ele vai além do código executável; inclui também os dados processados e a documentação essencial que o acompanha. O software atua não só como um produto, mas também como um meio para distribuir este produto, desempenhando um papel fundamental na transmissão de informações, um recurso valioso na era moderna. Desde a segunda metade do século XX, o papel do software evoluiu de forma significativa, estabelecendo-se como um componente crucial em diversas esferas da vida contemporânea, conforme observado por Pressman e Maxim (2016).

A conceituação do software, quando simplificada e vista como um produto viável no mercado, pode enganosamente sugerir que sua criação seja uma tarefa simples e descomplicada. Essa percepção simplista contrasta fortemente com a realidade, em que o desenvolvimento de software evoluiu de uma atividade artesanal e empírica para um processo sistematizado, estruturado e metodológico, impulsionado pelo avanço da Engenharia de Software. Contudo, nos estágios iniciais dessa evolução, a produção de software enfrentou períodos desafiadores.

Conforme Schach (2009), a expressão "crise do software", cunhada na década de 1960, reflete um período difícil para o desenvolvimento de software, caracterizado pela incerteza e pela incapacidade de atender à crescente demanda. Os programas frequentemente eram ineficientes, com processos de desenvolvimento falhos e dificuldade de manutenção. A imprecisão nas estimativas de custo e tempo minava a confiança das equipes e dos clientes.

A comunicação deficiente entre clientes e equipes de desenvolvimento naquela época levava a um levantamento de requisitos inadequado, resultando em produtos finais cheios de falhas. A falta de métricas confiáveis para avaliar a adequação dos produtos entregues exacerbava o problema. Após a entrega dos programas, a fase de implementação era muitas vezes caótica, negligenciando os impactos organizacionais e o treinamento dos usuários. A manutenção dos sistemas tornava-se um desafio adicional, frequentemente devido a projetos mal planejados. Este cenário turbulento da produção de software na década de 1960 ilustra as dificuldades enfrentadas antes da consolidação de práticas eficientes na Engenharia de Software.

Daquela época ficaram alguns mitos sobre o desenvolvimento de um software. Embora os profissionais atuais estejam mais capacitados a reconhecê-los e a evitarem seus efeitos, ainda é necessário dar atenção a eles. Estes incluem:

- Adicionar mais programadores não acelera necessariamente o desenvolvimento: Mais desenvolvedores podem complicar a comunicação e aumentar a complexidade do projeto.
- Software não é facilmente modificável após a construção: mudanças tardias podem ser custosas e difíceis.
- Software funcionando não significa que está pronto: testes, validação e documentação são essenciais antes da entrega.
- Software requer manutenção constante: é um produto dinâmico que necessita de atualizações e melhorias contínuas.
- Engenharia de software vai além da programação: inclui planejamento, análise, design, testes e manutenção.
- Habilidades de programação não equivalem a habilidades de gerenciamento: gerenciar um projeto de software requer habilidades específicas em gestão.

Reconhecer e abordar esses mitos é crucial para uma gestão eficaz de projetos de software e para o estabelecimento de expectativas realistas.

## Siga em Frente...

## Modelos de processos de desenvolvimento de software

Um modelo de processo de software é essencialmente um conjunto de práticas e procedimentos que guiam as equipes de desenvolvimento na produção de softwares. Ele serve como um roteiro abrangente que descreve as etapas necessárias para levar um projeto de software desde a concepção até a entrega e manutenção, garantindo que todos os aspectos essenciais sejam abordados de forma sistemática e organizada.

A função principal de um modelo de processo de software é oferecer uma estrutura que ajuda a organizar e controlar as várias atividades envolvidas no desenvolvimento de software, como análise de requisitos, design, codificação, testes e manutenção. Este modelo busca assegurar a qualidade do software, a eficiência no processo de desenvolvimento e a satisfação das necessidades do cliente.

O modelo é adaptado para atender às características específicas de cada projeto, considerando fatores como o tamanho da equipe, a complexidade do software, o orçamento disponível e os prazos de entrega. Ele fornece um método para rastrear o progresso do projeto, identificar riscos e lidar com mudanças de maneira eficaz.

O processo de software, por sua vez, é a aplicação prática de um modelo de processo de software. Ele envolve a execução real das atividades de desenvolvimento de software dentro do quadro fornecido pelo modelo escolhido. O processo de software inclui a definição de tarefas, a alocação de recursos, a gestão do tempo e a coordenação entre as diferentes equipes e atividades.

Durante o processo de software, as equipes trabalham em conjunto para definir os requisitos do projeto, desenhar a arquitetura do software, escrever e testar o código, e, finalmente, entregar um produto que atenda às expectativas do cliente. Após a entrega, o software geralmente passa por uma fase de manutenção e atualização, garantindo que continue a funcionar eficientemente e permaneça relevante ao longo do tempo.

Em resumo, enquanto o modelo de processo de software fornece o quadro teórico, o processo de software é a implementação prática desse quadro, adaptado às necessidades específicas de cada projeto. Juntos, eles desempenham um papel crucial na produção de software de alta qualidade, permitindo que as equipes de desenvolvimento naveguem com sucesso nos desafios complexos da engenharia de software.

## Modelo Cascata

O Modelo Cascata, também conhecido como Ciclo de Vida Clássico, é um dos modelos mais antigos e tradicionais de processo de desenvolvimento de software. Este modelo caracteriza-se por sua abordagem linear e sequencial, em que cada fase do processo deve ser concluída antes da próxima fase começar. A Figura 1 mostra o esquema geral das fases do modelo cascata.

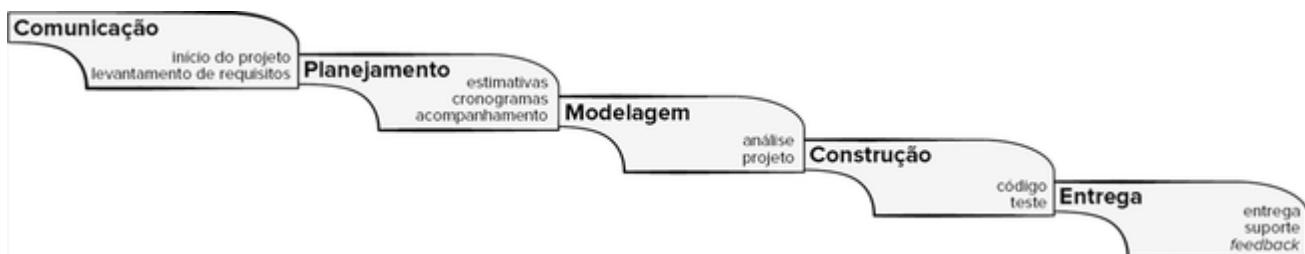


Figura 1 | Modelo Cascata. Fonte: Pressman (2021).

As principais fases do Modelo Cascata incluem:

- **Comunicação:** esta fase inicial envolve a interação direta com o cliente para entender suas necessidades e requisitos. É um momento crítico em que informações vitais sobre o projeto são coletadas e documentadas. A eficácia da comunicação nesta etapa é crucial para o sucesso do projeto, pois assegura que as necessidades do cliente sejam claramente compreendidas.
- **Planejamento:** após a coleta de requisitos, inicia-se o planejamento do projeto. Esta etapa envolve a definição de recursos, os cronogramas, os custos e a identificação de potenciais riscos. O planejamento detalhado é essencial para guiar as fases subsequentes e assegurar que o projeto permaneça no caminho certo.
- **Modelagem:** nesta fase, o foco está no design do sistema e do software. Com base nos requisitos coletados, são elaborados modelos e esquemas que representam a arquitetura do software, incluindo a estrutura de dados, a arquitetura do sistema, os algoritmos e as interfaces de usuário. Esta etapa é fundamental para transformar os requisitos em uma representação concreta do que será desenvolvido.

- **Construção:** com os modelos e o design em mãos, começa a fase de construção, ou seja, a codificação do software. Aqui, os desenvolvedores implementam as funcionalidades definidas nos modelos, criando um produto funcional. Esta fase também inclui testes iniciais para garantir a qualidade do código e a conformidade com as especificações.
- **Entrega:** após a construção, o software passa por uma fase de testes mais rigorosos para identificar e corrigir falhas. Uma vez que o software é considerado estável e pronto, ele é entregue ao cliente. Esta fase pode também incluir a implantação do software no ambiente do cliente, o treinamento de usuários e a transição para equipes de suporte e manutenção.

O Modelo Cascata, estruturado nessa sequência linear de fases, é bastante eficaz para projetos com requisitos bem definidos e estáveis. No entanto, é menos flexível para acomodar mudanças de requisitos que podem surgir durante o desenvolvimento, o que é uma limitação importante desse modelo.

## Modelo Espiral

No Modelo Espiral, o desenvolvimento do software acontece em incrementos mediante uma série de iterações, conhecidas como 'espirais'. Cada espiral no modelo representa uma fase do processo e inclui quatro passos fundamentais: definição de objetivos, análise e resolução de riscos, desenvolvimento efetivo e planejamento da próxima iteração. A cada ciclo, o software evolui, incorporando mais elementos e funcionalidades, ao mesmo tempo que são identificados e geridos os riscos associados. A Figura 2, apresenta esse tipo de modelo.

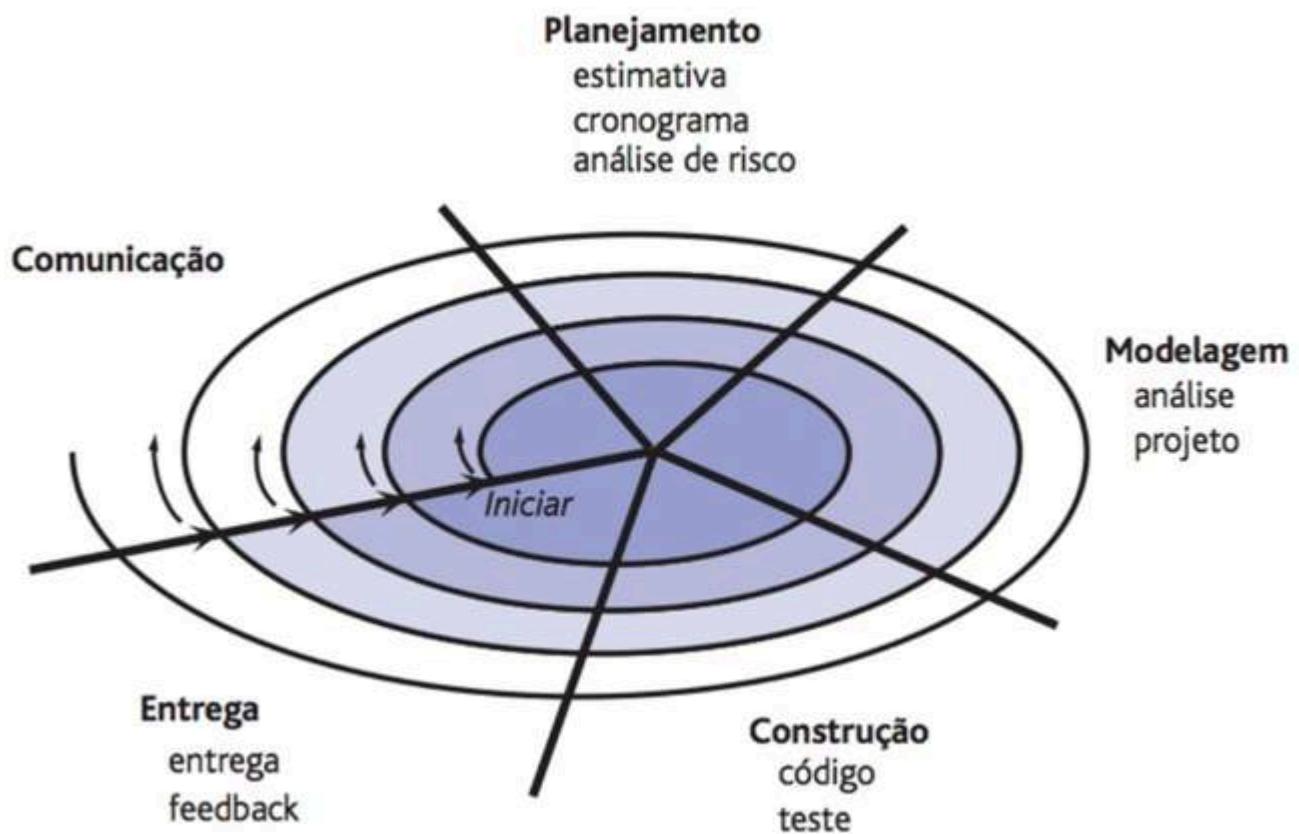


Figura 2 | Modelo Espiral. Fonte: Pressman (2016).

Um aspecto distintivo do modelo espiral é a sua capacidade de incorporar protótipos. Estes são criados e testados para validar requisitos e funcionalidades, permitindo ajustes e refinamentos antes de avançar para a próxima fase. Este processo de feedback contínuo, envolvendo clientes ou usuários finais, assegura que o produto final atenda às expectativas e necessidades reais.

O modelo permite uma adaptação e personalização significativas em cada etapa. Baseado na análise de riscos e no feedback dos usuários, as fases subsequentes podem ser ajustadas para melhor atender às demandas do projeto. Embora ofereça muitas vantagens, especialmente em termos de flexibilidade e gestão de riscos, o Modelo Espiral pode ser mais oneroso e complexo de gerir, em comparação com abordagens mais simples de desenvolvimento de software.

Em resumo, o Modelo Espiral é uma abordagem versátil e orientada a riscos para o desenvolvimento de software, altamente adequada para projetos em que os requisitos não são completamente conhecidos desde o início ou que envolvem muita incerteza e complexidade.

## Modelo Incremental

O modelo incremental é uma abordagem de desenvolvimento de software no qual o processo é dividido em várias iterações, e o software é construído e entregue em incrementos. Cada incremento é uma versão funcional do software, que adiciona uma funcionalidade ou um

conjunto de funcionalidades ao produto final. Esta abordagem permite que os desenvolvedores entreguem uma parte do software operacional em estágios iniciais do projeto, fornecendo uma forma eficaz de demonstrar a funcionalidade e coletar feedback do usuário final.

A ideia principal por trás do modelo incremental é desenvolver um sistema por meio de adições graduais, com cada incremento construído sobre os anteriores. Inicialmente, é feita uma análise básica dos requisitos e o projeto é iniciado com um conjunto fundamental de funcionalidades. Após a entrega do primeiro incremento, o software é avaliado, e novos requisitos ou melhorias são identificados. Subsequentemente, um novo incremento é planejado, desenvolvido e integrado ao sistema existente, e o ciclo continua. A Figura 3 apresenta esse modelo.

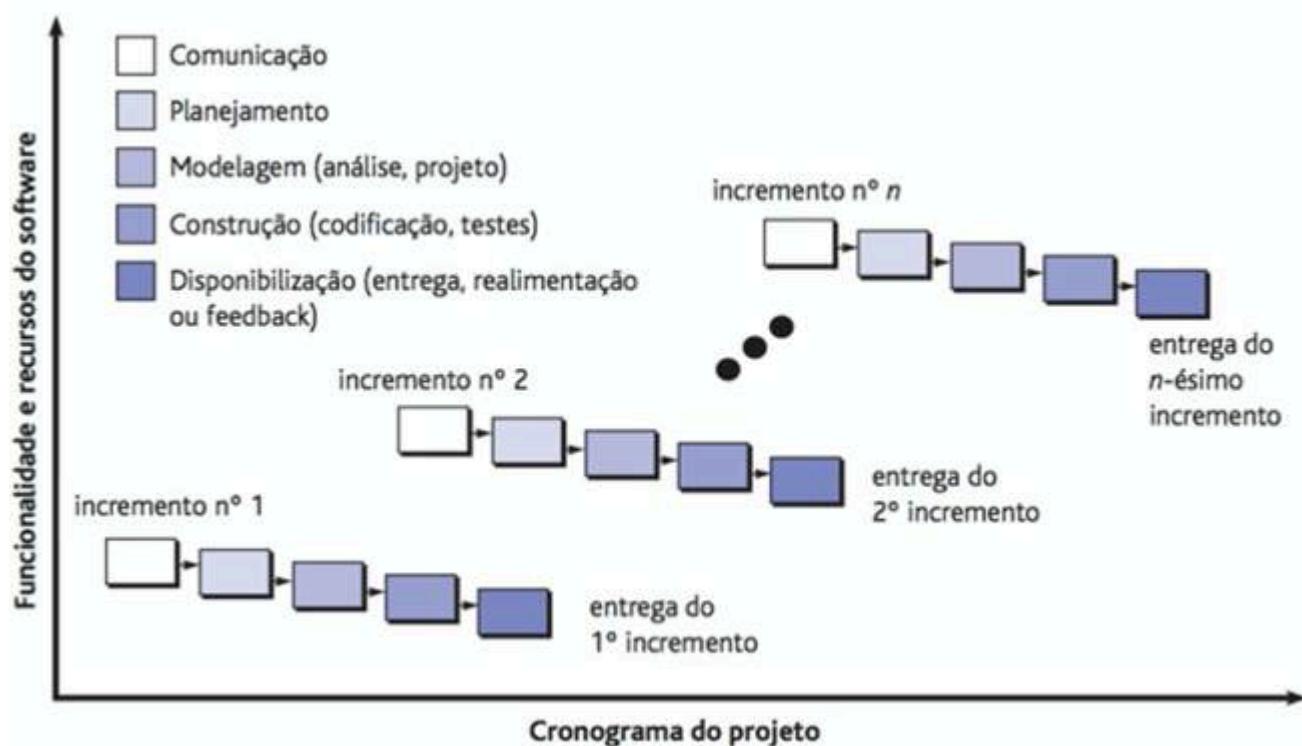


Figura 3 | Modelo Incremental. Fonte: Pressman (2016).

Uma vantagem significativa do modelo incremental é a capacidade de ajustar o software a mudanças de requisitos ao longo do desenvolvimento. Isso é particularmente útil em projetos em que não é possível ou prático definir todos os requisitos de antemão. Além disso, a entrega antecipada de uma versão funcional do software pode fornecer um retorno sobre o investimento mais cedo, além de permitir uma verificação contínua da adequação do produto às necessidades dos usuários.

Essa abordagem também facilita o teste e a depuração em cada estágio do desenvolvimento, já que cada incremento é menor e mais gerenciável. Isso pode levar a uma detecção mais rápida de problemas e a um software mais confiável. No entanto, o modelo incremental requer um planejamento cuidadoso e uma boa gestão para garantir que cada incremento se integre de

maneira coesa ao sistema como um todo, evitando problemas de compatibilidade ou inconsistências.

## Vamos Exercitar?

Esta atividade avalia a compreensão dos diferentes modelos de processos de desenvolvimento de software e a habilidade de aplicar esse conhecimento na análise de um projeto específico.

### Pesquisa sobre os Modelos

- Modelo Cascata: caracteriza-se por sua natureza linear e sequencial. É eficaz quando os requisitos são bem compreendidos e estáveis. As mudanças são difíceis de implementar uma vez que uma etapa é concluída.
- Modelo Espiral: combina elementos iterativos com avaliações de riscos em cada ciclo. É flexível e permite a incorporação contínua de feedback do cliente. Ideal para projetos grandes e de alto risco.
- Modelo Incremental: desenvolve o software em incrementos ou versões, facilitando a gestão de mudanças e a entrega rápida de partes funcionais do software.

### Estudo de Caso

- Escolha de um projeto de software real ou hipotético.
- Análise de como o projeto seria abordado sob cada modelo.
- Avaliação das implicações em termos de requisitos, participação do cliente, gestão de riscos e adaptabilidade.

### Elaboração do Relatório

- Compilação das descobertas da pesquisa e da análise do estudo de caso.
- Discussão sobre qual modelo seria mais apropriado para o projeto escolhido, com justificativas baseadas nas características e aplicabilidade de cada modelo.

### Conclusão do Relatório

- O modelo mais adequado depende das especificidades do projeto, como complexidade, clareza dos requisitos, riscos envolvidos e necessidade de adaptabilidade.
- Justificativa da escolha baseada na análise comparativa, destacando a adequação do modelo ao contexto do projeto escolhido.

## Saiba mais

Leia mais sobre a natureza e a definição do software em: [Engenharia de Software \(PRESSMAN, 2018\)](#), Capítulo 1, seção 1.1 e 1.2.

Quer compreender mais sobre a evolução de software? Leia o seguinte artigo: [Evolução de Software](#), de Douglas José Peixoto de Azevedo, em Batebyte.

O livro [Metodologias Ágeis - Engenharia de Software sob Medida](#), de José Henrique Teixeira de Carvalho Sbrocco e Paulo Cesar de Macedo, apresenta mais sobre as metodologias ágeis, além de apresentar outros métodos.

## Referências

IEEE Computer Society. **Guide to the Software Engineering Body of Knowledge**. Piscataway: The Institute of Electrical and Electronic Engineers, 2004.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software**: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

SCHACH, S. R. **Engenharia de software**: os paradigmas clássicos e orientados a objetos. 7. ed. São Paulo: McGraw-Hill, 2009.

## Aula 2

Métodos ágeis

### Métodos ágeis

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá as características fundamentais dos métodos ágeis, concentrando-nos no Scrum e XP. Entender esses princípios é essencial para sua prática

profissional, oferecendo agilidade, flexibilidade e eficiência no desenvolvimento de projetos. O Scrum proporciona uma abordagem estruturada, enquanto o XP enfatiza práticas ágeis. Estes métodos revolucionam a dinâmica de trabalho, capacitando você a enfrentar os desafios contemporâneos com sucesso. Prepare-se para mergulhar no universo da agilidade e transformar sua prática profissional!

## Ponto de Partida

Nesta aula abordaremos as características fundamentais dos métodos ágeis, uma abordagem revolucionária no desenvolvimento de software que enfatiza flexibilidade, colaboração em equipe, e a capacidade de adaptar-se rapidamente às mudanças. Diferentemente dos métodos tradicionais, que seguem um plano rígido e linear, os métodos ágeis permitem uma abordagem mais iterativa e incremental, focada na entrega rápida de produtos de alta qualidade que atendam às necessidades em constante evolução dos clientes. Esta abordagem é particularmente valiosa em um ambiente de negócios que está sempre mudando, em que a rapidez e a flexibilidade são essenciais para o sucesso.

Dentro dos métodos ágeis, o Scrum se destaca como um dos mais populares frameworks utilizados por equipes de desenvolvimento de software em todo o mundo. O Scrum é notável por sua estrutura que organiza o trabalho em ciclos curtos e regulares conhecidos como Sprints, cada um resultando em um incremento do produto final. Este framework é caracterizado por papéis bem definidos, como o Scrum Master e o Product Owner, e cerimônias específicas. Estes elementos trabalham juntos para garantir que o trabalho seja feito de forma eficiente, mantendo a equipe focada e alinhada com os objetivos do projeto.

Por outro lado, a *Extreme Programming* (XP) é outra metodologia ágil popular que se concentra mais intensamente nas práticas de engenharia de software e na qualidade do código. XP é conhecida por suas práticas como programação em pares, desenvolvimento orientado a testes, integração contínua e refatoração. Essas práticas garantem a qualidade do software e permitem que as equipes respondam rapidamente às mudanças nos requisitos dos clientes.

Para reforçar este conteúdo, nesta atividade, você terá a oportunidade de aplicar os princípios do Scrum para organizar e gerenciar suas atividades diárias e de estudo. Seu objetivo é criar um Quadro Scrum pessoal que o ajudará a visualizar as tarefas, priorizá-las e acompanhar seu progresso de forma eficiente.

Identifique suas tarefas: liste todas as suas atividades e tarefas pendentes que você precisa realizar nas próximas duas semanas. Isso pode incluir tarefas relacionadas aos estudos, compromissos pessoais, ou quaisquer outras responsabilidades importantes.

Crie o quadro Scrum: utilize um quadro branco, uma folha de papel grande, ou uma ferramenta digital para criar o seu quadro Scrum. Divida o quadro em pelo menos três colunas: "A Fazer" (*To Do*), "Em Andamento" (*Doing*) e "Concluído" (*Done*). Caso você prefira, pode adicionar mais colunas ou até mesmo mudar o nome sugerido.

- **Organize as tarefas:** coloque cada tarefa listada na coluna "A Fazer". As tarefas devem ser escritas em pequenos pedaços de papel ou *post-its* para que possam ser movidas facilmente entre as colunas.
- **Priorize e planeje:** avalie cada tarefa e decida a ordem de prioridade. Planeje iniciar com as tarefas mais urgentes ou importantes.
- **Execute e atualize:** à medida que você trabalha em suas tarefas, mova-as para a coluna "Em Andamento". Quando uma tarefa for concluída, mova-a para a coluna "Concluído".
- **Reflita e ajuste:** ao final das duas semanas, reflita sobre o processo. Que tarefas foram concluídas? Quais tarefas permaneceram inacabadas e por quê? Use essas informações para ajustar seu planejamento e sua metodologia para as próximas semanas.

## Vamos Começar!

## Características dos métodos ágeis

No ano de 2001, um grupo notável de desenvolvedores de software, escritores e consultores elaborou e endossou o *Manifesto para o Desenvolvimento Ágil de Software*. Este manifesto destacava princípios centrais para uma abordagem mais eficiente e adaptável no desenvolvimento de software. Eles priorizavam indivíduos e interações mais do que processos e ferramentas, software funcional mais do que documentação abrangente, colaboração com clientes mais do que negociações contratuais, e adaptabilidade a mudanças mais do que a rigidez de seguir um plano (Sbrocco, 2012). Este manifesto representou um ponto de virada significativo na maneira como o software é desenvolvido, enfatizando a flexibilidade, colaboração e adaptabilidade no lugar de estruturas e processos rígidos.

Os princípios que fundamentam o desenvolvimento ágil resultaram na criação dos métodos ágeis, concebidos para abordar tanto as falhas percebidas quanto as reais da engenharia de software tradicional. O desenvolvimento ágil traz vantagens significativas, mas não é universalmente adequado para todas as situações, projetos, equipes ou contextos. Importante frisar que ele não se opõe às práticas sólidas de engenharia de software e pode ser integrado como uma abordagem ampla no trabalho com software.

No cenário econômico atual, prever a trajetória de evolução de um sistema computacional, como um aplicativo móvel, é frequentemente desafiador. Mudanças no mercado ocorrem rapidamente, as necessidades dos usuários evoluem e novas competições emergem inesperadamente. Em muitos casos, os requisitos de um projeto não podem ser totalmente definidos antes de seu início. Neste contexto, a agilidade é essencial para se adaptar a um ambiente de negócios em constante transformação.

Mudanças são inerentes e podem ser onerosas, especialmente se mal administradas ou descontroladas. Uma das maiores vantagens da metodologia ágil é a sua capacidade de minimizar os custos associados às mudanças no processo de desenvolvimento de software.

Os processos ágeis de desenvolvimento de software são moldados por um conjunto de princípios fundamentais que destacam a natureza mutável da maioria dos projetos de software. Primeiramente, é desafiador prever quais requisitos de software permanecerão constantes e quais mudarão, assim como as prioridades dos clientes podem evoluir durante o projeto. Além disso, para muitos tipos de software, o design e a construção são processos que ocorrem simultaneamente, o que implica que os modelos de projeto devem ser testados à medida que são desenvolvidos. Não é possível prever com precisão a quantidade de trabalho de design necessário antes da construção para avaliar o projeto.

Diante desses preceitos, surge a questão de como gerenciar a imprevisibilidade inherente aos projetos de software. A resposta está na capacidade de adaptar rapidamente o projeto e as condições técnicas do processo. Porém, a adaptação contínua, sem avanço significativo, é inútil. Assim, um processo ágil deve evoluir de maneira incremental, requerendo feedback constante do cliente para assegurar que as adaptações sejam pertinentes.

Uma estratégia eficaz para obter esse feedback é mediante protótipos operacionais ou partes funcionais do sistema. A entrega de incrementos de software, sejam protótipos ou partes funcionais, deve ocorrer em intervalos curtos para que as adaptações acompanhem o ritmo das mudanças. Esse método iterativo permite ao cliente avaliar regularmente o progresso do software, fornecer feedback crucial à equipe de desenvolvimento e influenciar as modificações no processo, de modo a incorporar adequadamente as opiniões e necessidades do cliente.

A Agile Alliance estabelece uma série de princípios para a obtenção de agilidade no desenvolvimento de software. Estes princípios enfatizam a importância de satisfazer o cliente por meio de entregas rápidas e valiosas de software. Os desenvolvedores ágeis reconhecem que os requisitos mudarão ao longo do tempo, e, por isso, eles entregam incrementos de software frequentemente e mantêm uma comunicação próxima e significativa com todas as partes interessadas.

As equipes ágeis são formadas por indivíduos motivados, operando em um ambiente que favorece a comunicação face a face e o desenvolvimento de software de alta qualidade. Estas equipes seguem processos que incentivam a excelência técnica e um bom design, priorizando a simplicidade, que é vista como uma forma de reduzir o trabalho desnecessário. O objetivo principal é entregar um software operacional que atenda às necessidades do cliente, mantendo um ritmo de trabalho sustentável a longo prazo.

Equipes ágeis são auto-organizadas, capazes de desenvolver arquiteturas sólidas que resultam em projetos de sucesso e na satisfação do cliente. Uma parte essencial da cultura da equipe é a reflexão contínua sobre o trabalho, buscando constantemente aprimorar a maneira de atingir seus objetivos (Pressman, 2021).

É importante notar que nem todos os modelos de processo ágil dão o mesmo peso a cada um desses princípios, e alguns podem até mesmo optar por ignorar ou minimizar alguns deles. No entanto, esses princípios formam a essência do espírito ágil.

## SCRUM

Alinhado com os princípios do manifesto ágil, o Scrum direciona as atividades de desenvolvimento de software por meio de um processo que abrange várias atividades metodológicas essenciais, como requisitos, análise, projeto, evolução e entrega. Estas atividades são executadas em intervalos de tempo definidos, conhecidos como sprints. O número e a duração dos sprints variam conforme o tamanho e a complexidade do produto sendo desenvolvido.

Dentro de cada sprint, as tarefas são adaptadas ao problema específico e são definidas e frequentemente ajustadas pela equipe Scrum, conforme as necessidades do projeto. Este fluxo de trabalho flexível e adaptável é uma característica central do método Scrum, permitindo respostas rápidas e eficientes às mudanças durante o desenvolvimento do software. O fluxo geral do processo Scrum está ilustrado na Figura 1.

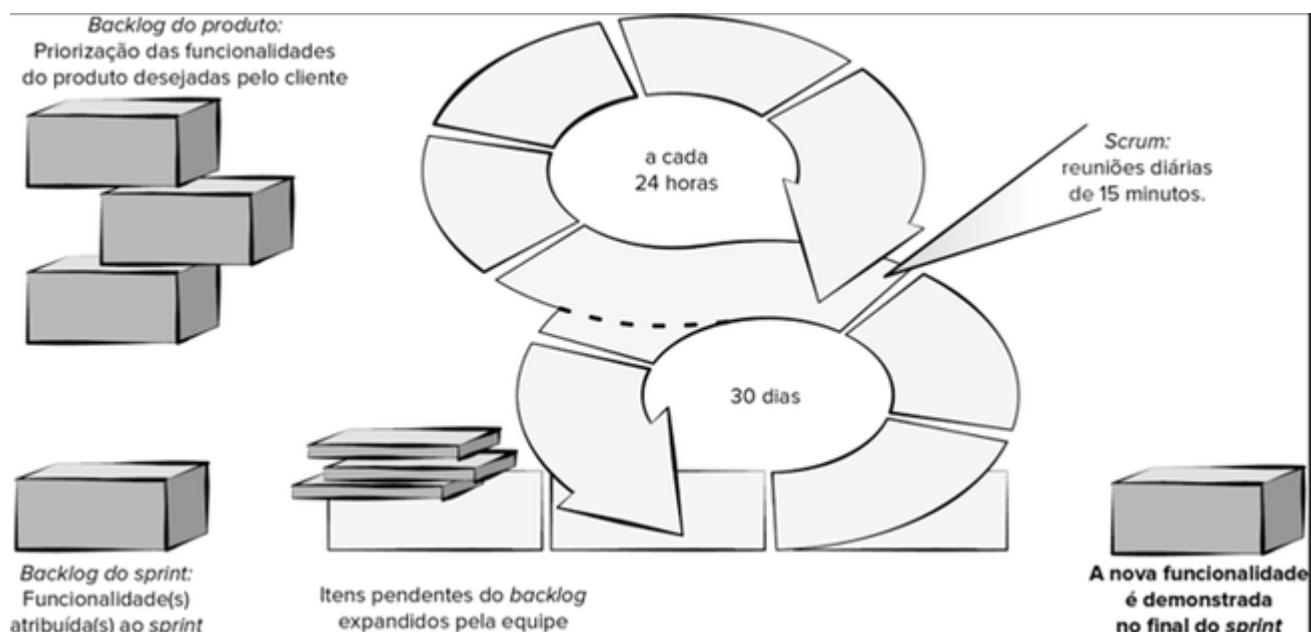


Figura 1 | Fluxo do processo Scrum Fonte: Pressman (2021).

Cada um dos elementos do ciclo é abordado na sequência (PRESSMAN, 2021):

- **Produto Backlog (*Product Backlog*)**: é uma lista dinâmica que abrange todas as funcionalidades desejadas para o produto no método Scrum. Esta lista não é imutável ou definitiva desde o início, mas sim evolutiva. Inicia-se com as funcionalidades mais claras e aparentes, sendo atualizada e expandida progressivamente com novas funcionalidades que emergem à medida que o projeto avança.
- **Backlog da Sprint (*Sprint Backlog*)**: é uma relação de tarefas que a equipe se compromete a realizar durante um Sprint específico. Esta lista é composta selecionando tarefas do *Product Backlog*, priorizadas pelo *Product Owner*, que serão abordadas no decorrer do Sprint.

- **Sprint:** é a fase no método Scrum em que ocorre a construção ativa do software, dividida em ciclos fixos, geralmente de duas a quatro semanas, mas podendo se estender conforme a complexidade das tarefas. Durante um Sprint, a equipe se dedica a desenvolver as funcionalidades listadas no *Backlog* da Sprint. Se novas funcionalidades surgirem, elas serão alocadas para o próximo Sprint. O *Product Owner* é responsável por manter o *Backlog* da Sprint atualizado, marcando tarefas concluídas e destacando as que ainda precisam ser finalizadas.

Embora o modelo Scrum defende que as equipes sejam auto-organizadas, ainda assim apresenta três perfis profissionais de relevância (Sbrocco, 2012):

- **Scrum Master:** é um facilitador essencial dentro do projeto Scrum, atuando como um especialista no modelo e garantindo sua aplicação correta em todas as fases do projeto. Ele também age como um moderador, assegurando que a equipe não se sobrecarregue com tarefas além de sua capacidade.
- **Product Owner:** é a figura central na definição do projeto, responsável por identificar e priorizar os requisitos críticos que serão abordados nos sprints.
- **Scrum Team:** é o grupo responsável pelo desenvolvimento, normalmente formado por seis a dez membros. Não há distinções rígidas de funções como programador, analista ou projetista.

As cerimônias do Scrum são reuniões essenciais que estruturam o fluxo de trabalho em um projeto Scrum. Estas cerimônias incluem (Sbrocco, 2012):

- **Sprint Planning (Planejamento do Sprint):** esta reunião marca o início de cada Sprint. Nela, o Product Owner e a equipe de desenvolvimento definem o que será trabalhado durante o Sprint. O Product Owner apresenta as prioridades do Product *Backlog* e, juntos, decidem quais itens serão incluídos no Sprint *Backlog*.
- **Daily Stand-up (Reunião Diária):** uma reunião rápida, normalmente com duração de cerca de 15 minutos, realizada todos os dias de um Sprint. O objetivo é que a equipe compartilhe o progresso, identifique possíveis impedimentos e planeje as atividades do dia. Cada membro responde geralmente a três perguntas: o que foi feito desde a última reunião, o que será feito até a próxima e se há algum obstáculo no caminho.
- **Sprint Review (Revisão do Sprint):** realizada no final de cada Sprint, esta reunião é uma demonstração do trabalho realizado. A equipe apresenta os incrementos de softwares desenvolvidos ao Product Owner e aos stakeholders, recebendo feedback e discutindo o que poderia ser melhorado ou adaptado.
- **Sprint Retrospective (Retrospectiva do Sprint):** após a Sprint Review e antes do próximo Sprint Planning, a equipe se reúne para discutir o que funcionou bem, o que poderia ser melhorado e como implementar essas melhorias no próximo Sprint. É um momento crucial para a melhoria contínua do processo de desenvolvimento.

Em ambientes onde as metodologias ágeis são aplicadas, é fundamental uma comunicação eficaz entre todos os participantes do projeto. Esta comunicação de qualidade não só envolve o cliente de forma integral e essencial, permitindo que ele compartilhe suas expectativas com

clareza, como também fortalece o vínculo entre os membros da equipe, garantindo que todos estejam alinhados com o progresso do projeto.

Suponha que você inicie sua carreira em uma empresa de desenvolvimento de software que adota o Scrum como sua metodologia de gerenciamento de projetos. Durante um dia de trabalho, você entra em uma sala e se depara com um quadro branco na parede, adornado com um arranjo matricial e vários *post-its* coloridos. Confuso, você pergunta a um colega sobre o propósito daquela configuração. Ele explica que o quadro é uma ferramenta visual usada pela equipe para monitorar as tarefas em andamento, bem como as que ainda precisam ser realizadas no projeto atual. Para que você possa reconhecer e entender melhor esse quadro em situações futuras, vamos explorar mais detalhes sobre sua estrutura e funcionalidade:

O quadro Scrum é uma ferramenta de gestão visual amplamente adotada pelas equipes para organizar as atividades de um projeto, apesar de não ser uma parte oficial do framework Scrum. Este quadro é estruturado de forma matricial, com colunas e linhas representando diferentes aspectos das tarefas. A Figura 2, apresenta esse quadro:

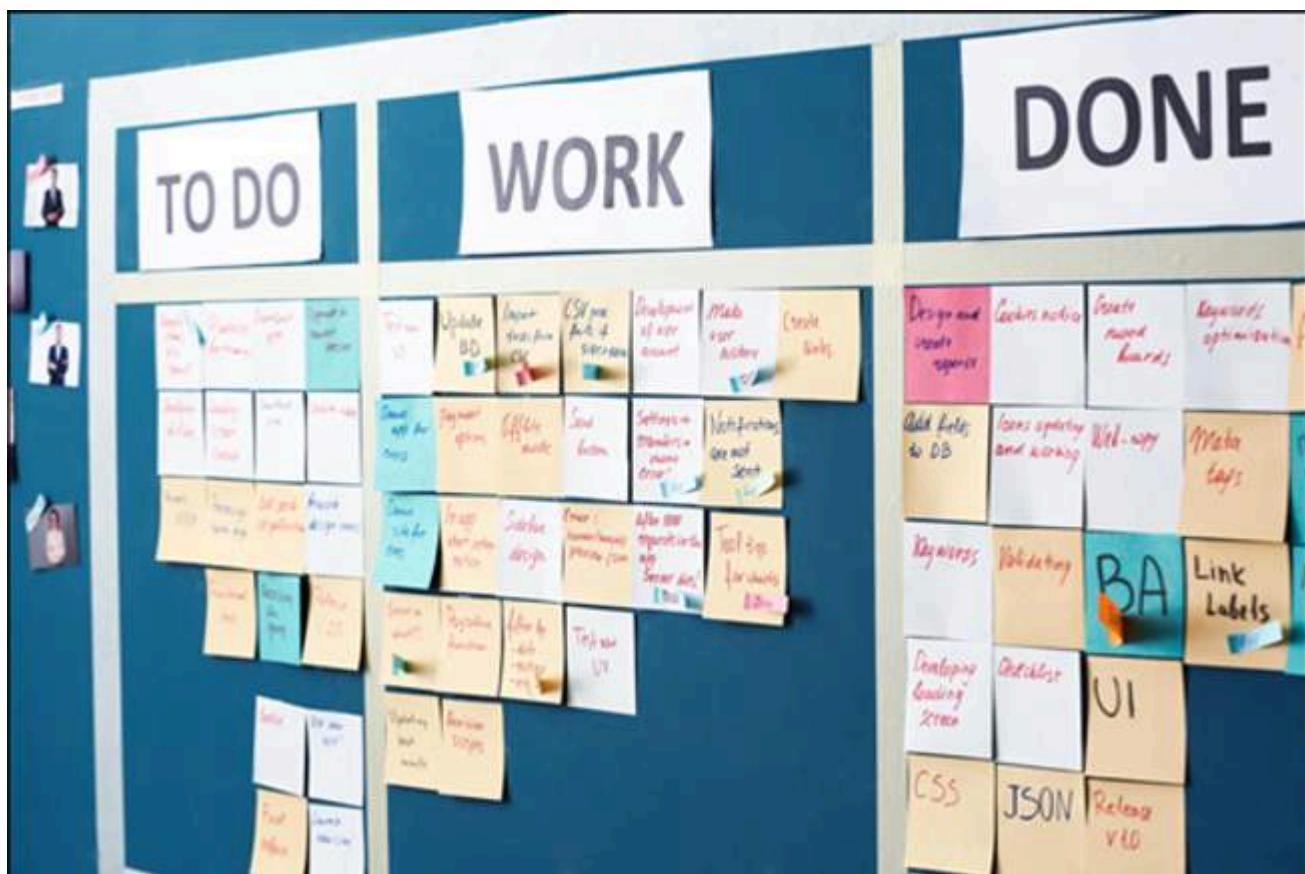


Figura 2 | Representação do quadro Scrum. Fonte: Canva.

Cada linha do quadro simboliza uma história do projeto, com as colunas divididas em três categorias: "a fazer" (*to do*), "em execução" (*doing/work*) e "concluída" (*done*). Cada empresa pode adicionar mais colunas, caso seja necessário. As tarefas relacionadas a cada história são

transferidas do *backlog* do produto para o quadro, marcando o progresso durante uma Sprint específica.

Para facilitar a visualização e o acompanhamento das tarefas, *post-its* coloridos são usados. Cada cor pode representar um tipo diferente de tarefa ou status. Por exemplo, um *post-it* vermelho pode indicar uma tarefa que envolve a correção de defeitos no código. Além disso, as cores podem ser usadas para identificar os membros da equipe responsáveis por determinadas tarefas. Essa organização visual ajuda a equipe a compreender rapidamente o estado das atividades e a gerenciar eficientemente o fluxo de trabalho do projeto. Entretanto, uma forma mais completa e racional de se usar o *post-it* está apresentado na Figura 3:

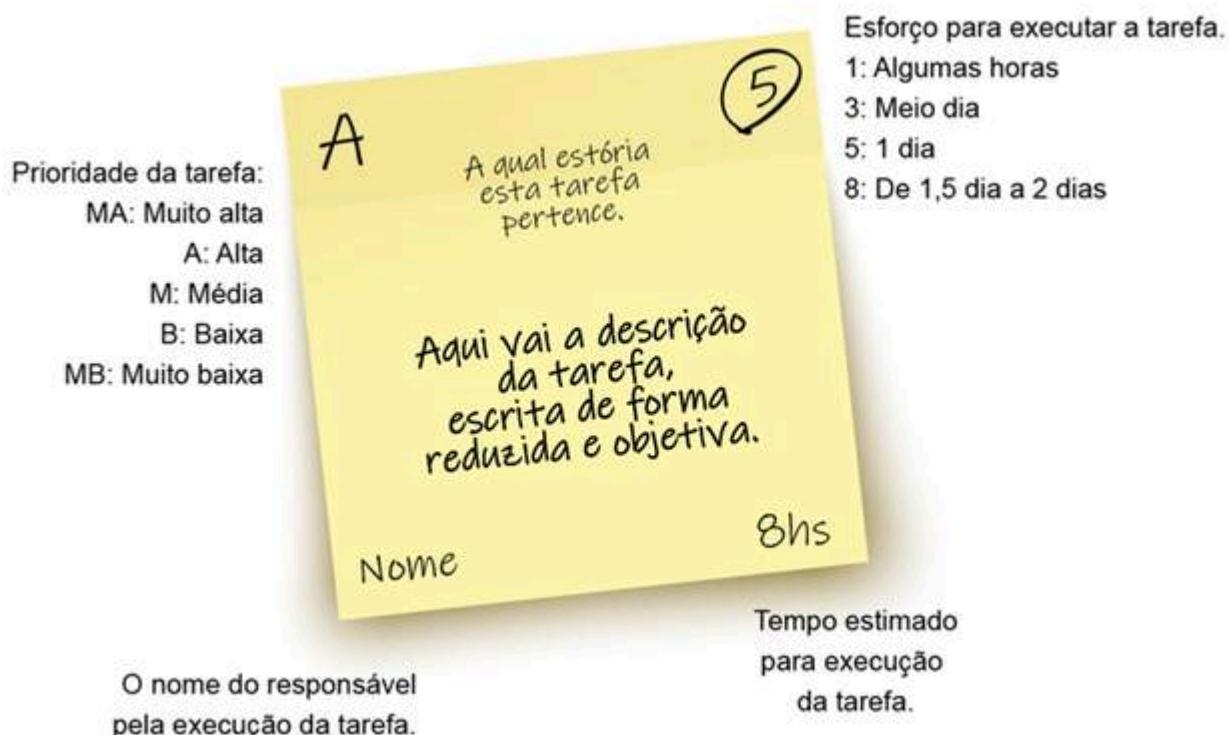


Figura 3 | Exemplo de controle de atividades por meio de post-it. Fonte: Maitino (2021).

O quadro Scrum, bem como a metodologia em si, são flexíveis e podem ser adaptados à cultura e às necessidades específicas de cada equipe. Contudo, o propósito fundamental do quadro permanece: ele serve como uma ferramenta de comunicação eficaz e direta, mostrando de maneira clara e comprehensível o que cada membro da equipe está realizando e em qual fase da atividade se encontra. Essa abordagem simplificada facilita o entendimento compartilhado das tarefas em andamento e promove uma melhor coordenação do trabalho do grupo.

**Siga em Frente...**

**XP (*Extreme Programming*)**

Extreme Programming (XP), é uma metodologia que incorpora um conjunto de regras e práticas rigorosas dentro de quatro áreas metodológicas principais: planejamento, projeto, codificação e testes. Este processo, ilustrado na Figura 4, destaca conceitos e tarefas essenciais associadas a cada uma dessas áreas. As principais atividades da XP, que formam o núcleo desta abordagem, são detalhadas a seguir.

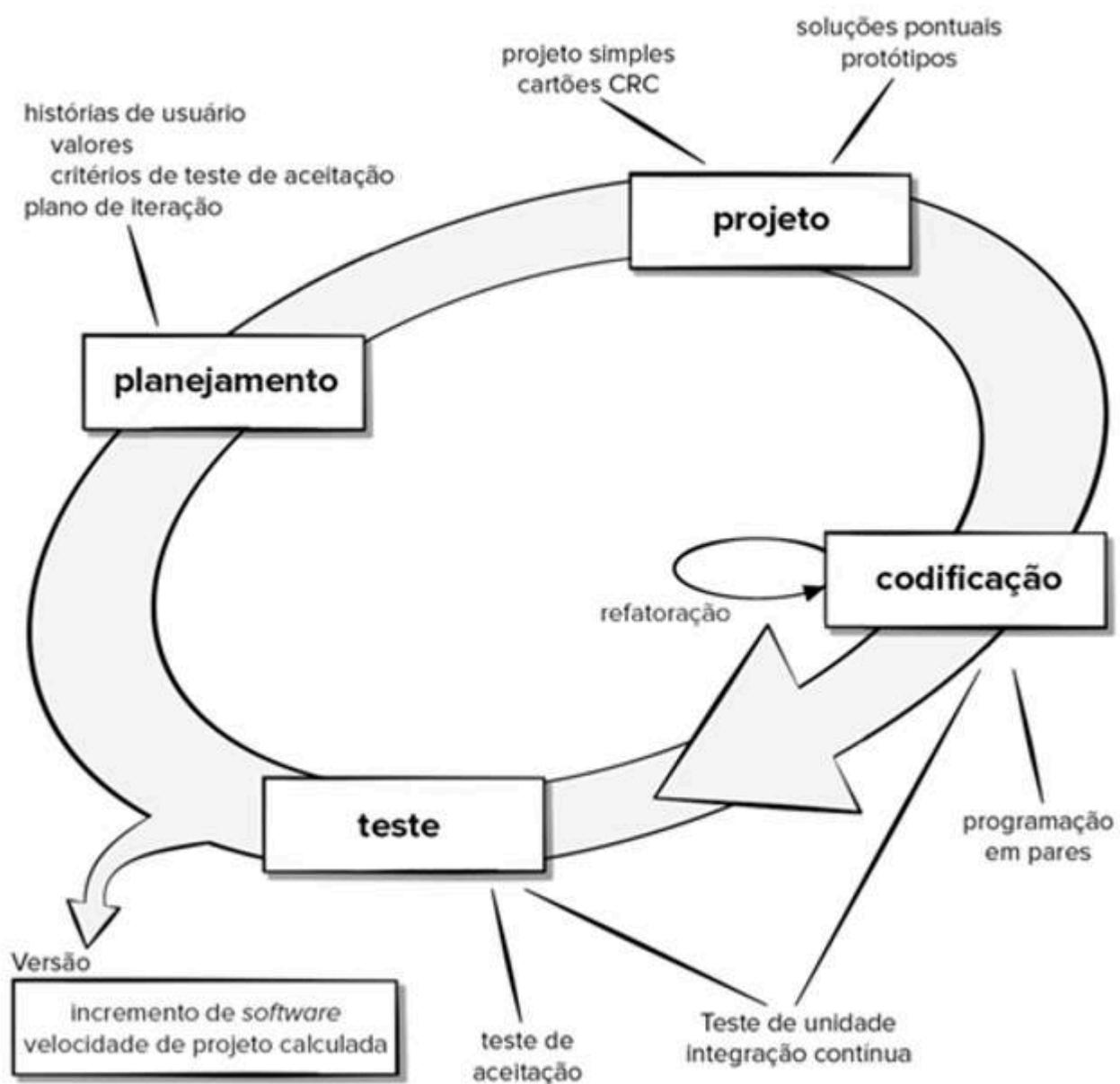


Figura 4 | O processo da Extreme Programming (XP). Fonte: Pressman (2021).

A fase de planejamento na Programação Extrema (XP), também conhecida como "o jogo do planejamento", começa com a prática de ouvir atentamente. Esta etapa leva à criação de "histórias" ou histórias de usuário, que detalham os resultados esperados, características e funcionalidades do software a ser desenvolvido. Cada história de usuário, que é elaborada pelo

cliente e registrada em uma ficha, recebe uma prioridade baseada no seu valor de negócio. A equipe XP, por sua vez, avalia cada história e estima um custo para sua implementação, expresso em semanas de desenvolvimento. Vale ressaltar que novas histórias podem ser adicionadas a qualquer momento.

Clientes e desenvolvedores colaboram para determinar quais histórias serão incluídas no próximo incremento de software. Uma vez alcançado um acordo básico sobre quais histórias serão desenvolvidas, a data de entrega e outros aspectos do projeto, a equipe XP organiza as histórias de acordo com uma destas três abordagens: implementar todas as histórias imediatamente, priorizar e implementar primeiro as histórias de maior valor, ou priorizar e implementar primeiro as histórias de maior risco.

Após a entrega do primeiro incremento do projeto, a equipe XP calcula a "velocidade do projeto", que corresponde ao número de histórias de clientes implementadas nesse incremento. Esta métrica serve para auxiliar na estimativa de datas de entrega e no planejamento de versões futuras do software, permitindo que a equipe ajuste seus planos de acordo com a velocidade observada.

O projeto adere ao princípio *Keep It Simple, Stupid* (KISS), que defende a simplicidade na concepção do projeto. A ideia é evitar o desenvolvimento de funcionalidades extras baseadas apenas na suposição de que elas podem ser necessárias no futuro.

A XP promove o uso de cartões Classe-Responsabilidade-Colaborador (CRC) como uma ferramenta eficiente para estruturar o software em um contexto orientado a objetos. Estes cartões ajudam a identificar e organizar as classes orientadas a objetos que são relevantes para o incremento de software em desenvolvimento. Os cartões CRC são os únicos artefatos de projeto gerados durante o processo XP (Sbrocco, 2012).

Quando surgem desafios complexos de projeto, a XP sugere a criação imediata de um protótipo operacional para essa parte específica do projeto. Um princípio central da XP é que o desenvolvimento do projeto ocorre tanto antes quanto após o início da codificação. A refatoração, que envolve a modificação ou otimização do código sem alterar seu comportamento externo, é uma prática contínua durante o desenvolvimento do sistema (Wazlawick, 2013). De fato, o próprio processo de desenvolvimento serve como guia para a melhoria contínua do projeto na XP.

Na etapa de codificação da XP, a equipe inicia criando testes unitários para cada uma das histórias que serão incluídas na versão atual do software. Estes testes servem para orientar os desenvolvedores sobre o que precisa ser implementado para passar nos testes. Uma vez que o código é desenvolvido, ele pode ser testado imediatamente, proporcionando feedback instantâneo para os desenvolvedores.

Um aspecto fundamental e frequentemente debatido na XP é a prática da programação em pares. A XP sugere que dois programadores trabalhem juntos em um único computador para escrever o código de uma história. Essa abordagem oferece um meio eficaz de solução de problemas em tempo real, já que duas mentes tendem a trabalhar melhor do que uma. Além

disso, oferece um mecanismo de garantia de qualidade em tempo real, uma vez que o código é revisado enquanto é escrito.

À medida que as duplas de programadores concluem suas partes do trabalho, o código é integrado ao trabalho dos demais membros da equipe. Esta estratégia de "integração contínua" é crucial para evitar problemas de compatibilidade e de interface no início do processo de desenvolvimento (Pressman, 2021).

Os testes unitários são elaborados de maneira que possam ser automatizados, permitindo que sejam executados de forma fácil e repetida. Esta abordagem favorece a realização de testes de regressão sempre que o código é alterado, uma prática frequente devido à filosofia de refatoração incorporada pela XP.

Além disso, a XP enfatiza a importância dos testes de aceitação, também conhecidos como testes de cliente. Estes são especificados pelo cliente e focam nas características e funcionalidades do sistema completo que são visíveis e avaliáveis por ele. Os testes de aceitação são derivados das histórias de usuário implementadas em cada versão do software, assegurando que o produto atenda às expectativas e aos requisitos do cliente.

## Vamos Exercitar?

Para esta atividade, é importante que você elabore um quadro Scrum para organizar as suas tarefas diárias ou de estudos. Com isso, este quadro deve conter colunas que representem o progresso da atividade. Um exemplo deste quadro está apresentado na Figura 5.

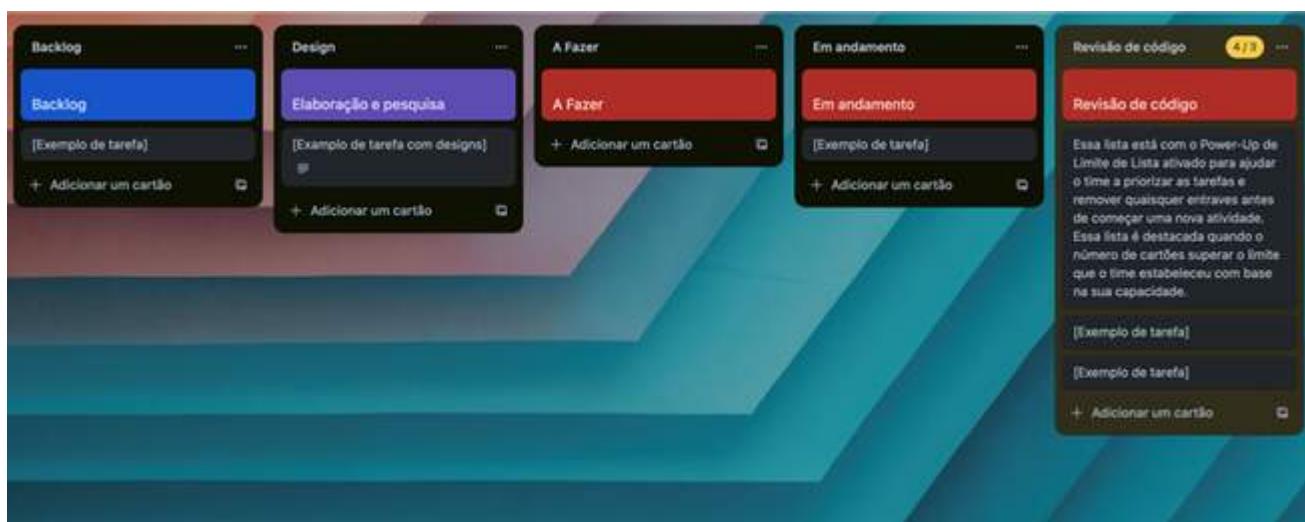


Figura 5 | Quadro Scrum no Trello. Fonte: captura de tela do Trello elaborada pela autora.

Importante ressaltar que cada pessoa terá um quadro diferente, pois ele quem determina as atividades e as colunas que o seu quadro terá. Esse quadro deve se adequar a sua realidade e ajudar você no seu planejamento.

## Saiba mais

No Capítulo 3 do livro [Engenharia de software \(Pressman\)](#) você pode se aprofundar mais no assunto sobre as metodologias ágeis.

O livro [Metodologias Ágeis - Engenharia de Software sob Medida](#) apresenta mais sobre as metodologias ágeis, além de apresentar outros métodos

O [Trello](#) é uma poderosa ferramenta de gerenciamento de projetos, simplificando a colaboração com seus quadros intuitivos. Facilita o acompanhamento de tarefas, organização de ideias e promove eficiência na equipe. Ideal para otimizar a produtividade de forma visual e colaborativa. Pode ser utilizado para gerenciar projetos ágeis.

## Referências

MAITINO NETO, R. **Engenharia de software**. Londrina: Editora e Distribuidora Educacional S.A., 2021.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SBROCCO, J. H. T. de C. **Metodologias ágeis**: engenharia de software sob medida. São Paulo: Érica, 2012.

WAZLAWICK, R. S. **Engenharia de software**: conceitos e práticas. Rio de Janeiro: Elsevier, 2013.

## Aula 3

Requisitos

### Requisitos

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá conceitos para sua prática profissional. Abordaremos os fundamentos dos requisitos funcionais, não funcionais e a essencial engenharia de requisitos. Compreender esses elementos é vital para o sucesso na elaboração de sistemas eficientes. Prepare-se para aprimorar suas habilidades e impulsionar sua carreira!

## Ponto de Partida

Nesta aula, exploramos três conceitos fundamentais no desenvolvimento de software: requisitos funcionais, requisitos não funcionais e engenharia de requisitos. Os requisitos funcionais descrevem as funções específicas que um sistema deve executar, como processamento de dados, execução de cálculos, e interações com o usuário. Estes requisitos são vitais para delinear o que o sistema é esperado fazer, fornecendo uma base clara para o design e a implementação.

Por outro lado, os requisitos não funcionais focam em aspectos como desempenho, segurança, usabilidade e confiabilidade. Estes requisitos são essenciais para assegurar que o sistema não apenas execute suas funções, mas também o faça de maneira eficiente, segura e acessível. Eles têm um impacto significativo na qualidade geral do sistema, na satisfação do usuário e na viabilidade operacional do projeto.

A engenharia de requisitos, que engloba a identificação, documentação, e manutenção dos requisitos, é um processo crítico que permeia todo o ciclo de vida do desenvolvimento de software. Este processo garante que os requisitos sejam bem definidos, compreensíveis, e alinhados com as necessidades e restrições do projeto. Uma boa prática de engenharia de requisitos ajuda a prevenir problemas comuns no desenvolvimento de software, como mal-entendidos sobre o que o sistema deve fazer, requisitos incompletos ou incompatíveis, e mudanças tardias que podem ser custosas.

A compreensão desses conceitos é crucial para qualquer profissional de TI. Requisitos bem definidos e bem gerenciados são a chave para o sucesso de qualquer projeto de software, pois orientam todas as fases subsequentes do processo de desenvolvimento e ajudam a garantir que o produto final atenda às expectativas dos usuários e às necessidades do negócio.

Considerando esses conceitos de requisitos, analise a seguinte lista para um sistema de gerenciamento de biblioteca. Para cada requisito, classifique-o como "Funcional" ou "Não Funcional", justificando brevemente a sua escolha.

- O sistema deve permitir que os usuários pesquisem livros por título, autor ou ISBN.
- O sistema deve garantir a privacidade dos dados dos usuários, em conformidade com as normas de proteção de dados.

- O tempo de resposta para uma pesquisa de livro no sistema não deve exceder dois segundos.
- O sistema deve permitir que os bibliotecários adicionem, removam ou atualizem informações de livros no banco de dados.
- O sistema deve ser acessível via navegadores web mais populares, como Chrome, Firefox e Safari.
- O sistema deve enviar uma notificação por e-mail aos usuários quando um livro reservado estiver disponível.
- O sistema deve manter um registro de todas as transações de empréstimo e devolução de livros.
- O sistema deve ser capaz de gerar relatórios mensais de atividade da biblioteca, incluindo empréstimos, devoluções e reservas.
- O sistema deve ser capaz de suportar até 1000 usuários simultâneos.
- O design da interface do sistema deve ser intuitivo e amigável para todos os tipos de usuários.

Vamos conhecer mais sobre esse assunto?!

Bons estudos!

## Vamos Começar!

Os requisitos de um sistema são as especificações detalhadas das funcionalidades que o sistema deve prover, juntamente com as limitações associadas ao seu funcionamento. Estes requisitos refletem as demandas dos clientes para um sistema que cumpre um propósito específico, como operar um dispositivo, processar um pedido ou acessar informações. A engenharia de requisitos é o processo de identificar, analisar, documentar e validar essas funcionalidades e limitações.

A indústria de software não emprega o termo "requisito" de maneira uniforme. Em algumas situações, pode representar apenas uma descrição genérica e de alto nível de uma funcionalidade desejada ou de uma restrição do sistema. Em contrapartida, em outros contextos, pode ser uma descrição formal e minuciosa de uma função específica do sistema.

Durante o processo de engenharia de requisitos, um desafio comum é a dificuldade em distinguir claramente entre diferentes níveis de descrição de requisitos. Para abordar esta questão, pode-se utilizar a terminologia "requisitos de usuário" para descrever requisitos abstratos e de alto nível, e "requisitos de sistema" para detalhar especificamente o que o sistema deve fazer. Estes podem ser definidos da seguinte forma (Sommerville, 2018):

- **Requisitos de Usuário:** são expressos em linguagem natural e complementados por diagramas, detalhando os serviços que o sistema deve oferecer aos usuários e as restrições sob as quais deve operar.

- **Requisitos de Sistema:** representam uma descrição mais aprofundada das funções, serviços e limitações operacionais do sistema de software. O documento de requisitos do sistema, também conhecido como especificação funcional, precisa especificar de forma precisa o que será implementado. Este documento pode ser uma parte do contrato entre o comprador do sistema e os desenvolvedores de software.

Esses diferentes níveis de requisitos são valiosos porque fornecem informações sobre o sistema para variados tipos de leitores e stakeholders.

Os requisitos de software são comumente classificados em requisitos funcionais e não funcionais:

- **Requisitos Funcionais:** estes são descrições dos serviços que o sistema deve fornecer, como ele deve responder a entradas específicas, e seu comportamento em certas situações. Em alguns casos, também podem incluir o que o sistema não deve fazer.
- **Requisitos Não Funcionais:** estes são restrições aos serviços ou funcionalidades fornecidas pelo sistema. Eles abrangem limitações de tempo, restrições no processo de desenvolvimento e exigências decorrentes de normas. Diferentemente das características ou dos serviços individuais do sistema, os requisitos não funcionais geralmente se aplicam ao sistema como um todo.

Contudo, a separação entre esses tipos de requisitos nem sempre é tão clara quanto essas definições básicas sugerem. Por exemplo, um requisito de usuário relativo à segurança, como uma limitação no acesso apenas a usuários autorizados, pode parecer um requisito não funcional. No entanto, ao ser mais detalhado, esse requisito pode levar a outros claramente funcionais, como a necessidade de implementar recursos de autenticação de usuário no sistema.

Isso indica que os requisitos não são isolados e frequentemente influenciam ou limitam outros requisitos. Assim, os requisitos de sistema não apenas especificam os serviços ou as características que o sistema precisa oferecer, mas também as funcionalidades necessárias para garantir a entrega correta desses serviços ou dessas características. Vamos entender mais sobre essas classificações de requisitos.

## Requisitos Funcionais

Os requisitos funcionais de um sistema especificam as ações que ele deve executar. Esses requisitos são influenciados pelo tipo de software a ser criado, pelo perfil dos usuários e pela metodologia adotada pela organização na elaboração desses requisitos. Quando formulados como requisitos de usuário, eles são geralmente apresentados de maneira abstrata para que sejam facilmente entendidos pelos usuários do sistema (Sommerville, 2018). Por outro lado, os requisitos funcionais do sistema, em sua forma mais específica, detalham minuciosamente as funcionalidades do sistema, incluindo suas entradas, saídas e exceções.

A amplitude dos requisitos funcionais do sistema varia desde exigências gerais, que delineiam as operações básicas do sistema, até requisitos altamente específicos, que refletem as particularidades dos sistemas e processos de trabalho dentro de uma organização. A seguir, está apresentado alguns exemplos de requisitos funcionais para um sistema hipotético:

- O sistema deve permitir que os usuários se autentiquem utilizando um nome de usuário e uma senha. Após a autenticação bem-sucedida, os usuários devem ter acesso às funcionalidades apropriadas com base em seu nível de permissão.
- O sistema deve permitir que os usuários criem, modifiquem, visualizem e excluam pedidos. Cada pedido deve incluir informações como data do pedido, detalhes dos itens pedidos, informações do cliente e status do pedido.
- O sistema deve fornecer a capacidade de gerar relatórios de vendas mensais que incluam total de vendas, número de pedidos e detalhes de itens mais vendidos. Os relatórios devem ser exportáveis em formatos como PDF e Excel.
- O sistema deve enviar automaticamente notificações por e-mail aos clientes quando seus pedidos forem processados, enviados ou entregues. O conteúdo do e-mail deve incluir detalhes do pedido e uma estimativa de tempo de entrega.

Estes requisitos funcionais descrevem tarefas e comportamentos específicos que o sistema deve ser capaz de realizar, refletindo funcionalidades concretas que um usuário espera do sistema.

A falta de clareza na definição de requisitos frequentemente leva a problemas na engenharia de software. É comum que desenvolvedores interpretem requisitos ambíguos de maneira que facilite a implementação, mas essa interpretação pode não estar alinhada com as expectativas do cliente. Como resultado, é necessário revisar e estabelecer novos requisitos, o que implica em modificações no sistema. Esse processo, por sua vez, costuma resultar em atrasos na entrega e no aumento dos custos do projeto.

Idealmente, a especificação de requisitos funcionais de um sistema deve ser completa e consistente. Completude implica que todos os serviços desejados pelos usuários sejam claramente definidos. Consistência requer que não haja contradições nas definições dos requisitos. No entanto, na prática, alcançar completude e consistência é extremamente desafiador, especialmente em sistemas grandes e complexos. Isso se deve, em parte, à facilidade de ocorrerem erros e omissões durante a elaboração de especificações para sistemas intrincados. Além disso, sistemas de grande escala geralmente envolvem múltiplos stakeholders, que são indivíduos ou entidades impactadas pelo sistema de alguma forma. Esses stakeholders podem ter necessidades e expectativas divergentes, frequentemente inconsistentes entre si. Tais inconsistências podem não ser aparentes inicialmente, durante a fase de especificação dos requisitos, e acabam sendo incorporadas na especificação (Pressman, 2021). Problemas decorrentes dessas inconsistências podem emergir em análises mais detalhadas ou após a entrega do sistema ao cliente.

**Siga em Frente...**

## Requisitos não funcionais

Requisitos não funcionais, como o termo indica, são aqueles que não se relacionam diretamente com as funcionalidades específicas que o sistema oferece aos seus usuários. Eles podem referir-se a características emergentes do sistema, tais como confiabilidade, tempo de resposta e eficiência no uso de recursos. Alternativamente, esses requisitos podem estabelecer restrições sobre a implementação do sistema, como as capacidades dos dispositivos de entrada/saída ou os formatos de dados utilizados nas interfaces com outros sistemas.

Os requisitos não funcionais, como desempenho, segurança ou disponibilidade, geralmente especificam ou limitam as propriedades do sistema como um todo. Frequentemente, eles são mais críticos do que requisitos funcionais individuais. Embora os usuários possam, muitas vezes, contornar uma funcionalidade que não atenda completamente às suas necessidades, falhar em cumprir um requisito não funcional pode comprometer todo o sistema. Por exemplo, um sistema de aviação que não atenda aos requisitos de confiabilidade não será certificado como seguro para uso; ou um sistema de controle embutido que não cumpre os requisitos de desempenho pode falhar em operar suas funções de controle adequadamente.

Embora seja comum identificar os componentes do sistema responsáveis por implementar requisitos funcionais específicos (por exemplo, componentes dedicados à formatação para atender aos requisitos de impressão de relatórios), relacionar os componentes aos requisitos não funcionais pode ser mais desafiador. A implementação desses requisitos muitas vezes se dispersa por todo o sistema. Isso ocorre por dois motivos principais:

- Requisitos não funcionais frequentemente impactam a arquitetura global do sistema, em vez de apenas componentes isolados. Por exemplo, para atender aos requisitos de desempenho, pode ser necessário estruturar o sistema de forma a reduzir a comunicação entre componentes.
- Um único requisito não funcional, como um requisito de segurança, pode originar vários requisitos funcionais relacionados que definam os serviços necessários no novo sistema. Além disso, esses requisitos não funcionais podem também impor restrições sobre os requisitos existentes.

Um desafio comum relacionado aos requisitos não funcionais é que eles são frequentemente estabelecidos pelos usuários ou clientes como objetivos gerais, refletindo desejos como facilidade de uso, capacidade do sistema de se recuperar de falhas ou a rapidez nas respostas às ações do usuário. Embora essas metas representem boas intenções, elas podem ser problemáticas para os desenvolvedores do sistema devido à sua natureza aberta à interpretação, o que pode levar a desentendimentos na entrega do sistema. Por exemplo, a seguinte meta de usabilidade, típica da forma como um gerente poderia expressar os requisitos, ilustra essa questão:

O sistema deve ser facilmente utilizável pelo pessoal médico e organizado de modo a minimizar erros dos usuários.

Para demonstrar como a meta mencionada anteriormente pode ser reformulada em um requisito não funcional testável, consideremos a seguinte descrição. Embora não seja possível verificar objetivamente a facilidade de uso global do sistema, é viável incluir mecanismos no software para monitorar os erros cometidos pelos usuários durante os testes.

Após receber quatro horas de treinamento, o pessoal médico deve ser capaz de operar todas as funções do sistema. Além disso, a média de erros cometidos por usuários experientes não deve ultrapassar dois por hora de uso do sistema.

Idealmente, os requisitos não funcionais devem ser expressos de forma quantitativa, permitindo testes objetivos. Por exemplo, as métricas listadas na Tabela 1 podem ser utilizadas para especificar as propriedades não funcionais do sistema. Estas características podem ser medidas durante os testes do sistema para verificar se os requisitos não funcionais foram atendidos.

Propriedade	Medida
Velocidade	Transações processadas/segundo Tempo de resposta do usuário/evento Tempo de atualização de tela
Tamanho	Megabytes Número de chips de memória ROM
Facilidade de uso	Tempo de treinamento Número de ícones de ajuda
Confiabilidade	Tempo médio para falha Probabilidade de indisponibilidade Taxa de ocorrência de falhas Disponibilidade
Robustez	Tempo de reinício após falha Percentual de eventos que causam falhas Probabilidade de corrupção de dados em caso de falha
Portabilidade	Percentual de declarações dependentes do sistema-alvo Número de sistemas-alvo

Tabela 1 | Métricas para especificar requisitos não funcionais. Fonte: Sommerville (2018).

Na realidade, os clientes frequentemente enfrentam dificuldades em converter suas metas em requisitos mensuráveis. Algumas metas, como a manutenibilidade, não possuem métricas claramente definidas. Em outros casos, mesmo quando é possível uma especificação quantitativa, os clientes podem não conseguir vincular suas necessidades a essas especificações numéricas. Eles podem não compreender, por exemplo, o que um número especificando a confiabilidade necessária realmente significa em termos práticos de uso diário dos sistemas computacionais. Além disso, o custo para verificar objetivamente requisitos não funcionais mensuráveis pode ser proibitivo, e os clientes, que financiam o sistema, podem não considerar esses custos justificáveis.

Requisitos não funcionais muitas vezes entram em conflito ou interagem com outros requisitos, sejam eles funcionais ou não funcionais (Pressman, 2021). Por exemplo, um requisito de autenticação pode demandar a instalação de leitores de cartão em cada computador do sistema. Contudo, se existir um requisito para acesso móvel ao sistema, como nos laptops de médicos ou enfermeiros que geralmente não possuem leitores de cartão, será necessário desenvolver um método alternativo de autenticação.

Na documentação de requisitos, separar requisitos funcionais dos não funcionais pode ser desafiador. Se apresentados separadamente, a relação entre eles pode se tornar difícil de compreender. No entanto, os requisitos diretamente relacionados às propriedades emergentes do sistema, como desempenho ou confiabilidade, devem ser claramente enfatizados. Isso pode ser feito colocando-os em uma seção separada do documento de requisitos ou destacando-os de alguma forma dos outros requisitos do sistema.

## Engenharia de Requisitos

Os processos de engenharia de requisitos geralmente abrangem quatro atividades principais, cada uma com um objetivo específico. Estas atividades incluem avaliar a viabilidade do sistema para a empresa (estudo de viabilidade), identificar os requisitos (elicitação e análise), formalizá-los em um formato padronizado (especificação), e assegurar que os requisitos definam corretamente o sistema conforme desejado pelo cliente (validação) (Reinehr, 2020). Contudo, na prática, a engenharia de requisitos é um processo iterativo, em que essas atividades não ocorrem em sequência linear, mas sim de maneira entrelaçada.

A Figura 1 ilustra a natureza intercalada das atividades no processo de engenharia de requisitos, organizadas em torno de uma espiral representativa de um processo iterativo. O resultado desse processo é um documento de requisitos do sistema. A quantidade de tempo e esforço dedicada a cada atividade em cada iteração varia de acordo com o estágio do processo e o tipo de sistema em desenvolvimento. No início do processo, há um enfoque maior na compreensão dos requisitos de negócio e não funcionais em um nível mais abstrato, assim como dos requisitos de usuário. Conforme o processo avança para os anéis mais externos da espiral, aumenta o foco na elicitação e detalhamento dos requisitos do sistema.

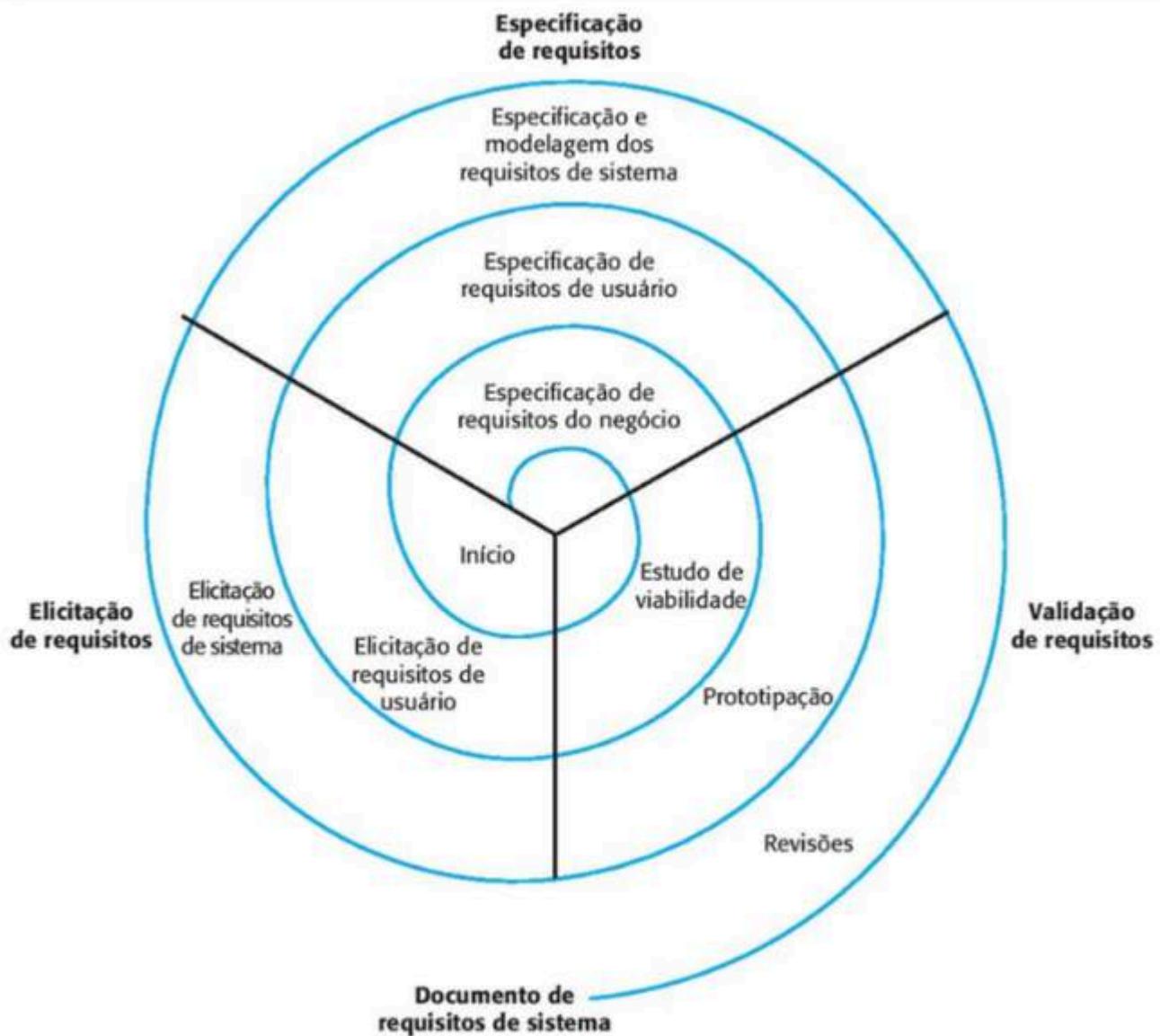


Figura 1 | Visão em espiral do processo de engenharia de requisitos. Fonte: Sommerville (2018).

Este modelo em espiral permite abordagens em que os requisitos são desenvolvidos com diferentes níveis de detalhe. O número de iterações ao redor da espiral pode variar, podendo a espiral ser concluída após a definição de alguns ou todos os requisitos de usuário. Em vez de prototipagem, métodos de desenvolvimento ágil podem ser adotados, permitindo que os requisitos e a implementação do sistema evoluam conjuntamente.

Após um estudo inicial de viabilidade, a próxima etapa no processo de engenharia de requisitos envolve a elicitação e análise de requisitos (Sommerville, 2018). Durante esta fase, os engenheiros de software colaboram com os clientes e usuários finais para coletar informações sobre o domínio da aplicação, os serviços que o sistema deve prover, seu desempenho esperado, as limitações de hardware e outros aspectos relevantes.

O processo de elicitação e análise de requisitos pode envolver diferentes indivíduos dentro de uma organização. Um stakeholder do sistema é alguém que exerce influência direta ou indireta sobre os requisitos do sistema. Isso inclui não apenas os usuários finais que interagirão diretamente com o sistema, mas também qualquer outra pessoa na organização que possa ser impactada por ele (Reinehr, 2020). Além disso, outros stakeholders importantes podem ser engenheiros envolvidos no desenvolvimento ou manutenção de sistemas correlatos, gerentes de negócios, especialistas no domínio de aplicação e representantes de entidades sindicais.

O modelo do processo de elicitação e análise, ilustrado na Figura 2, representa um esquema geral que cada organização adapta conforme suas necessidades específicas. A forma como este modelo é implementado varia de acordo com diversos fatores locais, tais como o nível de especialização dos funcionários, o tipo de sistema que está sendo desenvolvido, as normas e os padrões aplicáveis, entre outros aspectos. Essa adaptação permite que as organizações personalizem o processo para melhor atender às suas circunstâncias e aos seus objetivos.



Figura 2 | O processo de elicitação e análise de requisitos. Fonte: Sommerville (2018).

No processo de engenharia de requisitos, inicialmente ocorre a descoberta de requisitos por meio da interação com stakeholders para identificar suas necessidades e seus requisitos de domínio. Diversas técnicas podem ser empregadas nesta etapa. Em seguida, os requisitos identificados são classificados e organizados, frequentemente utilizando um modelo de arquitetura do sistema para agrupá-los por subsistemas. A etapa subsequente envolve a priorização e negociação de requisitos, resolvendo conflitos entre diferentes stakeholders, que

muitas vezes têm necessidades divergentes. Por fim, os requisitos são documentados, formal ou informalmente, e preparados para a próxima fase do ciclo de desenvolvimento. Este processo é iterativo, com cada etapa contribuindo para a definição clara e organizada dos requisitos do sistema.

O processo de especificação de requisitos envolve documentar os requisitos de usuário e de sistema em um documento formal. Idealmente, estes requisitos devem ser claros, inequívocos, fáceis de compreender, completos e consistentes, embora na prática seja desafiador atingir todos esses critérios devido a diferentes interpretações e conflitos inerentes entre os stakeholders. Os requisitos de usuário são geralmente redigidos em linguagem natural e podem ser acompanhados por diagramas e tabelas para maior clareza. Quanto aos requisitos de sistema, eles também podem ser descritos em linguagem natural, mas é comum o uso de outras notações, como formulários baseados, gráficos ou modelos matemáticos, para representá-los de forma mais precisa. A Tabela 2 apresenta um resumo das várias notações possíveis para a redação de requisitos de sistema.

Notação	Descrição
Sentenças em linguagem natural	Os requisitos são escritos usando frases numeradas em linguagem natural. Cada frase deve expressar um requisito.
Linguagem natural estruturada	Os requisitos são escritos em linguagem natural em um formulário ou template. Cada campo fornece informações sobre um aspecto do requisito.
Notações gráficas	Modelos gráficos, complementados por anotações em texto, são utilizados para definir os requisitos funcionais do sistema. São utilizados com frequência os diagramas de casos de uso e de sequência da UML.
Especificações matemáticas	Essas notações se baseiam em conceitos matemáticos como as máquinas de estados finitos ou conjuntos. Embora essas especificações inequívocas possam reduzir a ambiguidade em um documento de requisitos, a maioria dos clientes não comprehende uma especificação formal. Eles conseguem averiguar

se ela representa o que desejam e relutam em aceitar essa especificação como um contrato do sistema.

Tabela 2 | Notação para escrever requisitos de sistema. Fonte: Sommerville (2018).

Os requisitos de usuário de um sistema devem claramente detalhar os requisitos funcionais e não funcionais de uma forma acessível para usuários que não possuem conhecimento técnico aprofundado. Idealmente, esses requisitos deveriam focar somente no comportamento externo do sistema, sem incluir aspectos de sua arquitetura ou seu design. Assim, ao redigir requisitos de usuário, deve-se evitar o uso de jargões técnicos, notações complexas ou formalismos, preferindo-se a linguagem natural complementada por tabelas, formulários e diagramas simples e compreensíveis.

Por outro lado, os requisitos de sistema são elaborações dos requisitos de usuário, usados pelos engenheiros de software como base para o desenvolvimento do sistema. Eles adicionam detalhes e descrevem como o sistema satisfará os requisitos de usuário. Esses requisitos são fundamentais para a implementação do sistema e, portanto, devem ser uma especificação completa e detalhada de todo o sistema, podendo inclusive fazer parte do contrato de desenvolvimento.

A validação de requisitos é crucial no processo de desenvolvimento de sistemas, consistindo em verificar se os requisitos realmente correspondem ao que o cliente deseja. Esta etapa é fundamental para identificar problemas, e ela se intercala com a elicitação e análise de requisitos. Erros nos requisitos podem resultar em custos significativos de correção, especialmente se descobertos durante o desenvolvimento ou após a implantação do sistema. Geralmente, corrigir um problema nos requisitos é mais dispendioso do que consertar falhas de projeto ou programação, pois alterações nos requisitos frequentemente exigem modificações no projeto e na implementação do sistema.

Durante a validação, é essencial realizar várias conferências nos requisitos, incluindo (Sommerville, 2018):

- **Conferência de Validade:** verifica se os requisitos atendem às necessidades reais dos usuários, considerando possíveis mudanças nas circunstâncias desde a elicitação inicial.
- **Conferência de Consistência:** assegura que não existam conflitos ou contradições entre os requisitos documentados.
- **Conferência de Completude:** confirma que todos os requisitos necessários, abrangendo funções e restrições, estão incluídos no documento.
- **Conferência de Realismo:** avalia se os requisitos são tecnicamente viáveis dentro do orçamento e do cronograma estipulados para o projeto.
- **Verificabilidade:** para diminuir o potencial de conflito entre o cliente e o desenvolvedor, é crucial que os requisitos do sistema sejam formulados de forma verificável. Isso implica na

capacidade de elaborar testes específicos que comprovem que o sistema finalizado atende a cada um dos requisitos definidos.

Existem várias técnicas de validação de requisitos que podem ser empregadas individualmente ou em combinação:

- **Revisões de Requisitos:** neste método, uma equipe de revisores analisa os requisitos de forma sistemática para identificar erros e inconsistências.
- **Prototipação:** consiste no desenvolvimento de um modelo funcional do sistema, que é utilizado com usuários finais e clientes para verificar se atende às suas necessidades e expectativas. Essa interação permite que os stakeholders avaliem o sistema e sugiram modificações nos requisitos.
- **Geração de Casos de Teste:** os requisitos devem ser testáveis. Criar testes durante a validação pode expor falhas nos requisitos. Se for difícil ou impossível elaborar um teste, isso geralmente indica que os requisitos serão complexos de implementar e devem ser revistos. Desenvolver testes com base nos requisitos de usuário antes da programação é uma prática do desenvolvimento orientado a testes.

Validar requisitos é um processo desafiador, e é difícil garantir que eles atendam completamente às necessidades dos usuários, que precisam visualizar como o sistema funcionará em seu ambiente de trabalho, o que pode ser complicado mesmo para profissionais de TI, e mais ainda para usuários leigos. Portanto, é comum que não sejam identificados todos os problemas nos requisitos durante a validação. Mudanças adicionais nos requisitos são frequentemente necessárias para corrigir omissões e mal-entendidos, mesmo após se chegar a um consenso sobre o documento de requisitos.

## Vamos Exercitar?

Este gabarito fornece a classificação de cada requisito, com base na distinção entre funções específicas do sistema (funcionais) e propriedades ou restrições gerais do sistema (não funcionais). A justificativa de cada classificação é importante para entender o raciocínio por trás da diferenciação entre os dois tipos de requisitos.

- Funcional: especifica uma função do sistema – permitir que usuários pesquisem livros.
- Não Funcional: relaciona-se com a segurança e privacidade dos dados dos usuários, uma propriedade do sistema.
- Não Funcional: define um requisito de desempenho (tempo de resposta) do sistema.
- Funcional: descreve a função de gerenciamento de informações de livros no sistema.
- Não Funcional: trata da compatibilidade e acessibilidade do sistema em diferentes navegadores.
- Funcional: define a funcionalidade de notificação por e-mail aos usuários.
- Funcional: especifica a função do sistema de manter um registro de transações.
- Funcional: descreve a capacidade do sistema de gerar relatórios de atividade.

- Não Funcional: refere-se à capacidade do sistema de suportar um número específico de usuários simultâneos, um requisito de escalabilidade.
- Não Funcional: relacionado à usabilidade e design da interface do usuário.

## Saiba mais

O Capítulo 4 do livro [Engenharia de software \(Sommerville\)](#) apresenta a engenharia de requisitos.

Quer ler mais sobre requisitos funcionais e não funcionais?! O artigo a seguir traz uma apresentação detalhada sobre o tema: [O que são requisitos funcionais e não funcionais?](#)

A elicitação de requisitos é uma etapa importante da engenharia de requisitos. Para saber mais sobre essa etapa, acesse o artigo a seguir: [Técnicas de elicitação de requisitos](#).

## Referências

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

REINEHR, S. **Engenharia de requisito**. Porto Alegre: SAGAH, 2020.

## Aula 4

Controle de Versões

### Controle de versões

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá o universo do gerenciamento de configuração de software, mergulhando no essencial controle de versão e nas poderosas ferramentas que o sustentam. Compreender esses elementos é crucial para sua prática profissional, proporcionando controle, colaboração eficiente e qualidade no desenvolvimento de software. Do controle de versão à seleção adequada de ferramentas, cada tópico abordado contribuirá diretamente para aprimorar suas habilidades e sua eficácia no ambiente de desenvolvimento. Vamos começar essa jornada prática rumo ao sucesso!

## Ponto de Partida

O desenvolvimento de um sistema é um esforço coletivo que demanda diversas competências devido à sua natureza complexa e às exigências cognitivas envolvidas. A colaboração de diferentes profissionais é essencial para lidar com a complexidade dos algoritmos e a ampla gama de elementos que compõem um software. A união dessas diversas habilidades traz benefícios significativos, mas também requer uma gestão cuidadosa e coordenada das diversas versões do produto em desenvolvimento para evitar a perda de controle sobre as mudanças realizadas.

Neste contexto, a aula foca em explorar os aspectos teóricos relacionados ao controle de versões em desenvolvimento de software. Isso inclui a discussão sobre a importância de um repositório eficaz e as estratégias para um compartilhamento eficiente de componentes do software. Além disso, o conteúdo também abrange o funcionamento das principais ferramentas utilizadas no gerenciamento de configuração de software, essenciais para manter a organização e a eficiência no processo de desenvolvimento.

A atividade proposta tem como objetivo familiarizar você com as ferramentas básicas de controle de versão usando Git e GitHub. Você irá criar um repositório no GitHub, adicionar um projeto existente a este repositório e, em seguida, realizar um commit com as alterações feitas. Este exercício é fundamental para desenvolver suas habilidades em gerenciamento de código e colaboração em projetos de desenvolvimento.

Ao concluir, você terá praticado a criação de um repositório no GitHub, o gerenciamento de versões com o Git e a colaboração em projetos de software. Essas são habilidades valiosas para qualquer profissional de TI.

Bons estudos!!!

## Vamos Começar!

### Gerenciamento de configuração de software

Embora estejamos focados no controle de versões, é essencial entender que ele faz parte de um contexto mais amplo, conhecido como Gerenciamento da Configuração do Software (GCS). De maneira concisa, o GCS oferece um controle rigoroso sobre o processo de desenvolvimento de software, o que é crucial devido à complexidade e ao volume de componentes envolvidos em um software (Leon, 2015).

A principal razão para o estabelecimento e aprimoramento do GCS reside na necessidade de gerenciar as alterações que ocorrem durante o desenvolvimento de um programa. Isso é alcançado por meio de métodos e ferramentas específicos, visando otimizar a produtividade e reduzir os erros ao longo do desenvolvimento do software.

A GCS é, portanto, um conjunto de práticas que controlam e notificam as inúmeras correções, extensões e adaptações aplicadas no software durante seu ciclo de vida, com o objetivo de assegurar um processo de desenvolvimento e de evolução organizado e passível de ser rastreado (Dantas, 2009).

O GSC atua como uma atividade de suporte vital para a gestão geral da qualidade do software. O fluxo de trabalho padrão desta área, ilustrado na Figura 1, destaca que, devido à possibilidade de mudanças ocorrerem a qualquer momento, é essencial que as atividades de gerenciamento de configuração abordem quatro aspectos-chave: (1) identificar a alteração, (2) controlar a alteração, (3) assegurar que a alteração esteja sendo implementada corretamente e (4) relatar as alterações a outros envolvidos.

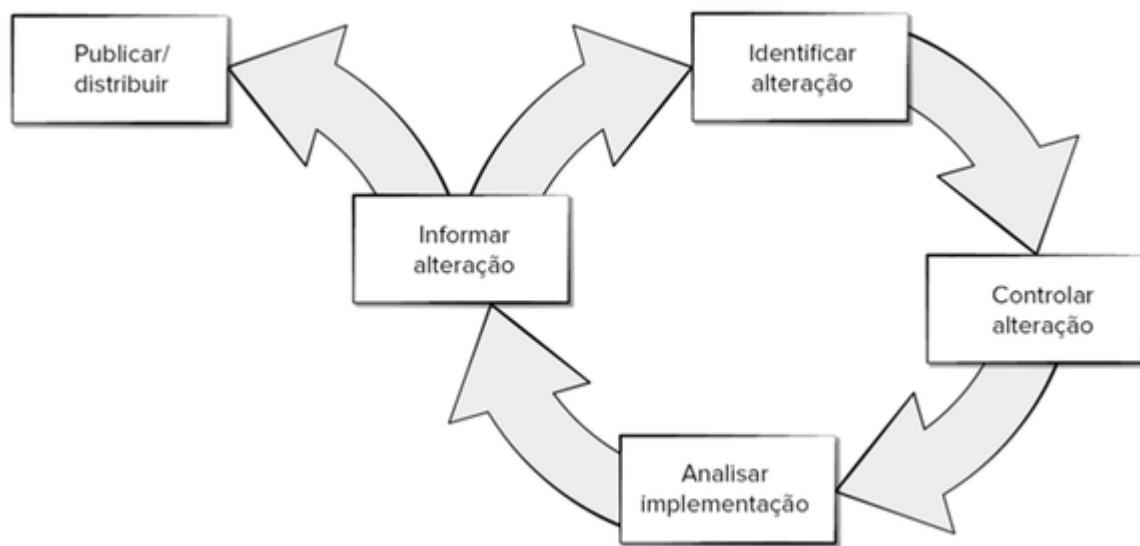


Figura 1 | Fluxo de trabalho do gerenciamento de configuração de software. Fonte: Pressman (2021).

Mesmo com a evidente necessidade de se gerenciar a configuração de um software, essa prática sofre com alguns mitos, muitos deles com potencial para desencorajar sua adoção nas organizações. Nas próximas linhas trataremos de três desses mitos e daremos bons motivos para que eles sejam desconstruídos, sempre com base na visão de Leon (2015).

O primeiro mito sugere que a implementação da gestão de configuração de software implica mais trabalho e novos procedimentos. Embora a transição para um sistema de gestão de configuração efetivo possa ser complexa, envolvendo o desenvolvimento de novas habilidades e procedimentos, os benefícios superam os desafios. A automação de tarefas repetitivas por ferramentas de gestão de configuração facilita o processo, economizando tempo e reduzindo erros.

O segundo mito é a noção de que a gestão da configuração é uma responsabilidade exclusiva da administração do sistema. Na realidade, é uma tarefa compartilhada por todos os envolvidos no desenvolvimento do software. Embora a administração desempenhe um papel crucial na criação de um ambiente propício para a gestão de configuração, uma equipe eficaz de gestão de configuração requer o suporte de toda a equipe de desenvolvimento.

Por fim, o terceiro mito é a ideia de que a gestão da configuração beneficia apenas os desenvolvedores. Apesar dos desenvolvedores serem os principais usuários e beneficiários de um sistema eficiente de gestão de configuração, a realidade é que outros profissionais, como testadores e equipes de manutenção e suporte, também se beneficiam. A educação sobre os princípios e benefícios da gestão de configuração pode ajudar a reduzir a resistência e maximizar a eficácia da implementação.

## Controle de versão

No contexto do Gerenciamento de Configuração de Software (GCS), um aspecto crucial é o controle de versões, que desempenha um papel vital na administração consistente das alterações feitas em um sistema. Segundo Caetano (2004), o controle de versões oferece várias funções essenciais para o gerenciamento eficaz de um projeto de software:

- Proporciona a capacidade de acessar versões anteriores do sistema, facilitando a restauração ou a análise de estados passados.
- Possibilita a auditoria das modificações, identificando quem fez as alterações, quando e o que foi alterado, garantindo transparência e rastreabilidade.
- Automatiza o acompanhamento de arquivos, simplificando a gestão de documentos e códigos do projeto.
- Facilita a visualização do status de um projeto em um momento específico, permitindo avaliações e comparações precisas.
- Ajuda a evitar conflitos entre os membros da equipe de desenvolvimento, especialmente em projetos com múltiplos colaboradores.
- Viabiliza o desenvolvimento simultâneo de um ou mais sistemas, promovendo a eficiência e a colaboração entre equipes.

Todas essas funções parecem convergir para um elemento bem definido: a centralização dos dados. E se não tivéssemos essa centralização? Bem, o uso descentralizado de um arquivo de código-fonte, por exemplo, permitiria que vários programadores acessassem vários arquivos e cada alteração feita nele não seria refletida nos demais. Assim, o desenvolvimento paralelo seria inviável e o conflito entre desenvolvedores inevitável. O recurso que as ferramentas de controle

de versão (abordadas a seguir) usam é o repositório, local em que programas em desenvolvimento, fotos, vídeos e demais arquivos ficam armazenados e podem ser acessados de forma controlada por todos os envolvidos no desenvolvimento do produto.

## ***Baselines (linhas de base)***

Dentro do escopo do controle de versões, um aspecto fundamental é a baseline de software. Essa baseline compreende um conjunto de itens de configuração que foram formalmente aprovados e servem como ponto de partida para as fases subsequentes do desenvolvimento. Ao final de uma iteração, quando ocorre uma entrega formal ao cliente, essa entrega é conhecida como release. Tanto baselines quanto releases são marcadas no repositório de software, geralmente por meio de etiquetas, também chamadas de tags, conforme explicado por Dantas (2009).

Baseline também se refere a uma versão específica de um item de configuração de software que foi acordada, e qualquer mudança em uma baseline só pode ocorrer por meio de procedimentos formais de controle de mudanças. Uma baseline, incluindo todas as alterações aprovadas nela, representa a configuração aprovada atual do software.

As baselines mais comuns são funcionais, alocadas, de desenvolvimento e de produto. A baseline funcional se relaciona aos requisitos do sistema que foram revisados. A baseline alocada inclui a especificação de requisitos de software revisada e a especificação de requisitos de interface, e a de desenvolvimento representa a configuração do software em evolução, em momentos específicos do ciclo de vida do software. Geralmente, a autoridade para alterar essa baseline é da organização de desenvolvimento, mas pode ser compartilhada com outras organizações. Por último, a baseline do produto se refere ao software completo, entregue para a integração do sistema, conforme descrito pelo IEEE (2004).

## ***BRANCHES***

A Gerência de Configuração de Software facilita a implementação simultânea de novas funcionalidades por diferentes membros da equipe, assegurando que as modificações sejam feitas de maneira isolada e independente. Esse isolamento é alcançado mediante a criação de ramificações, conhecidas como *branches*. As *codelines*, ou linhas de desenvolvimento, são atribuídas a cada projeto e utilizadas conjuntamente por diversos desenvolvedores. A primeira linha de desenvolvimento estabelecida em um projeto é geralmente chamada de *mainline*. *Branches* são ramificações criadas no repositório que representam linhas secundárias de desenvolvimento. Estas podem ser posteriormente reintegradas à linha principal, a *mainline*, por meio de uma operação chamada merge. De acordo com Dantas (2009), a capacidade de atender simultaneamente a diversas demandas de um projeto faz com que o uso de *branches* seja um aspecto crucial e diferenciador na gerência de configurações de software.

**Siga em Frente...**

## Ferramentas de controle de versões

Como era de se esperar, as funções do controle de versões são mais eficientemente realizadas por meio de ferramentas computacionais, oferecendo vantagens significativas em comparação com a execução manual dessas tarefas. Existem várias ferramentas de controle de versão de alta qualidade disponíveis no mercado, das quais três serão discutidas a seguir. A escolha da ferramenta mais apropriada varia de acordo com as necessidades específicas de cada organização. Contudo, é importante ter em mente que essas ferramentas não substituem outras funções críticas, como as de um compilador, um gerente de projeto, ou as de automação de testes. Com isso em mente, vamos agora examinar duas das ferramentas de controle de versão mais importantes e amplamente utilizadas.

### ***Git e Github***

Ao abordar as ferramentas Git e GitHub, é importante estabelecer a distinção entre elas. Embora compartilhem nomes semelhantes e sejam criações do mesmo desenvolvedor, desempenham funções diferentes. O Git é uma ferramenta de controle de versões amplamente utilizada por desenvolvedores, conhecida por seus avançados recursos de colaboração, que incluem fóruns de discussão para projetos em andamento e para tratativas de mudanças em andamento. Como em outras ferramentas de controle de versão, o Git oferece a funcionalidade de *branches*, permitindo que os membros da equipe trabalhem paralelamente em diferentes aspectos do projeto. Isso possibilita o gerenciamento de subprojetos individuais, como a correção de bugs ou o aprimoramento de funcionalidades, sem alterar os arquivos principais do repositório, além de permitir o compartilhamento desses experimentos com outros membros da equipe (Mailund, 2017).

O Git é um sistema de controle de versão gratuito e *open-source*, projetado para ser eficaz em todos os tipos de projetos, desde os menores até os maiores e mais complexos, envolvendo equipes amplas. Este sistema é conhecido por ser de fácil aprendizado, eficiente no uso do espaço de armazenamento e oferecer alto desempenho. Um dos aspectos mais importantes do Git é a segurança: seu modelo de dados assegura a integridade criptográfica de cada parte dos projetos. Cada arquivo e *commit* são submetidos a uma verificação de *checksum*, que também é aplicada durante a recuperação de dados, garantindo que apenas os dados exatos adicionados sejam recuperados.

O que realmente distingue o Git de quase todos os outros sistemas de controle de versão é seu modelo avançado de *branching* (ramificação). O Git permite a gestão de múltiplos *branches* locais, que podem operar independentemente uns dos outros. As operações de criação, fusão (merge) e exclusão de *branches* são realizadas em segundos, proporcionando uma troca rápida de contextos e a possibilidade de criar *branches* para testar novas ideias. Mesmo após *commits*, é possível retornar ao ponto de origem, ou mesmo aplicar patches com facilidade e eficiência.

Para iniciar seu cadastro no GitHub, visite a página de inscrição e preencha as informações necessárias. Uma vez escolhida a senha e realizada a verificação da conta, você será direcionado para uma página onde definirá aspectos como o principal tipo de trabalho que realiza, o nível de

experiência em programação, o propósito de uso do GitHub e áreas de interesse, para que a plataforma possa conectá-lo a comunidades similares. Por exemplo, você pode optar por identificar-se como estudante, com experiência básica em programação, interessado em aprender sobre Git e GitHub, e deixar em branco a questão sobre interesses específicos.

Depois de confirmar seu endereço de e-mail, você será levado para uma tela em que poderá escolher entre criar um repositório ou projeto, formar uma organização ou iniciar um aprendizado. Esta escolha definirá seus primeiros passos na plataforma, conforme mostra a Figura 2.

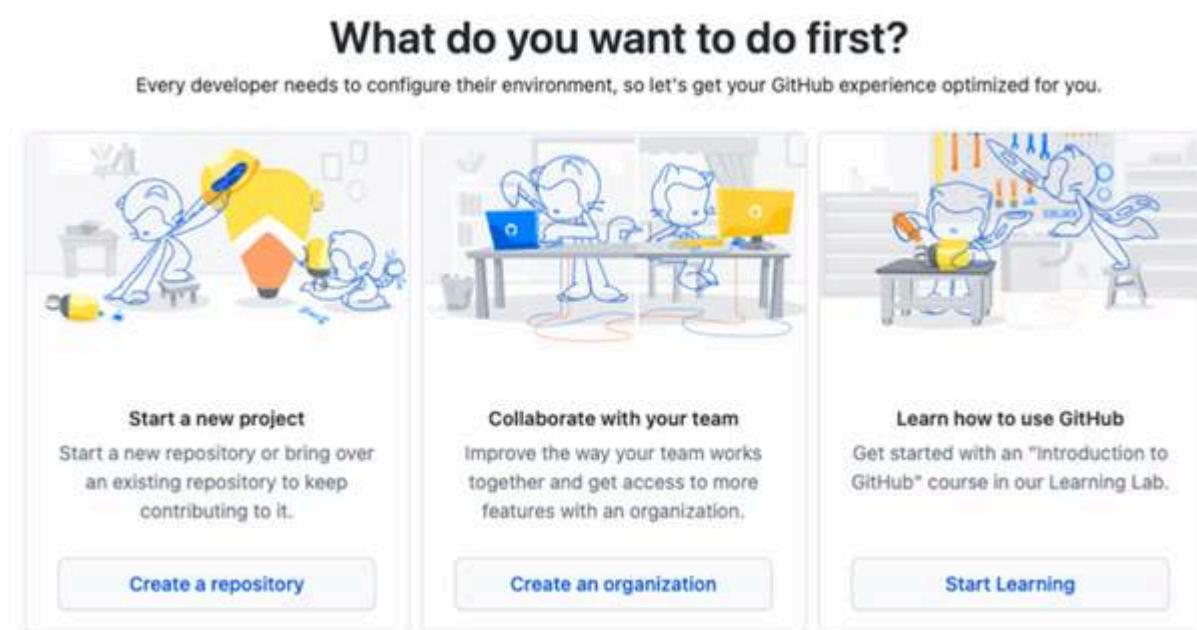


Figura 2 | Tela de escolha da ação inicial no GitHub. Fonte: Maitino (2021).

Quando você decide iniciar um novo projeto no GitHub, o primeiro passo é selecionar um nome para o seu repositório. Este nome será associado ao nome de usuário que você definiu durante o processo de inscrição. Por exemplo, no caso apresentado, o nome do repositório escolhido foi "engsoft" e a visibilidade definida foi pública, sem optar por qualquer recurso de inicialização. Em seguida, você será apresentado a uma tela com várias opções e configurações para o seu novo repositório, como pode ser visto na Figura 3.

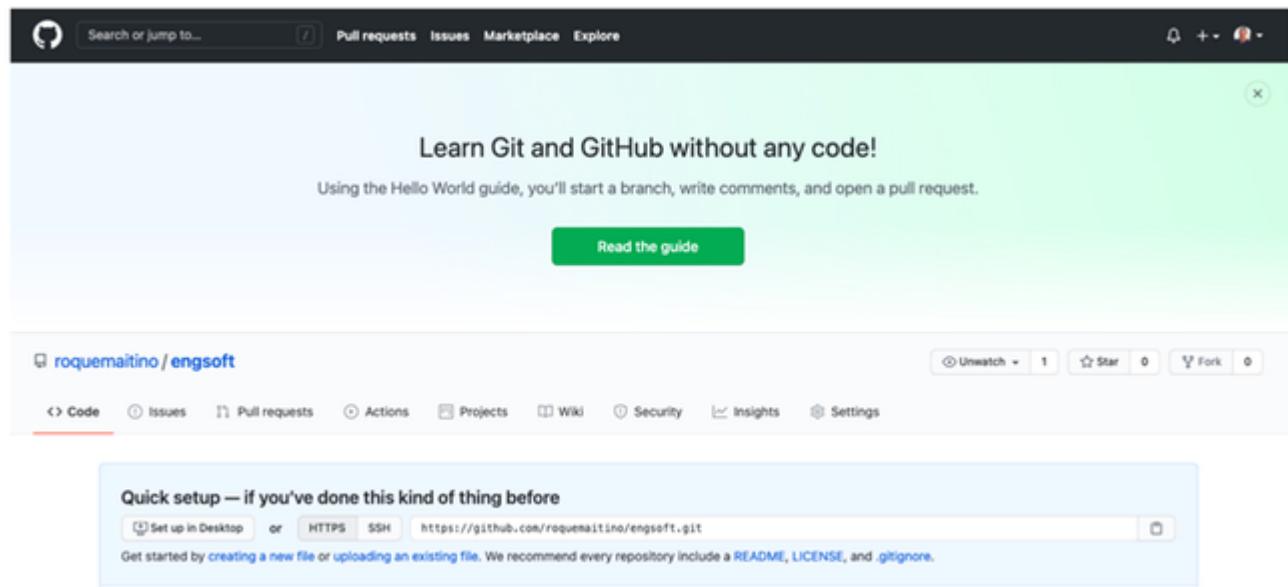


Figura 3 | Visão parcial da tela de opções do GitHub. Fonte: Maitino (2021).

A partir desse momento, você tem a liberdade de começar a criar seus próprios arquivos de código ou, se preferir, fazer o upload de arquivos já existentes em seu computador para o repositório. Observe que o nome de usuário e o nome do repositório (aparecem destacados na parte central e à esquerda da tela. Para adicionar um arquivo ao repositório, foi selecionada a opção de upload de um arquivo já existente, seguida do acionamento do botão de *Commit*. Nesse caso específico, o arquivo escolhido foi "Maior.java", um programa simples em Java que determina e exibe o maior número dentre oito valores fornecidos pelo usuário. Agora, ao visitar a página inicial desse perfil, é possível acessar a aplicação Java disponibilizada e até mesmo modificar seu código criando um *branch*.

## CVS

O CVS é uma ferramenta de código aberto (*open source*) que executa funções essenciais no processo de controle de versões. Esta ferramenta mantém um registro de todas as alterações feitas em um arquivo ao longo do tempo em seu repositório. Cada modificação no arquivo é identificada por um número único, conhecido como revisão. Essa revisão detalha as mudanças feitas no arquivo, quem as fez, quando foram feitas, entre outras informações relevantes. A representação visual na Figura 4 destaca as principais funções executadas pelo CVS, oferecendo uma compreensão clara de como ele administra e monitora as alterações em arquivos ao longo do tempo.

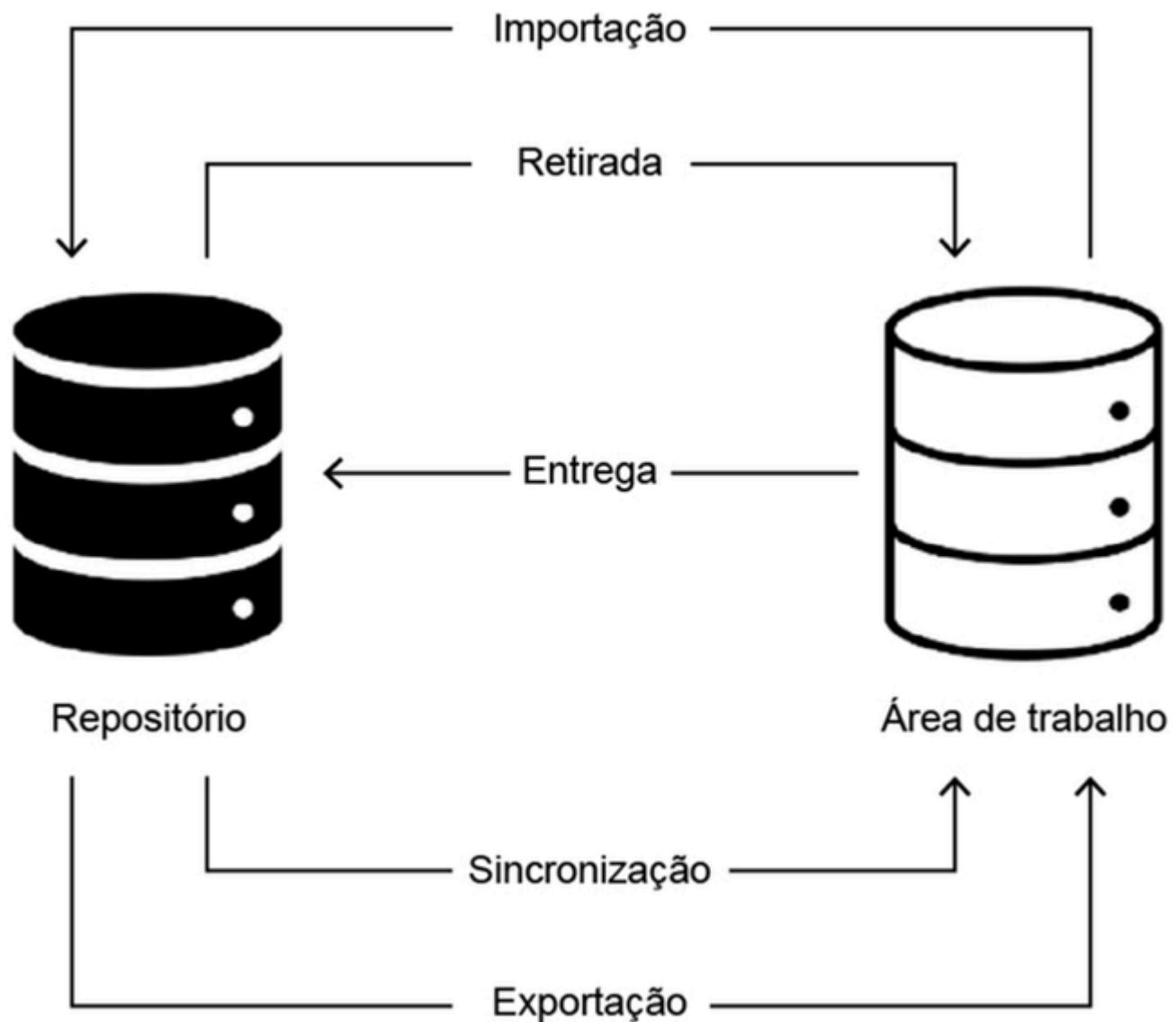


Figura 4 | Principais operações realizadas pelo CVS. Fonte: adaptada de Caetano (2004, p. 15).

O repositório do CVS, assim como os de outras ferramentas similares, armazena uma versão completa de todos os arquivos e pastas que estão sob o controle de versão. Geralmente, um desenvolvedor não acessa os arquivos no repositório diretamente. Ao invés disso, comandos específicos do CVS são utilizados para extrair uma cópia desses arquivos para um espaço de trabalho local, em que as modificações são realizadas. Após concluir as alterações, o desenvolvedor executa uma operação conhecida como "commit", enviando as mudanças de volta para o repositório central. Esse processo permite que o repositório registre detalhadamente as mudanças feitas, incluindo o que foi alterado, quando e outras informações pertinentes. É importante notar que o repositório e a área de trabalho não são subdiretórios um do outro, devendo estar localizados em locais distintos.

A estrutura do repositório é uma árvore de diretórios que reflete a organização dos diretórios na área de trabalho. Por exemplo, se o repositório está localizado em /usr/local/cvsroot, a estrutura

de diretórios seria similar à representada na Figura 5.

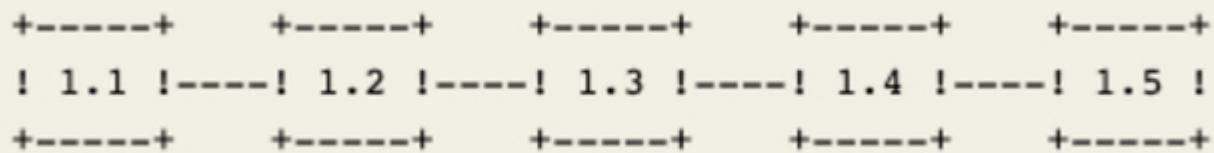
```
/usr
|
+--local
|   |
|   +--cvsroot
|       |
|       +--CVSROOT
|           |
|           |     (administrative files)
|
|   +--gnu
|       |
|       +--diff
|           |
|           |     (source code to GNU diff)
|
|       +--rcs
|           |
|           |     (source code to RCS)
|
|       +--cvs
|           |
|           |     (source code to CVS)
|
|   +--yoyodyne
|       |
|       +--tc
|           |
|           +--man
|           |
|           +--testing
|
|       +--(other Yoyodyne software)
```

Figura 5 | Uma possível estrutura de diretórios do repositório CVS. Fonte: GNU ([s. d.]).

Dentro da estrutura de diretórios do repositório CVS, cada arquivo sob controle de versão possui um arquivo de histórico associado. O nome desse arquivo de histórico é derivado do nome do arquivo original, com a adição de ',v' no final. Por exemplo, "index.php,v" e "frontend.c,v" são exemplos de como esses arquivos de histórico podem ser nomeados. Esses arquivos de histórico contêm informações cruciais, como todas as revisões do arquivo, um registro de todas as mensagens de *commit* associadas e o nome do usuário que fez cada revisão. Esses arquivos são chamados de arquivos RCS, um acrônimo para *Revision Control System*, que foi o primeiro sistema de controle de versão a utilizar esse formato para armazenar informações de arquivo (GNU, [s. d.]).

No CVS, cada modificação em um programa resulta em uma nova versão identificada por um número exclusivo. O sistema CVS designa automaticamente números como 1.1, 1.2, e assim sucessivamente, para cada versão criada. Um único arquivo pode ter várias versões, e o mesmo vale para um produto de software completo, que pode ser identificado por um número de versão mais complexo, como "4.1.1". Cada versão de um arquivo é marcada com um número de revisão único, que pode variar de "1.1" a formatos mais complexos como "1.3.2.2" ou "1.3.2.2.4.5".

Esses números de revisão seguem um padrão de dois números decimais separados por pontos. Por convenção, a revisão "1.1" é a primeira de um arquivo, e cada revisão subsequente recebe um incremento no número mais à direita. Na Figura 6, é possível visualizar algumas dessas revisões, com as mais recentes posicionadas à direita. Revisões com mais de dois segmentos, como "1.3.2.2", também são possíveis e indicam uma estrutura mais complexa de controle de versões.



```
+----+      +----+      +----+      +----+      +----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !
+----+      +----+      +----+      +----+      +----+
```

Figura 6 | Representação de revisões mais recentes no número à direita. Fonte: GNU ([s. d.]).

Antes de encerrarmos o conteúdo do CVS, vale tratarmos de mais algumas terminologias relacionadas ao assunto (GNU, [s. d.]).

- **Checkout:** denominação dada à primeira recuperação (ou download) de um módulo do sistema vindo do repositório do CVS.
- **Commit:** trata-se do envio do artefato modificado ao repositório do CVS.
- **Export (ou exportação):** trata-se da recuperação (ou download) de um módulo inteiro a partir de um repositório, sem os arquivos administrativos CVS. Módulos exportados não ficam sob controle do CVS.
- **Import (ou importação):** esse termo identifica a criação de um módulo completo no âmbito de um repositório CVS, feita por meio de um upload de uma estrutura de diretórios.
- **Module (ou módulo):** é a representação de uma hierarquia de diretórios. Um projeto de determinado software efetiva-se como um módulo dentro do repositório.

- **Release:** este termo equivale a um “lançamento”. Um número de release identifica a versão de um produto completo ou inteiro.
- **Merge:** refere-se à fusão das diversas modificações feitas por diferentes usuários na cópia local de um mesmo arquivo. Sempre que alguém altera o código, é necessário realizar uma atualização antes da aplicação da operação de commit, a fim de que seja feita a fusão das mudanças.

Este foi, portanto, o conteúdo que queríamos compartilhar com você. O entendimento de importância do gerenciamento da configuração de um software e a familiaridade com os termos e o funcionamento de uma ferramenta de controle de versões são habilidades imprescindíveis para o desenvolvedor inserido em uma equipe de trabalho e ao gestor do software. Esperamos que este conteúdo seja útil a você em breve.

## Vamos Exercitar?

Esta atividade tem como objetivo familiarizar o aluno com as ferramentas básicas de controle de versão usando Git e GitHub. O aluno aprenderá a criar um repositório no GitHub, adicionar um projeto a este repositório e fazer um commit das alterações realizadas.

Instruções:

### 1. Crie uma Conta no GitHub (se você ainda não tiver uma):

- Acesse [github.com] (<https://github.com>) e crie uma conta gratuita.
- Preencha as informações necessárias e complete o processo de registro.

### 2. Instale o Git:

- Baixe e instale o Git em seu computador, seguindo as instruções no site oficial: [git-scm.com](<https://git-scm.com>).

### 3. Configure o Git:

- Abra o terminal ou prompt de comando e configure seu nome de usuário e e-mail do Git com os seguintes comandos:

```
git config --global user.name "Seu Nome"  
git config --global user.email seuemail@example.com
```

### 4. Crie um Novo Repositório no GitHub:

- Acesse sua conta no GitHub.
- Clique em "New repository" e crie um novo repositório com um nome apropriado.

- Marque a opção para inicializar o repositório com um README.

## 5. Clone o Repositório em sua Máquina Local:

- Copie o URL do repositório recém-criado.
- No terminal, navegue até o diretório onde deseja clonar o repositório e execute:

```
git clone [URL do repositório]
```

## 6. Adicione um Projeto ao Repositório:

- Copie os arquivos do seu projeto para o diretório do repositório clonado.
- No terminal, navegue até o diretório do repositório.

## 7. Faça o Primeiro Committ:

- Adicione os arquivos ao Git com:

```
git add .
```

- Faça o commit das alterações com uma mensagem descritiva:

```
git commit -m "Primeiro commit: adicionando projeto ao repositório"
```

## 8. Envie as Alterações para o GitHub:

- Faça o push das alterações para o GitHub com:

```
git push origin master
```

## Saiba mais

A gestão de configuração é a arte de identificar, organizar e controlar modificações no software que está em construção por uma equipe de programação. O objetivo é maximizar a produtividade minimizando os erros. Para saber mais sobre a gestão de configuração, o Capítulo 22 do [Engenharia de Software \(Pressman\)](#) aborda mais sobre esse assunto.

Quer aprender a configurar o Git?! Leia o artigo indicado: [Configurar o Git](#).

Acesse o artigo indicado a seguir, para saber mais sobre [Instalação, Configuração e Primeiros Passos](#).

## Referências

- CAETANO, C. **CVS** – Controle de Versões e Desenvolvimento Colaborativo de Software. São Paulo: Novatec Editora, 2004.
- GITHUB. J. **GitHub**, [S. l.], c2020. Disponível em: <https://github.com/join>. Acesso em: 23 jan. 2024.
- GNU. **The Repository**. GNU, [S. l.: s. d.]. Disponível em: [https://www.gnu.org/software/trans-coord/manual/cvs/html\\_node/Repository.html](https://www.gnu.org/software/trans-coord/manual/cvs/html_node/Repository.html). Acesso em: 23 jan. 2024.
- LEON, A. **Software Configuration Management Handbook**. 3. ed. Boston: Artech House, 2015.
- MAILUND, T. **The Beginner's Guide to GitHub**. [S. l.: s. n.], 2017.
- MAITINO NETO, R. **Engenharia de software**. Londrina: Editora e Distribuidora Educacional S.A., 2021.
- PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 9. ed. Porto Alegre: AMGH, 2021.

## Aula 5

Encerramento da Unidade

### Videoaula de Encerramento



#### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá o universo do desenvolvimento de software, explorando o processo de software, métodos ágeis e requisitos essenciais. Compreender esses elementos é crucial para a excelência profissional, pois moldam a forma como as equipes colaboram, entregam projetos e atendem às necessidades do cliente de maneira eficaz. Prepare-se para aprimorar suas habilidades e se destacar no dinâmico cenário da tecnologia.

## Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é compreender os fundamentos de processos de softwares tradicionais e ágeis, você deverá primeiramente conhecer os conceitos dos princípios da engenharia de software.

A Engenharia de Software é um processo que comprehende uma série de métodos e ferramentas voltados para a criação de software de alta qualidade. Este campo é dinâmico, adaptando-se continuamente às mudanças nas demandas e exigências dos programas de computador, com um foco crescente na qualidade e rapidez de entrega. Entender a Engenharia de Software é fundamental para a implementação de práticas eficazes que atendam às necessidades das organizações. Essa compreensão envolve adaptar conceitos à realidade individual ou organizacional, mantendo a essência dos princípios fundamentais. A "Engenharia" refere-se ao design e à produção de um artefato, com ênfase em requisitos e especificações, enquanto o "Software" comprehende instruções executáveis, estruturas de dados e documentação relacionada. Segundo a IEEE Computer Society, a Engenharia de Software é a prática de aplicar abordagens sistemáticas, disciplinadas e quantificáveis ao desenvolvimento, operação e manutenção de software.

A Engenharia de Software tem como elemento central o próprio software, o qual abrange não apenas os programas executáveis em computadores, mas também os dados e informações gerados durante a execução desses programas e a documentação necessária para referência e manutenção. Este conceito ampliado enfatiza que o software vai além do código executável, incluindo dados processados e documentação essencial (Pressman, 2021). O software desempenha um papel fundamental na transmissão de informações, tornando-se um componente crucial em diversas áreas da vida moderna. No entanto, a criação de software não é uma tarefa simples, tendo evoluído de uma atividade artesanal para um processo sistematizado e estruturado, exigindo abordagens metodológicas avançadas.

Durante a década de 1960, a produção de software enfrentou desafios significativos, conhecidos como "crise do software". Esses desafios incluíam ineficiências, desenvolvimento falho, manutenção complicada e comunicação deficiente entre clientes e desenvolvedores, resultando em requisitos inadequados e produtos finais com falhas. A manutenção dos sistemas era problemática devido a projetos mal planejados, e a fase de implementação frequentemente negligenciava impactos organizacionais e treinamento de usuários. Além disso, alguns mitos sobre o desenvolvimento de software persistem até hoje, como a ideia de que adicionar mais programadores não acelera necessariamente o desenvolvimento, mudanças tardias em software podem ser custosas, software funcional não significa que está pronto para entrega, software

requer manutenção constante, engenharia de software vai além da programação, e habilidades de programação não são equivalentes a habilidades de gerenciamento. Reconhecer e abordar esses mitos é vital para uma gestão eficaz de projetos de software.

Um modelo de processo de software é um conjunto de práticas e procedimentos que orientam equipes de desenvolvimento na criação de softwares, servindo como um roteiro detalhado que abrange desde a concepção até a entrega e manutenção de um projeto de software. Esse modelo proporciona uma estrutura para organizar e controlar diversas atividades do desenvolvimento de software, como análise de requisitos, design, codificação, testes e manutenção, com o objetivo de garantir a qualidade do software, eficiência no desenvolvimento e satisfação das necessidades do cliente. O modelo é adaptável às características específicas de cada projeto, considerando fatores como o tamanho da equipe, complexidade do software, orçamento e prazos, e é essencial para rastrear o progresso, identificar riscos e gerenciar mudanças eficientemente (Pressman, 2021).

O processo de software, por sua vez, é a aplicação prática desse modelo, envolvendo a execução real das atividades de desenvolvimento dentro do quadro fornecido pelo modelo selecionado. Isso inclui a definição de tarefas, alocação de recursos, gestão do tempo e coordenação entre diferentes equipes e atividades. Durante o processo, as equipes trabalham juntas para definir requisitos, projetar a arquitetura do software, escrever e testar o código, e entregar um produto que atenda às expectativas do cliente. Após a entrega, o software normalmente passa por fases de manutenção e atualização para garantir sua eficiência e relevância ao longo do tempo. Assim, enquanto o modelo de processo de software fornece a estrutura teórica, o processo de software é a implementação prática dessa estrutura, adaptada às necessidades específicas de cada projeto, sendo ambos essenciais para o sucesso na produção de software de alta qualidade.

Outro assunto relevante são os métodos ágeis. Em 2001, um grupo de desenvolvedores, escritores e consultores formulou o *Manifesto para o Desenvolvimento Ágil de Software*, marcando um ponto de virada na abordagem do desenvolvimento de software. Esse manifesto enfatizou a importância de indivíduos e interações, software funcional, colaboração com clientes e adaptabilidade a mudanças, em vez de seguir rigidamente processos, documentação extensa, negociações contratuais e planos inflexíveis. Esse foco em flexibilidade, colaboração e adaptabilidade trouxe uma nova perspectiva ao desenvolvimento de software, destacando-se da estrutura e processos rígidos tradicionais.

A adoção do desenvolvimento ágil trouxe métodos inovadores para abordar desafios e falhas da engenharia de software tradicional. Essa abordagem é particularmente valiosa em ambientes de negócios dinâmicos e incertos, onde os requisitos podem mudar rapidamente e a previsibilidade é limitada. Os processos ágeis minimizam os custos associados a mudanças e são moldados por princípios que reconhecem a natureza mutável dos projetos de software. Esses princípios incluem a entrega incremental de software, feedback constante dos clientes e adaptação rápida a mudanças. A Agile Alliance destaca a importância de satisfazer o cliente com entregas rápidas, manter uma comunicação próxima e valorizar indivíduos motivados e auto-organizados em equipes. Apesar da variabilidade entre diferentes modelos ágeis, esses princípios formam a base do espírito ágil, promovendo um desenvolvimento de software eficaz e adaptativo.

O método ágil mais utilizado é o Scrum. Esse método utiliza os princípios do desenvolvimento ágil, estrutura o processo de desenvolvimento de software em etapas distintas, como definição de requisitos, análise, design, evolução e entrega. Essas etapas são executadas em períodos definidos chamados sprints, cuja duração varia de acordo com o projeto. Dentro de cada sprint, as tarefas são adaptadas às necessidades específicas do projeto e frequentemente revisadas pela equipe, proporcionando um fluxo de trabalho flexível e responsável a mudanças.

Cada sprint no Scrum começa com a definição do *Product Backlog*, uma lista dinâmica de funcionalidades desejadas, seguida pela seleção de tarefas para o Sprint Backlog. Durante o sprint, a equipe se concentra em desenvolver as funcionalidades escolhidas, com o *Product Owner* gerenciando e atualizando o Backlog da Sprint. O Scrum enfatiza a auto-organização das equipes, que são apoiadas por figuras-chave como o Scrum Master, que facilita o processo, e o *Product Owner*, que define as prioridades do projeto.

As reuniões do Scrum, ou cerimônias, estruturam o fluxo de trabalho do projeto. Elas incluem o Planejamento do Sprint, as Reuniões Diárias para compartilhar progressos e desafios, a Revisão do Sprint para apresentar o trabalho realizado, e a Retrospectiva do Sprint para discutir melhorias no processo. A comunicação eficaz é essencial em ambientes ágeis, envolvendo não apenas os membros da equipe, mas também o cliente, assegurando que as expectativas sejam claras e atendidas.

Além disso, o quadro Scrum, uma ferramenta visual não oficial, é frequentemente utilizado para organizar e monitorar as atividades do projeto. Ele é dividido em categorias como "a fazer", "em execução" e "concluída", com post-its coloridos representando diferentes tipos de tarefas ou responsabilidades. Esta organização visual facilita o entendimento do progresso do projeto e promove uma melhor coordenação das atividades da equipe (Pressman, 2021). Em resumo, o Scrum oferece uma estrutura ágil e adaptável para o desenvolvimento de software, priorizando a comunicação, colaboração e resposta rápida às mudanças.

Outro método ágil é o *Extreme Programming* (XP), que também segue um conjunto de práticas em áreas metodológicas principais como planejamento, projeto, codificação e testes. A XP começa com a fase de planejamento, conhecida como "o jogo do planejamento", em que se escuta atentamente os clientes para criar histórias de usuário que detalham os resultados esperados e as características do software. Estas histórias são priorizadas com base no valor de negócio e estimadas em semanas de desenvolvimento pela equipe XP. Durante o planejamento, a equipe decide quais histórias incluir no próximo incremento de software, levando em consideração fatores como valor e risco. A "velocidade do projeto" é calculada após a entrega do primeiro incremento, auxiliando na estimativa para futuras versões (Sbrocco, 2021).

Na fase de projeto, o princípio KISS (*Keep It Simple, Stupid*) é seguido para manter a simplicidade no design, e cartões Classe-Responsabilidade-Colaborador (CRC) são utilizados para estruturar o software em um contexto orientado a objetos. Protótipos operacionais podem ser criados para desafios complexos de projeto e a refatoração é uma prática contínua. Na codificação, testes unitários são escritos para cada história, guiando os desenvolvedores na implementação necessária. A programação em pares é uma prática chave na XP, proporcionando solução de problemas em tempo real e garantia de qualidade. A integração contínua do código é crucial para

evitar problemas de compatibilidade. Testes unitários automatizados facilitam a execução frequente de testes de regressão.

A XP também enfatiza os testes de aceitação, especificados pelo cliente, que avaliam características e funcionalidades do sistema completo visíveis ao cliente. Estes testes são derivados das histórias de usuário implementadas em cada versão do software, assegurando que o produto atenda às expectativas e requisitos do cliente (Sbrocco, 2021). Em resumo, a XP é uma metodologia que combina práticas rigorosas com uma abordagem flexível e adaptável ao desenvolvimento de software, enfatizando a colaboração entre clientes e desenvolvedores, e a capacidade de responder rapidamente a mudanças.

Requisitos são especificações essenciais no desenvolvimento de software, delineando as expectativas, funcionalidades e parâmetros operacionais de um sistema. A correta identificação e documentação dos requisitos são fundamentais para guiar o processo de desenvolvimento, assegurando que o produto final atenda às necessidades dos usuários e aos objetivos do projeto. Requisitos bem definidos e gerenciados minimizam os riscos de mal-entendidos, retrabalhos e falhas no desenvolvimento, contribuindo para a eficiência, eficácia e sucesso do software.

Os requisitos funcionais de um sistema são especificações que definem as ações e as funcionalidades que o sistema deve executar, variando de acordo com o tipo de software, o perfil dos usuários e a metodologia adotada pela organização. Eles podem ser apresentados de forma abstrata como requisitos de usuário para fácil compreensão, ou detalhados minuciosamente como requisitos de sistema, incluindo entradas, saídas e exceções. A amplitude dos requisitos funcionais pode variar desde exigências gerais até requisitos específicos, refletindo as particularidades dos sistemas e processos de trabalho de uma organização. Exemplos de requisitos funcionais incluem autenticação de usuários, gerenciamento de pedidos, geração de relatórios e envio automático de notificações por e-mail. A falta de clareza na definição de requisitos pode levar a problemas na engenharia de software, resultando em atrasos e custos adicionais. Idealmente, os requisitos funcionais devem ser completos e consistentes, mas alcançar esses critérios é desafiador, especialmente em sistemas grandes e complexos, devido à possibilidade de erros, omissões e inconsistências decorrentes das diferentes necessidades e expectativas dos múltiplos stakeholders (Sommerville, 2018).

Requisitos não funcionais são cruciais no desenvolvimento de software, definindo critérios de qualidade e operacionais que não estão diretamente relacionados às funcionalidades específicas do sistema. Eles abrangem aspectos como confiabilidade, tempo de resposta, eficiência e segurança, e podem impor restrições na implementação do sistema, como capacidades dos dispositivos de entrada/saída e formatos de dados em interfaces. Esses requisitos são frequentemente mais críticos que os funcionais, pois falhar em atendê-los pode comprometer todo o sistema. Um desafio comum é traduzir metas abstratas dos clientes em requisitos não funcionais mensuráveis e testáveis, que são fundamentais para garantir a qualidade do software.

Os requisitos não funcionais podem impactar a arquitetura global do sistema e gerar requisitos funcionais relacionados, e frequentemente interagem ou entram em conflito com outros requisitos. Transformar metas gerais dos clientes em requisitos quantitativos é complexo, e os

clientes podem ter dificuldades em compreender ou justificar os custos associados à verificação desses requisitos. Na documentação de requisitos, é importante separar e enfatizar claramente os requisitos não funcionais, apesar do desafio de manter a relação compreensível entre eles e os requisitos funcionais. Essa separação ajuda a garantir que todas as propriedades emergentes do sistema, como desempenho e confiabilidade, sejam adequadamente consideradas e testadas.

A engenharia de requisitos envolve quatro atividades principais: avaliação da viabilidade, identificação dos requisitos, formalização em um formato padronizado e validação para garantir que atendam às necessidades do cliente. Este processo é iterativo, com as atividades ocorrendo de forma entrelaçada, não linear. Inicialmente, o foco recai sobre a compreensão dos requisitos de negócio e não funcionais, avançando para a elicitação e para o detalhamento dos requisitos do sistema. A engenharia de requisitos ajusta-se às necessidades específicas de cada organização, variando de acordo com fatores como especialização dos funcionários e tipo de sistema em desenvolvimento (Sommerville, 2018).

A elicitação e a análise de requisitos envolvem a colaboração com clientes e usuários finais para coletar informações relevantes, como desempenho esperado e limitações de hardware. Stakeholders, que exercem influência direta ou indireta sobre os requisitos, incluem usuários finais e outros membros da organização. A especificação de requisitos documenta os requisitos de usuário e de sistema, buscando clareza e consistência, apesar dos desafios práticos. A validação de requisitos é crucial para evitar problemas significativos, utilizando técnicas como revisões de requisitos, prototipagem e geração de casos de teste. A validação assegura que o sistema atenda às necessidades reais dos usuários e é adaptável a mudanças. Apesar da complexidade e dos desafios, a engenharia de requisitos é essencial para o desenvolvimento bem-sucedido de sistemas de software, abordando tanto a teoria quanto a prática dos requisitos do sistema.

Outro assunto relevante para alcançarmos a competência da nossa unidade é o controle de versões. O Gerenciamento da Configuração do Software (GCS) é uma prática essencial no desenvolvimento de software, proporcionando um controle rigoroso sobre o processo, o que é crucial devido à complexidade e ao volume de componentes envolvidos. O GCS visa gerenciar as mudanças durante o desenvolvimento, otimizando a produtividade e reduzindo erros. As práticas de GCS controlam e notificam as correções, extensões e adaptações aplicadas ao software ao longo de seu ciclo de vida, garantindo um processo de desenvolvimento organizado e rastreável.

O GCS é vital para a gestão da qualidade do software, abordando quatro aspectos-chave: identificação, controle, implementação correta e relato de alterações. Apesar de sua importância, a prática enfrenta mitos que podem desencorajar sua adoção. Um desses mitos é a crença de que a gestão de configuração implica mais trabalho e novos procedimentos, mas os benefícios, como a automação de tarefas repetitivas, superam os desafios iniciais. Outro mito é que a gestão de configuração é exclusiva da administração do sistema, enquanto na verdade é uma responsabilidade compartilhada por toda a equipe de desenvolvimento. Por fim, há um mito de que somente os desenvolvedores se beneficiam da gestão de configuração, quando na realidade outros profissionais, como testadores e equipes de manutenção e suporte, também se beneficiam. A conscientização sobre os princípios e os benefícios do GCS pode ajudar a reduzir a resistência e maximizar a eficácia da implementação.

No Gerenciamento de Configuração de Software (GCS), o controle de versões é um componente-chave, oferecendo várias funções essenciais para o gerenciamento eficaz de um projeto de software. Ele permite acessar versões anteriores do sistema, facilita a auditoria das modificações, automatiza o rastreamento de arquivos, ajuda a visualizar o status do projeto em momentos específicos, evita conflitos entre membros da equipe e viabiliza o desenvolvimento paralelo de sistemas. A centralização de dados em um repositório é crucial para essas funções, permitindo acesso controlado a arquivos por todos os envolvidos no desenvolvimento (Pressman, 2021).

Dentro do controle de versões, as baselines são pontos de referência importantes, representando conjuntos de itens de configuração formalmente aprovados que servem de base para fases subsequentes do desenvolvimento. As baselines podem ser funcionais, alocadas, de desenvolvimento ou de produto, cada uma relacionada a diferentes aspectos e estágios do software. Além disso, o conceito de branches é fundamental no GCS, permitindo o desenvolvimento simultâneo e isolado de novas funcionalidades por diferentes membros da equipe, com a possibilidade de reintegração posterior à linha principal de desenvolvimento, a mainline. Essas práticas de GCS, como o controle de versões, baselines e branches, são essenciais para um desenvolvimento de software organizado, eficiente e adaptável a mudanças.

O controle de versões é mais eficiente quando realizado por meio de ferramentas computacionais. Essas ferramentas proporcionam vantagens significativas, como o acesso a versões anteriores do sistema, auditoria de modificações, automação no acompanhamento de arquivos e facilidade na visualização do status do projeto. Dentre as ferramentas de controle de versões disponíveis, Git e GitHub são amplamente utilizadas. Git é um sistema de controle de versão gratuito e *open-source*, conhecido por sua segurança, eficiência e avançado modelo de *branching* (GITHUB, 2020). Ele permite o gerenciamento paralelo de subprojetos, facilitando a colaboração e a experimentação sem afetar o repositório principal.

O *Concurrent Versions System* (CVS) é outra ferramenta de código aberto para controle de versões, que mantém um registro detalhado das alterações em arquivos ao longo do tempo. Cada modificação é identificada por um número de revisão exclusivo, com um repositório armazenando versões completas de todos os arquivos. O CVS utiliza comandos específicos para gerenciar a interação entre o repositório central e o espaço de trabalho local dos desenvolvedores. Ele também emprega terminologias como *checkout*, *commit*, *export*, *import*, *module*, *release* e *merge* para facilitar a gestão de mudanças e colaboração entre vários usuários. Essas ferramentas de controle de versões são essenciais para um gerenciamento eficaz de projetos de software, permitindo rastreamento preciso e colaboração eficiente no desenvolvimento.

## É Hora de Praticar!

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

O mundo do desenvolvimento de software, a escolha da metodologia de gerenciamento de projetos desempenha um papel crucial no sucesso e na eficiência da entrega de um sistema. Duas abordagens amplamente utilizadas, mas muitas vezes contrastantes, são os métodos tradicionais, os métodos ágeis. Além disso, o controle de versões desempenha um papel vital na administração consistente das alterações feitas em um sistema.

Este exercício explorará esses três tópicos fundamentais e como eles se relacionam no contexto do desenvolvimento de software. Os métodos tradicionais representam uma abordagem sequencial e linear para gerenciar projetos, enquanto os métodos ágeis promovem uma abordagem iterativa e adaptativa. Cada abordagem tem suas vantagens e desvantagens, e a escolha entre elas pode afetar significativamente o resultado do projeto. Por outro lado, o controle de versões oferece a capacidade de gerenciar alterações de forma organizada, rastreável e colaborativa, independentemente da metodologia adotada.

Uma startup deseja desenvolver um sistema de reservas de hotéis para oferecer aos clientes uma plataforma fácil de usar para reservar acomodações em todo o mundo. Eles têm um prazo apertado para lançar o sistema, mas também querem garantir que ele seja flexível o suficiente para acomodar futuras expansões e melhorias.

- Como os métodos de processos tradicionais podem se adaptar e evoluir para enfrentar os desafios e as demandas das práticas modernas de desenvolvimento de software, levando em consideração a crescente necessidade de agilidade, flexibilidade e inovação?
- Como os métodos ágeis têm impactado positivamente a forma como as equipes de desenvolvimento de software abordam os projetos, promovendo a colaboração, a adaptação a mudanças e a entrega contínua de valor aos clientes, e quais são os desafios enfrentados na implementação eficaz dessas abordagens ágeis nas organizações?
- Como podemos equilibrar a necessidade de documentar e definir requisitos detalhados em projetos de desenvolvimento de software com a agilidade e a flexibilidade necessárias para acomodar mudanças e evolução ao longo do ciclo de vida do projeto?

## Dê o Play!

[Clique aqui](#) para acessar os slides do Dê o play!

A startup decidiu adotar uma abordagem híbrida para o desenvolvimento de seu sistema de reservas de hotéis, combinando elementos do método cascata e do Scrum.

### Fase 1: Método Cascata (Análise e Planejamento Iniciais)

Na primeira fase, a equipe da empresa usou o método cascata para realizar uma análise detalhada dos requisitos do sistema. Eles conduziram entrevistas com clientes em potencial, identificaram os recursos essenciais e criaram uma especificação detalhada do sistema.

Nesta fase, eles também definiram a arquitetura do sistema, escolheram as tecnologias apropriadas e elaboraram um plano de desenvolvimento detalhado. Isso ajudou a estabelecer

uma base sólida para o projeto e garantiu que todos os requisitos essenciais fossem bem compreendidos e documentados.

### **Fase 2: Scrum (Desenvolvimento Iterativo)**

Com a especificação e o plano em mãos, a empresa começou a implementação usando o Scrum. Eles dividiram o projeto em iterações de duas semanas, conhecidas como Sprints. Durante cada Sprint, a equipe se concentrou em implementar um conjunto específico de recursos.

Isso permitiu que eles respondessem rapidamente a mudanças nos requisitos à medida que surgiam. Por exemplo, se os clientes expressassem a necessidade de um novo recurso de pesquisa avançada, a equipe poderia priorizá-lo e adicioná-lo em uma Sprint subsequente.

### **Fase 3: Controle de Versões (Git)**

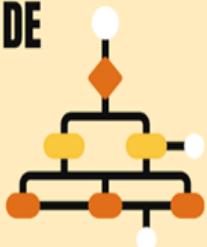
Para gerenciar as alterações de código e colaborar eficazmente, a equipe da empresa usou um sistema de controle de versões, como o Git. Isso garantiu que todas as alterações fossem rastreadas, facilitando a reversão a versões anteriores, se necessário. Eles criaram branches para desenvolver novos recursos isoladamente e, em seguida, fizeram merges para incorporar essas alterações à versão principal do sistema.

### **Resultados**

Ao adotar essa abordagem híbrida, a "startup" conseguiu lançar seu sistema de reservas de hotéis dentro do prazo apertado, ao mesmo tempo em que manteve a flexibilidade para acomodar mudanças nos requisitos. A combinação do método cascata para análise e planejamento iniciais com o Scrum para desenvolvimento iterativo, juntamente com o controle de versões, permitiu que eles gerenciassem eficazmente o projeto.

Este estudo de caso demonstra como a aplicação adequada das metodologias do método cascata, Scrum e controle de versões pode ser benéfica ao desenvolver um sistema de software complexo em um ambiente dinâmico e com prazos apertados.

A imagem a seguir apresenta visualmente as diferenças entre abordagens de desenvolvimento de software. Descubra como os Métodos Ágeis oferecem flexibilidade e adaptação contínua, contrastando com o Modelo Cascata, que segue uma abordagem sequencial e rígida. Este infográfico oferece insights visuais para ajudá-lo a escolher a melhor abordagem para seu projeto, destacando as vantagens e os desafios de cada método.



**METODOLOGIAS DE GESTÃO DE PROJETOS**

## Metodologias ágeis

A metodologia de gerenciamento de projetos ágil oferece um processo flexível e iterativo de design e construção. O Ágil vai além de uma metodologia. Ele abrange um conjunto de processos para projetos extensos em ambientes dinâmicos.

## Método Cascata

O modelo cascata é uma divisão das atividades do projeto em fases lineares sequenciais, onde cada fase depende dos entregáveis da anterior e corresponde à especialização de tarefas.



## Scrum

O Scrum se concentra na colaboração, transparência e adaptação contínua. Ele é conhecido por dividir o trabalho em ciclos curtos chamados "sprints", geralmente com duração de 2 a 4 semanas, nos quais uma equipe auto-organizada trabalha para entregar incrementos de um produto.



## Extreme Programming (XP)

O XP promove a colaboração próxima entre desenvolvedores e clientes, enfatizando a comunicação contínua e a programação em pares. O XP também incorpora práticas como testes automatizados, integração contínua e refatoração de código para garantir a qualidade do software. Essa abordagem ágil valoriza a flexibilidade para se adaptar às mudanças de requisitos e enfatiza a entrega frequente de pequenos incrementos de funcionalidade.



GITHUB. J. GitHub, [S. l.], c2020. Disponível em: <https://github.com/join>. Acesso em: 23 jan. 2024.

SBROCCO, J. H. T. de C. **Metodologias ágeis:** engenharia de software sob medida. São Paulo: Érica, 2012.

SOMMERRVILLE, I. **Engenharia de software.** 10. ed. São Paulo, SP: Pearson, 2018.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional.** 9. ed. Porto Alegre: AMGH, 2021.

## Unidade 2

### Qualidade de Software

#### Aula 1

Introdução à qualidade de software

#### Introdução à qualidade de software



##### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Bem-vindo à aula que aprofundará seus conhecimentos sobre Qualidade de Software. Exploraremos conceitos-chave, adentrando na Garantia de Qualidade de Software (SQA) e investigando as métricas que delineiam o padrão de excelência. Estes insights são indispensáveis para aprimorar a entrega de produtos de software confiáveis. Esteja preparado para adquirir as habilidades necessárias e garantir qualidade em todas as fases de sua prática profissional.

#### Ponto de Partida

Caro estudante, certamente você já deve ter procurado um aplicativo para resolver determinado problema, ou comprado um serviço ou produto, que, quando foi entregue, estava muito aquém do

esperado. Pois bem, como usuário você deve ter percebido que sua concepção sobre a empresa, após a deceção, mudou radicalmente. Porém, nesse momento, seu papel não é mais somente o de usuário, mas o de profissional de TI, o qual deve utilizar as técnicas de desenvolvimento de software para evitar tais situações.

Uma das técnicas para esse fim está relacionada com a qualidade de software e com suas ferramentas. Essa área do conhecimento da engenharia de software se preocupa em utilizar métodos, processos e normas nas atividades de desenvolvimento de software, a fim de se agregar qualidade nos processos e produtos finais. Para isso, é necessário que se tenha conhecimentos acerca de qualidade de software, qualidade do produto e qualidade dos processos. Como você pode ter percebido, existem diversas competências necessárias para se garantir a qualidade de um desenvolvimento.

Agora que você já se convenceu que a qualidade de software é indispensável, vamos supor que uma empresa de consultoria em tecnologia está encarregada de desenvolver um sistema integrado de gestão para uma grande corporação. O projeto enfrenta desafios relacionados à complexidade, prazos apertados e expectativas elevadas do cliente. Diante desse cenário, a equipe decide implementar práticas robustas de Garantia de Qualidade de Software (SQA) para assegurar a entrega de um produto confiável e de alta qualidade.

Os seguintes desafios são identificados:

- Garantir a conformidade com os requisitos do cliente.
- Identificar e mitigar riscos relacionados ao desenvolvimento.
- Estabelecer padrões de qualidade ao longo do ciclo de vida do software.

Sabendo disso, estratégias e práticas de SQA podem ser implementadas por essa empresa, além de apresentar uma síntese do resultado.

## Vamos Começar!

## Conceitos de Qualidade de software

Até mesmo os programadores mais experientes concordam que alcançar um software de alta qualidade é um objetivo fundamental. Contudo, surge a dúvida sobre como definir essa qualidade. De maneira geral, a qualidade de software pode ser descrita como a aplicação eficaz de gestão de qualidade, visando criar um produto útil que ofereça valor mensurável tanto para os desenvolvedores quanto para os usuários finais.

Um produto de alta qualidade deve ser útil, atendendo às exigências explícitas e implícitas dos usuários, proporcionando confiabilidade. Isso gera benefícios para a empresa, reduzindo manutenção e suporte, permitindo mais inovação. A comunidade de usuários também se beneficia, agilizando processos de negócio. O resultado é maior receita, rentabilidade e disponibilidade de informações cruciais.

O termo "qualidade de software" é comumente definido como a combinação de conformidade funcional e desempenho conforme as expectativas do patrocinador. No entanto, de acordo com Sommerville (2018), a abordagem à qualidade no desenvolvimento de software deve ser mais abrangente, considerando-a em três níveis distintos:

1. **Organizacional:** este nível amplo concentra-se no estabelecimento de padrões de trabalho para o desenvolvimento de software, incorporando as melhores práticas para minimizar erros e falhas.
2. **Projeto:** envolve o desenvolvimento baseado em padrões estabelecidos por gestores de projetos, variando de acordo com as peculiaridades do projeto, a política da empresa e o uso de frameworks, entre outros fatores.
3. **Planejamento:** exige a elaboração de um plano de qualidade, com uma equipe designada para verificar os requisitos acordados. Tanto os processos quanto os produtos desenvolvidos devem passar por revisões para evitar falhas inadvertidas.

Para esclarecer aspectos que impactam diretamente na qualidade, é fundamental observar de que maneira esses elementos se manifestam.

## Falhas de Software

Uma falha de software pode manifestar-se como um comportamento inesperado do sistema, resultante de um ou mais erros (Zanin *et al.*, 2018). Para uma compreensão mais aprofundada sobre como essa falha pode impactar um sistema, recomenda-se observar a Figura 1.

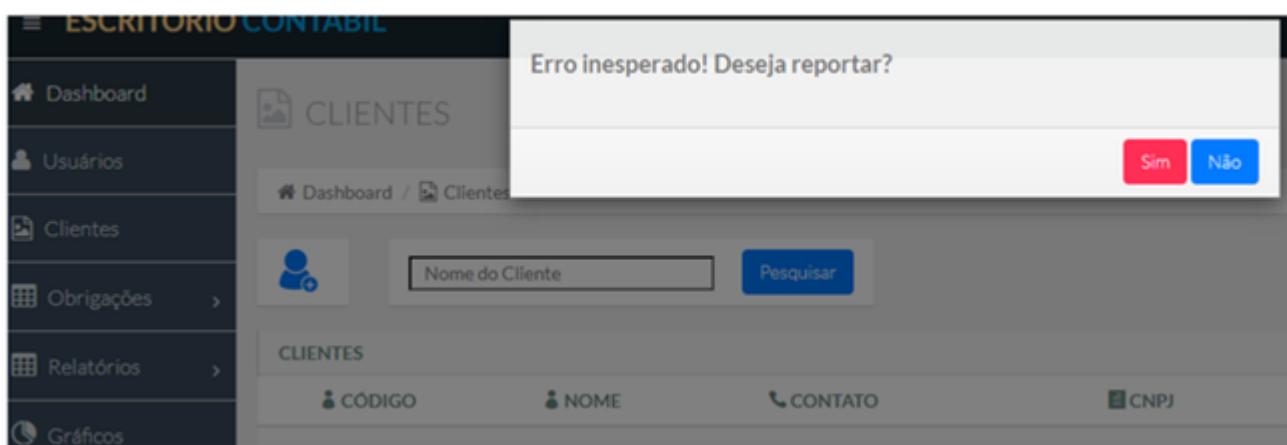


Figura 1 | Falha de software. Fonte: Maitino (2021).

Neste exemplo, o sistema gera uma mensagem de erro, impedindo o cadastro de um novo cliente, já que a confirmação do aceite é essencial para concluir o processo. Esse impacto é significativo para o usuário, especialmente se o sistema pertencer a uma loja de departamentos que depende dele para realizar vendas e emitir notas fiscais. O prejuízo resultante seria considerável e preocupante.

É importante destacar que cabe ao desenvolvedor a responsabilidade pela implementação das rotinas destinadas a lidar com falhas e erros. Essas situações podem ser comunicadas ao usuário por meio de mensagens, redirecionamento para uma nova página com imagens indicativas de falhas, ou ainda, utilizando um sistema de relatório para encaminhar os problemas a um repositório, visando ajustes por parte dos desenvolvedores.

## Erros de Software

Grande parte dos erros de software estão relacionados a execuções incorretas, o que faz com que os resultados gerados não reflitam a verdade (Zanin *et al.*, 2018). Um exemplo pode ser observado na Figura 2.

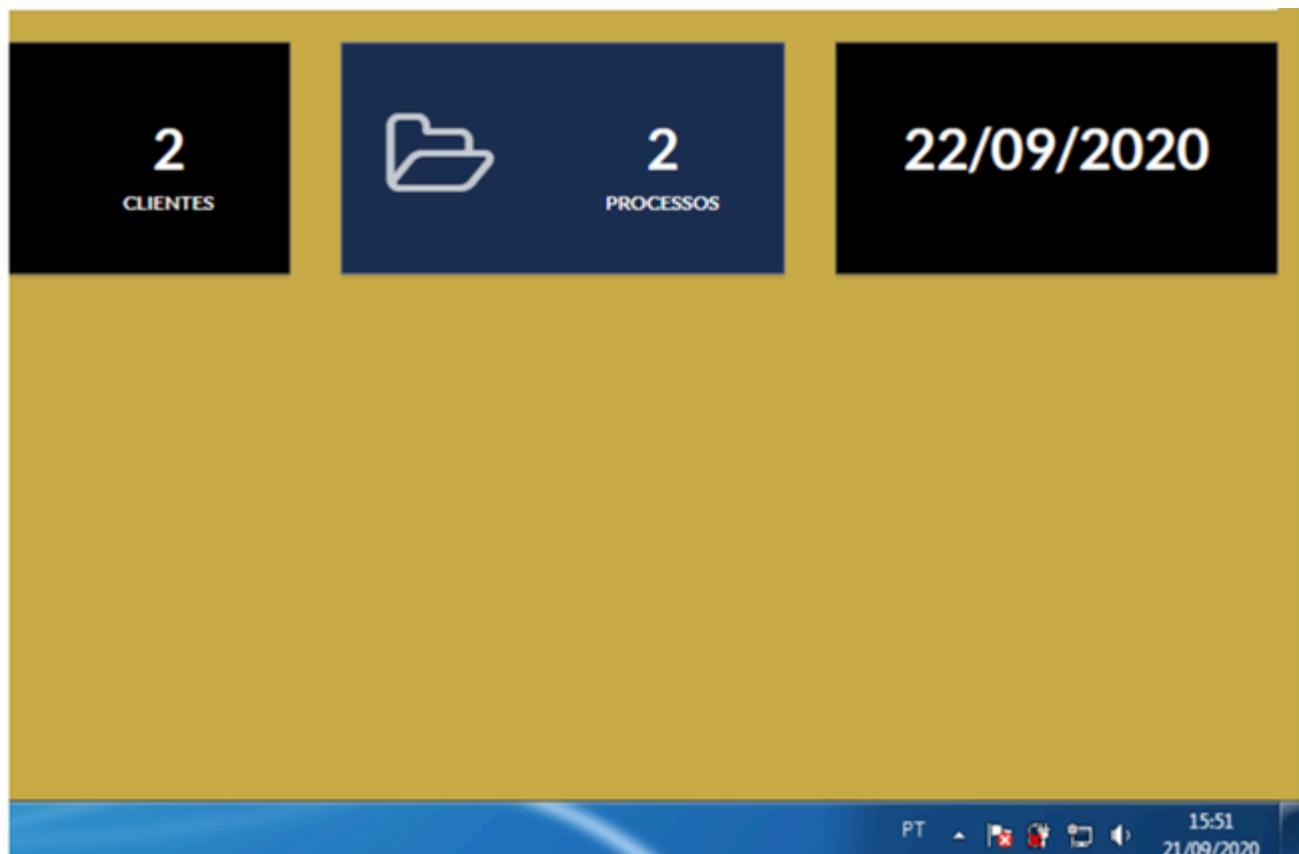


Figura 2 | Erro de software. Fonte: Maitino (2021).

Neste exemplo, atente-se para a comparação entre a data do sistema e a data indicada no relógio. Um erro na data do sistema pode acarretar um impacto significativo na integridade dos dados, resultando em registros que não são confiáveis quanto à sua data. Em setores como vendas, logística, contabilidade e fiscal, tal situação pode ser desastrosa.

## Defeito de Software

O defeito de software refere-se a uma implementação incorreta, resultando em erro, interrupção de serviço ou mau funcionamento. Um exemplo seria um sistema de cadastro que, mesmo após o preenchimento dos campos obrigatórios sem retorno de erros, falha em inserir as informações no banco de dados (Zanin *et al.*, 2018).

Quanto às atividades de desenvolvimento, localizar defeitos é mais desafiador, pois em alguns casos, erros de lógica, falhas na escrita do código ou outras referências não fornecem pistas claras para ajudar o desenvolvedor a identificar o problema.

## Bugs

Esse termo tornou-se comum entre os jovens para descrever comportamentos inesperados de softwares. Conforme Zanin *et al.* (2018), um bug de sistema refere-se a erros e falhas inesperados, geralmente mais complexos, exigindo maior tempo e conhecimento técnico para identificação e resolução.

Um problema que causou preocupação significativa entre os administradores de sistemas foi o bug do milênio. Em 1999, havia apreensão sobre o impacto nos sistemas durante a transição de ano, pois os sistemas mais antigos interpretavam apenas os dois últimos dígitos do ano. Isso poderia resultar em uma leitura errônea, levando o sistema, em vez de para o ano 2000, a retroceder para o ano 1900. Daí o nome "Bug do Milênio".

Agora que você entendeu a conexão entre a qualidade de software e os processos de desenvolvimento computacional, certamente surgirá a clássica dúvida: como podemos assegurar a qualidade de um software? Para compreender os aspectos relacionados à garantia da qualidade dos softwares, é importante, neste momento, ampliar a visão além das funcionalidades, desempenho, escalabilidade, entre outros. Isso implica considerar a qualidade dos processos envolvidos no desenvolvimento, teste e liberação para utilização.

## Siga em Frente...

## Garantia de qualidade de software

A garantia da qualidade conhecida como *Software Quality Assurance* (SQA). Sua abrangência permeia todo o ciclo de vida do projeto de desenvolvimento de software e requer (Sommerville, 2018):

- A utilização de ferramentas e/ou métodos que viabilizem a análise dos desenvolvimentos e dos testes.
- A condução de revisões técnicas nos componentes e na funcionalidade, sendo realizadas em cada uma das fases do ciclo.
- O controle documental por meio de procedimentos de versionamento.

- A aplicação de métodos para assegurar padrões de desenvolvimento e boas práticas que atendam às exigências das equipes de desenvolvimento.
- A implementação de mecanismos de aferição.

Conforme destacado por Pressman (2021), a garantia da qualidade refere-se aos procedimentos, aos métodos e às ferramentas empregados por profissionais de Tecnologia da Informação para garantir padrões previamente estabelecidos entre as partes ao longo de todo o ciclo de vida do desenvolvimento de um software. É saliente que os padrões de qualidade podem variar conforme as características específicas de cada projeto, e, portanto, a garantia da qualidade deve pautar-se pelos acordos estabelecidos entre as partes envolvidas.

A Figura 3 apresenta os seguintes elementos da SQA: (1) um processo dedicado à SQA; (2) tarefas específicas voltadas para garantia e controle da qualidade, incluindo revisões técnicas e uma estratégia de testes multicamadas; (3) a implementação eficaz de práticas de engenharia de software, compreendendo métodos e ferramentas; (4) o controle abrangente de todos os artefatos de software e das alterações realizadas nesses produtos; (5) um procedimento destinado a assegurar a conformidade com os padrões de desenvolvimento de software, quando aplicáveis; e (6) mecanismos de medição e geração de relatórios.

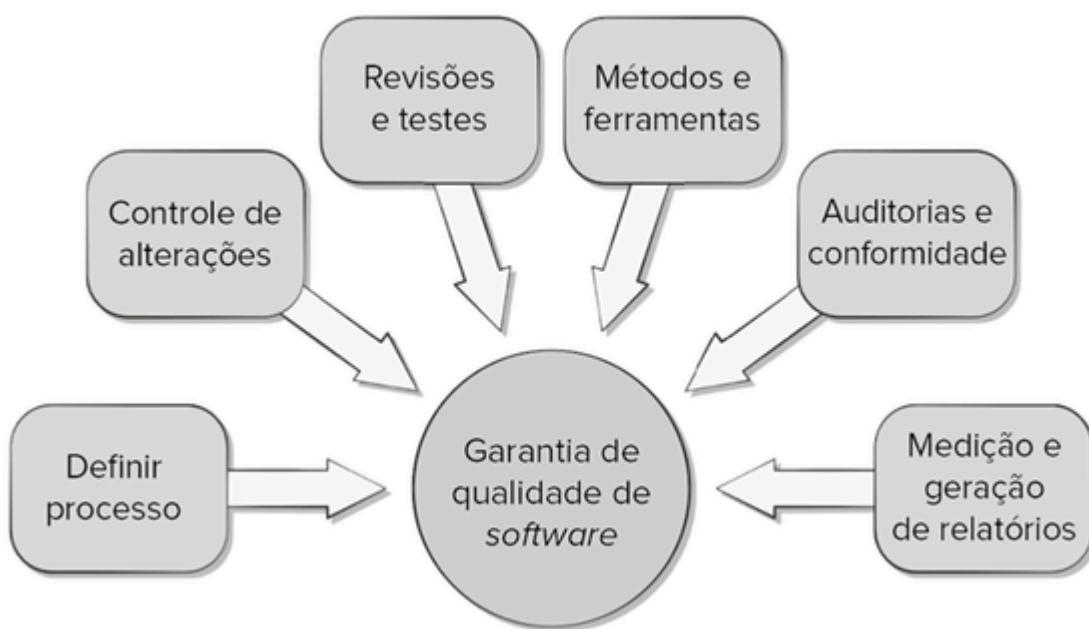


Figura 3 | Garantia de qualidade de software. Fonte: Pressman (2021).

A garantia de qualidade de software abrange diversas preocupações e atividades relacionadas à gestão da qualidade de software. As principais áreas incluem (Pressman, 2021):

- **Padrões:** garantir que os padrões estabelecidos por organizações como o IEEE e a ISO sejam seguidos na engenharia de software, seja por escolha da organização ou da imposição de clientes.

- **Revisões e Auditorias:** realizar revisões técnicas para identificar erros e auditorias para assegurar a conformidade com diretrizes de qualidade no trabalho de engenharia de software.
- **Testes de Software:** garantir o planejamento e a execução eficiente de testes para encontrar erros e alcançar os objetivos de qualidade.
- **Coleta e Análise de Erros/Defeitos:** medir o desempenho mediante coleta e análise de dados de erros para entender como são introduzidos e quais atividades são mais eficazes para sua eliminação.
- **Gerenciamento de Mudanças:** assegurar que práticas adequadas de gerenciamento de mudanças sejam implementadas para evitar confusões e garantir qualidade.
- **Educação:** liderar o aperfeiçoamento do software por meio da educação contínua de engenheiros de software, gerentes e outros envolvidos.
- **Gerência dos Fornecedores:** garantir a qualidade do software adquirido de fornecedores externos, sugerindo práticas de garantia de qualidade e incorporando requisitos de qualidade em contratos.
- **Administração da Segurança:** implementar políticas de segurança para proteger dados em todos os níveis e garantir a segurança do software contra alterações não autorizadas.
- **Proteção:** avaliar o impacto de falhas de software, especialmente em sistemas críticos, e iniciar medidas para redução de riscos.
- **Gestão de Riscos:** assegurar que as atividades de gestão de riscos sejam conduzidas adequadamente e que planos de contingência relacionados a riscos sejam estabelecidos.

Além disso, a SQA trabalha para garantir que atividades de suporte ao software, como manutenção, suporte on-line, documentação e manuais, sejam realizadas com qualidade como preocupação principal.

## Métricas de qualidade

A medição de software envolve derivar valores numéricos ou perfis para atributos de componentes de software, sistemas ou processos. Esses valores são comparados entre si e com padrões para avaliar a eficácia de métodos, ferramentas e processos de software. Por exemplo, ao introduzir uma nova ferramenta de teste, é possível registrar o número de defeitos antes e depois do uso da ferramenta para avaliar sua eficácia.

O objetivo a longo prazo da medição de software é substituir revisões para julgar a qualidade do software. Idealmente, métricas de software podem ser usadas para avaliar sistemas com base em diversas métricas, deduzindo um valor para a qualidade. No entanto, apesar dessa aspiração, avaliações automatizadas de qualidade ainda não estão próximas de se tornarem realidade (Somerville, 2018).

Métricas de software são características mensuráveis de um sistema, uma documentação ou um processo de desenvolvimento. Podem ser métricas de controle, relacionadas aos processos de gerenciamento, ou às métricas de previsão, associadas às características do software. Exemplos incluem o tamanho do código, índice Fog (medida de legibilidade), número de defeitos relatados e complexidade ciclomática de um módulo.

As métricas de controle e previsão desempenham um papel crucial na tomada de decisões de gerenciamento, conforme ilustrado na Figura 4. As métricas de processo guiam os gerentes ao determinar se alterações no processo são necessárias, enquanto as métricas de previsão ajudam a estimar o esforço necessário para implementar mudanças no software (Sommerville, 2018).

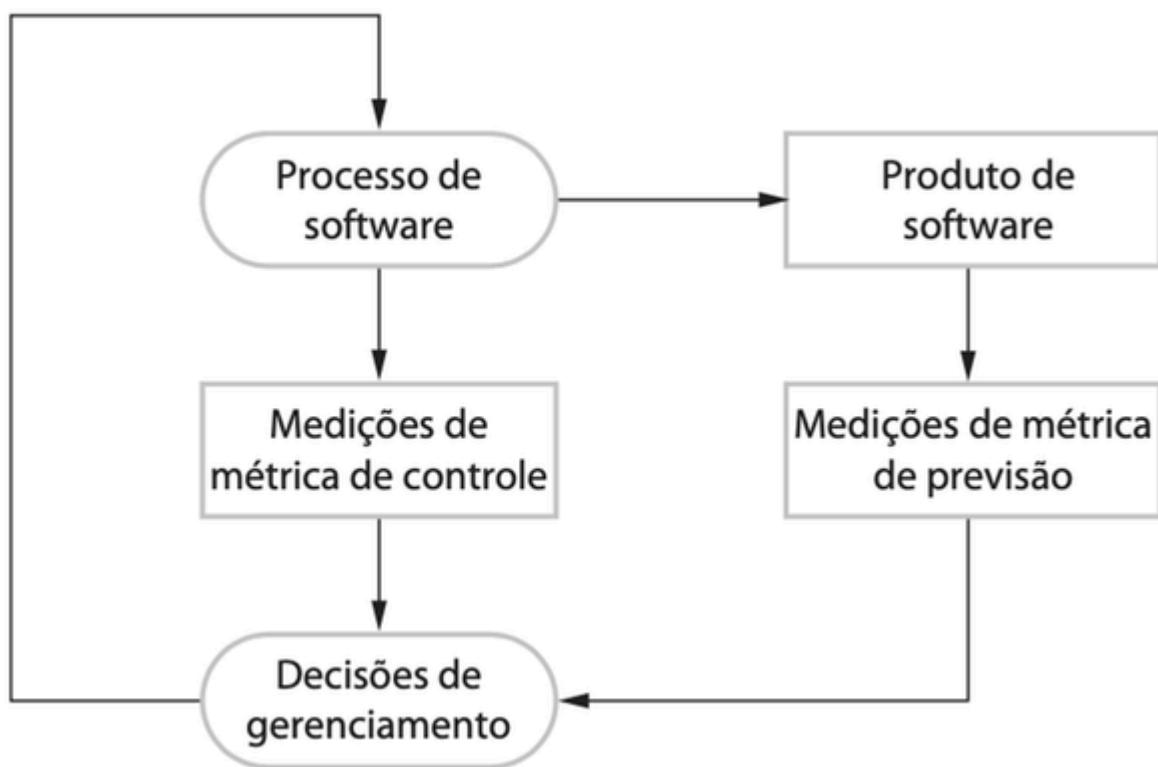


Figura 4 | Medições de previsão e de controle. Fonte: Sommerville (2018).

Há duas abordagens para o uso de medições em um software de sistema (Sommerville, 2018):

- Atribuir valores aos atributos de qualidade do sistema, avaliando características individuais dos componentes do sistema, como sua complexidade ciclomática, e agregando essas medições para avaliar atributos de qualidade do sistema, como manutenibilidade.
- Identificar os componentes do sistema que não atendem aos padrões de qualidade. As medições podem destacar componentes individuais com características que se desviam da norma. Por exemplo, medir a complexidade de componentes pode revelar aqueles com maior propensão a bugs devido à dificuldade de compreensão associada à alta complexidade.

## Vamos Exercitar?

A Garantia da Qualidade de Software (SQA) desempenha um papel essencial na engenharia de software, sendo crucial para assegurar a entrega de um produto confiável e de alta qualidade. Ao

estabelecer padrões, diretrizes e práticas robustas ao longo do ciclo de vida do desenvolvimento de software, a SQA visa garantir a conformidade com os requisitos do cliente, identificar e mitigar riscos, e estabelecer uma base sólida para a produção de software de excelência. A implementação efetiva de práticas de SQA resulta em benefícios tangíveis, incluindo a redução de defeitos, a otimização do desempenho, e a melhoria da satisfação do cliente. Além disso, ao criar uma cultura de avaliação contínua, treinamento e inovação, a SQA contribui para o desenvolvimento de produtos que atendem não apenas às expectativas, mas também estabelecem padrões elevados de qualidade na indústria de software. Sabendo disso, uma possível solução para a problematização proposta, está descrita a seguir:

## Estratégias e Práticas de SQA Implementadas

### 1. Definição de Padrões e Diretrizes:

- Estabelecimento de normas para codificação, documentação e testes.
- Utilização de padrões reconhecidos, como ISO/IEC 25010, para orientar o desenvolvimento.

### 2. Revisões e Auditorias:

- Realização de revisões de código e auditorias de processos regularmente.
- Envolvimento de uma equipe independente para garantir imparcialidade.

### 3. Testes Abrangentes:

- Desenvolvimento de planos de teste abrangentes, incluindo testes unitários, de integração e de sistema.
- Automação de testes sempre que possível para garantir consistência e eficiência.

### 4. Gestão de Mudanças:

- Implementação de um sistema eficaz de controle de mudanças.
- Avaliação de impacto e revisões sistemáticas antes da aprovação de alterações.

### 5. Treinamento Contínuo:

- Oferta de treinamentos regulares para a equipe de desenvolvimento.
- Fomento de uma cultura de aprendizado contínuo e melhoria.

## Resultados

A implementação efetiva das práticas de SQA resultou em diversos benefícios para o projeto. A conformidade com os requisitos do cliente foi consistentemente atingida, evitando retrabalho e garantindo a satisfação do cliente. A identificação precoce de riscos permitiu ações preventivas, minimizando impactos negativos. A introdução de padrões elevou a qualidade geral do código e da documentação. A gestão de mudanças eficiente assegurou a estabilidade do sistema, mesmo

diante de atualizações frequentes. Além disso, a cultura de treinamento contínuo fortaleceu as habilidades da equipe, contribuindo para o sucesso do projeto e consolidando a importância da SQA no desenvolvimento de software.

## Saiba mais

Para ler mais sobre a Qualidade de Software, acesse o livro de Pressman, [Engenharia de Software](#), Capítulo 15.

Quer saber mais sobre a qualidade de Software? Leia o Capítulo 24 de [Engenharia de Software - Sommerville](#).

Leia mais sobre SQA acessando o artigo indicado [Garantia da Qualidade de Software \(SQA\)](#).

## Referências

MAITINO NETO, R. **Engenharia de software**. Londrina: Editora e Distribuidora Educacional S.A., 2021.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

ZANIN, A. *et al.* **Qualidade de Software**. Porto Alegre: Sagah, 2018.

## Aula 2

Qualidade de produto

### Qualidade de produto

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá a Gestão da Qualidade do Produto. Exploraremos desde os fundamentos até as Métricas de Produto e Normas ISO de Qualidade. Esses temas são fundamentais para sua prática profissional, capacitando-o a assegurar padrões elevados, medir a eficácia do produto e alinhar-se às normativas internacionais. Prepare-se para adquirir conhecimentos essenciais que impulsionarão a excelência na gestão da qualidade em seus projetos profissionais.

## Ponto de Partida

Caro estudante, a Engenharia de Software é uma área na qual são discutidos assuntos como gerenciamento, qualidade, processos, entre outros. Dentro da qualidade existem discussões com especificidades, como é o caso da qualidade do produto. Com isso, você deve imaginar que essas discussões tendem à área de desenvolvimento de software.

Onde nós podemos ver a qualidade do produto de software no dia a dia? Diversos sistemas, que utilizamos nos computadores, nos smartphones, nos sistemas embarcados, etc., foram validados em todas as suas características. Tendo isso em vista, as normas de qualidade de produto são um guia para que os processos, as avaliações e as métricas, sejam ajustadas conforme as necessidades do projeto.

Inicialmente, serão discutidos os modelos de qualidade de produto de software quanto a suas características, necessidades e aplicações, o que é de grande importância para que você possa, posteriormente, compreender as estruturas das normas.

Com isso, o próximo passo é compreender como as métricas são utilizadas para aferir a qualidade do produto de software, sendo possível que você entenda como encontrar os pontos que necessitam de avaliações e quais medidas podem ser utilizadas para tal finalidade.

Em seguida, serão discutidas as normas ISO/IEC (NBR), suas características e subcaracterísticas. E, embora toda essa estrutura seja complexa, você terá exemplos de momentos em que são aplicadas as técnicas durante o ciclo de vida do projeto.

Sobre esses assuntos, vamos imaginar que uma empresa de desenvolvimento de games para mobile adotou as normas da ISO 9126 para ajustes de qualidade de produto. Isso fez com que, após alguns projetos, os resultados pudessem ser sentidos por clientes, gestores de projetos e desenvolvedores. Porém, as atividades relacionadas à ISO 9126 têm se tornado uma prática diferente para cada gerente de projetos, causando desconfortos aos desenvolvedores quando são alocados para outros times de desenvolvimento.

Como você é o gerente de projetos com mais tempo de empresa, o gerente geral solicitou um relatório, no qual você deve propor uma solução viável e que, de preferência, não gere custos. Assim, espera-se que haja uma gestão das atividades relacionadas à qualidade do produto durante os projetos de desenvolvimento de software.

Bons estudos!

## Vamos Começar!

### Gestão da qualidade do produto

Prezado estudante, é provável que você já tenha experimentado a frustração de baixar um aplicativo que, apesar de prometer funcionalidades específicas, revelou-se complexo demais em sua operação, resultando em um desempenho distante das expectativas inicialmente anunciadas. Este descompasso entre a promessa na descrição da loja de aplicativos e a efetiva qualidade do produto final é abordado na Engenharia de Software sob o tema da qualidade de produto de software. Embora possa parecer peculiar considerar o software como um produto tangível, as discussões sobre a qualidade de produtos abrangem uma ampla gama de tópicos, desde modelos e normas de qualidade até gestão, medições, requisitos e avaliações. Dada a complexidade inerente a esses conceitos, exploraremos exemplos concretos para uma compreensão mais aprofundada.

Conforme Mello (2011), ao contrário dos produtos manufaturados, o desenvolvimento de softwares é um processo projetado, demandando habilidades criativas e técnicas dos profissionais envolvidos. Enquanto a produção de bens manufaturados frequentemente se baseia em processos mecanizados, seguindo a automação previamente concebida pelos engenheiros, o desenvolvimento de software é caracterizado por sua natureza criativa e técnica.

Mello (2011) ilustra essa diferença na prática ao abordar os setores de manufatura e desenvolvimento. No setor de manufatura, a produção envolve matérias-primas, processos de transformação, embalagem, acondicionamento e transporte, com diferentes níveis de complexidade no sistema produtivo. Por exemplo, uma padaria especializada em bolos utiliza matérias-primas específicas em seus processos, produzindo bolos com diversos graus de complexidade. Já uma empresa farmacêutica constrói as características de seu setor produtivo com base em insumos variados e diferentes etapas de processamento.

No setor de desenvolvimento, a maioria dos projetos demanda recursos computacionais, como computadores, acesso à internet e aos programas. No entanto, o sucesso do projeto depende significativamente das habilidades, das competências e da criatividade dos profissionais que utilizam o desenvolvimento de software para encontrar soluções em diversas áreas do conhecimento.

A complexidade em assegurar a qualidade do produto de software é evidente. Para enfrentar esse desafio, existem ferramentas conhecidas como modelos de qualidade de produto. Normas

internacionais foram estabelecidas para garantir a aderência a premissas específicas, independentemente do local de desenvolvimento do software.

## Métricas de Produto

As métricas de produto são medidas de previsão utilizadas para avaliar atributos internos de um sistema de software, como o tamanho do sistema em linhas de código ou o número de métodos associados a cada classe de objeto. No entanto, características facilmente mensuráveis, como tamanho e complexidade ciclomática, não apresentam uma relação clara e consistente com atributos de qualidade, como capacidade de compreensão e manutenibilidade. Essas relações variam conforme os processos de desenvolvimento, tecnologias utilizadas e o tipo de sistema em desenvolvimento.

As métricas de produto podem ser categorizadas em duas classes distintas (Sommerville, 2018):

- **Métricas Dinâmicas:** coletadas por meio de medições realizadas enquanto um programa está em execução. Essas métricas podem ser obtidas durante testes de sistema ou após a implementação do sistema. Exemplos incluem o número de relatórios de bugs e o tempo necessário para concluir uma computação.
- **Métricas Estáticas:** coletadas por meio de medições feitas em representações estáticas do sistema, como o projeto, o código-fonte ou a documentação. Exemplos de métricas estáticas incluem o tamanho do código e o comprimento médio de identificadores utilizados.

Esses tipos de métricas estão associados a diferentes atributos de qualidade. Métricas dinâmicas auxiliam na avaliação da eficiência e confiabilidade de um programa, enquanto métricas estáticas ajudam a avaliar a complexidade, compreensibilidade e manutenibilidade de um sistema ou de seus componentes.

Normalmente, há uma relação evidente entre as métricas dinâmicas e as características de qualidade do software. Medir o tempo de execução de funções específicas e avaliar o tempo necessário para iniciar um sistema é relativamente simples, e essas métricas estão diretamente associadas à eficiência do sistema. Da mesma forma, registrar o número de falhas do sistema e o tipo de falha permite uma correlação direta com a confiabilidade do software.

Métricas estáticas, exemplificadas na Tabela 1, mantêm um relacionamento indireto com atributos de qualidade. Diversas métricas foram propostas, e muitos experimentos foram conduzidos na tentativa de estabelecer e validar conexões entre essas métricas e atributos como complexidade do sistema e manutenibilidade. Embora nenhum desses experimentos tenha fornecido conclusões definitivas, observa-se que o tamanho do programa e a complexidade de controle parecem ser os indicadores mais confiáveis para prever a compreensibilidade, complexidade e manutenibilidade de um sistema.

Métrica de software	Descrição
---------------------	-----------

<i>Fan-in/Fan-out</i>	<p><i>Fan-in</i> é a medida do número de funções ou métodos que chamam outra função ou outro método (digamos X). <i>Fan-out</i> é o número de funções que são chamadas pela função função de X. Um valor alto para <i>fan-in</i> significa que X está fortemente acoplado ao resto do projeto e as alterações em X terão repercussões extensas. Um valor alto para <i>fan-out</i> sugere que a complexidade geral do X pode ser alta por causa da complexidade da lógica de controle necessário para coordenar os componentes chamados.</p>
Comprimento de código	<p>Esta é uma medida do tamanho de um programa. Geralmente, quanto maior o tamanho do código de um componente, mais complexo e sujeito a erros o componente é. O comprimento em código tem mostrado ser uma das métricas mais confiáveis para prever a propensão a erros em componentes.</p>
Complexidade ciclomática	<p>Está é uma medida de complexidade de controle de um programa. Essa complexidade de controle pode estar relacionada à comprehensibilidade de programa.</p>
Comprimento de identificadores	<p>Esta é uma medida do comprimento médio dos identificadores (nomes de variáveis, classes, métodos etc) em um programa. Quanto mais longos os identificadores, mais provável que sejam significativos e, portanto, mais comprehensível o programa.</p>
Profundidade de aninhamento condicional	<p>Esta é uma medida da profundidade de aninhamento de</p>

	declarações <i>if</i> em um programa. Declarações <i>if</i> profundamente aninhadas são difíceis de entender e potencialmente sujeitas a erros.
Índice <i>Fog</i>	Esta é uma medida do comprimento médio de palavras e sentenças em documentos. Quanto maior o valor de um índice <i>Fog</i> de um documento, mais difícil a sua compreensão.

Tabela 1 | Métricas estáticas de produto de software. Fonte: Sommerville (2018).

Na Figura 1, é apresentado um processo de medição que pode integrar-se a uma avaliação de qualidade de software. Cada componente do sistema pode ser analisado individualmente, utilizando diversas métricas. Os valores dessas métricas podem ser comparados entre diferentes componentes e, possivelmente, com dados históricos de medição obtidos em projetos anteriores. Medições anômalas, que se desviam significativamente da norma, podem indicar problemas na qualidade desses componentes.

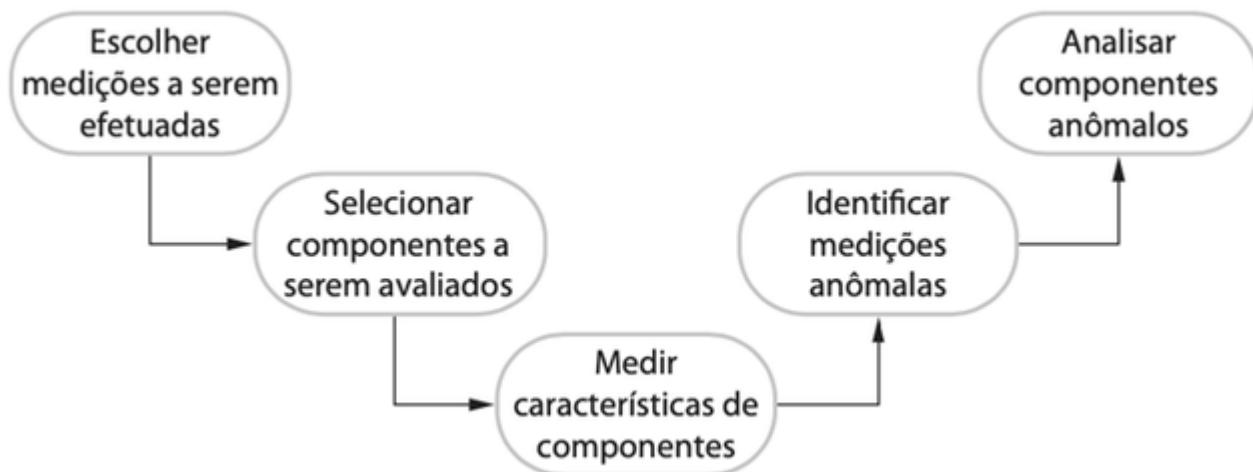


Figura 1 | O processo de medição de produto. Fonte: Sommerville (2018).

Neste processo de medição de componentes, os estágios principais incluem (Sommerville, 2018):

- **Seleção de Medições:** formular questões específicas que as medições devem responder e definir as métricas necessárias. Utilizar o paradigma Meta-Questão-Métrica (GQM) para decidir quais dados coletar.
- **Seleção de Componentes:** escolher os componentes a serem avaliados, podendo focar em um conjunto representativo ou nos componentes principais do sistema.

- **Medição de Características:** medir os componentes selecionados, envolvendo o processamento da representação do componente usando ferramentas automatizadas de coleta de dados.
- **Identificação de Medições Anômalas:** comparar os valores das métricas entre os componentes e com medições anteriores para identificar valores anormalmente altos ou baixos, indicando possíveis problemas de qualidade.
- **Análise de Componentes Anômalos:** examinar os componentes com valores anômalos para determinar se isso compromete a qualidade do componente. Valores anômalos não necessariamente indicam má qualidade, podendo haver outras razões.

Manter os dados coletados como recurso organizacional e registros históricos é essencial. Ao estabelecer um banco de dados de medições abrangente, é possível comparar a qualidade do software entre projetos e validar as relações entre atributos internos dos componentes e das características de qualidade.

## Siga em Frente...

## Normas ISO de qualidade

A *International Organization for Standardization* (ISO) é uma entidade internacional que desenvolve e publica normas reconhecidas mundialmente em diversas áreas, incluindo a gestão de qualidade.

### ISO 9126 (NBR 13596)

A ISO 9126 (NBR 13596) é uma norma destinada à avaliação da qualidade, das características e dos atributos de software. Seu propósito também inclui a padronização das atividades e dos métodos de avaliação da qualidade do produto, proporcionando feedbacks valiosos para equipes de desenvolvimento de software. A estrutura da norma é composta por quatro partes, cada uma abordando aspectos específicos (Wazlawick, 2013):

- ISO 9126-1 de 2001: trata das características, subcaracterísticas e métricas da qualidade do produto de software (foco desta seção).
- ISO 9126-2 de 2003: aborda métricas externas e o controle de falhas.
- ISO 9126-3 de 2003: tem como objetivo verificar a quantidade de ocorrências de falhas e estimar o tempo de recuperação.
- ISO 9126-4 de 2004: lida com User Experience, produtividade, eficácia e segurança.

Segundo Wazlawick (2013), a ISO 9126 é considerada parte integrante da família de normas de qualidade 9000, com ênfase na qualidade do produto de software. No contexto brasileiro, a NBR 13596 é equivalente à ISO, no entanto, a NBR foi substituída pela ISO/IEC 9126-1. A norma presente na ISO 9126 é estruturada em características, subcaracterísticas e métricas, sendo que as seis características podem ser visualizadas na Figura 2.



Figura 2 | Características da ISO 9126. Fonte: adaptada de Wazlawick (2013).

Para melhor compreensão das características e subcaracterísticas da ISO 9126, a seguir vamos ver o detalhamento de cada uma delas segundo Wazlawick (2013).

### Funcionalidade

Descreve os atributos das funções incorporadas nos softwares, abordando diversas subcaracterísticas:

- **Adequação:** refere-se à concordância do funcionamento adequado da funcionalidade.
- **Acurácia:** concentra-se nas saídas geradas pelo software, as quais devem estar em conformidade com as necessidades.

- **Interoperabilidade:** indica a capacidade do software de ser utilizado por diferentes tecnologias.
- **Conformidade:** deve estar em conformidade com normas, regras e leis específicas.
- **Segurança de acesso:** envolve a capacidade de prevenir intrusões e incidentes relacionados à segurança da informação.

Para ilustrar a identificação de funcionalidades em um projeto, considere a seguinte situação: uma empresa necessita de um chat para facilitar a comunicação entre colaboradores na rede local, com a exigência de utilizar o protocolo IP em ambas as versões, IPv4 e IPv6. Como evidenciado, a funcionalidade de comunicação interna apresenta características de grande importância em relação à adequação e interoperabilidade.

## Confiabilidade

Refere-se à habilidade do software em permanecer operacional e atender ao desempenho esperado ou estabelecido. Destacam-se as seguintes subcaracterísticas:

- **Maturidade:** demonstra a frequência das ocorrências de falhas no software.
- **Tolerância a falhas:** envolve a verificação, em situações de falhas ou incidentes relacionados à segurança (sejam provocados ou decorrentes de violações), dos serviços que permanecerão acessíveis e com a garantia da integridade.
- **Recuperabilidade:** diz respeito ao tempo necessário para que, após a ocorrência de uma falha específica, a funcionalidade ou sistema esteja novamente disponível.

Os aspectos relacionados à confiabilidade são mais evidentes em nossas atividades diárias, tornando-se mais fácil identificá-los. Por exemplo, ao projetar aplicativos de streaming para smart TVs, foi essencial compreender a confiabilidade, levantando questões como: em caso de falha, o filme ou série será interrompido? Em caso de falhas, quais são? Qual o tempo estimado para que o sistema se restabeleça?

## Usabilidade

Refere-se a uma medida para avaliar o nível de experiência do usuário (*User Experience*) em relação à facilidade de operação do software. As subcaracterísticas podem ser delineadas da seguinte forma:

- **Inteligibilidade:** relaciona-se à lógica da aplicabilidade, sendo intuitiva e de fácil operação.
- **Apreensibilidade:** avalia o esforço que o usuário emprega para aprender a utilizar uma funcionalidade ou o software como um todo.
- **Atratividade:** observa o grau em que as interfaces do software capturam a atenção do usuário.

A usabilidade, em termos de experiência do usuário, é uma das características que demanda especial atenção. É comum que alguns usuários instalem softwares de edição de vídeo e, devido à complexidade para realizar determinadas operações, desistam e busquem soluções mais

acessíveis em smartphones. Isso está diretamente relacionado às características de usabilidade do software.

## Eficiência

Nesta característica, a atenção está voltada para a capacidade de modificação, que pode ocorrer em diferentes situações, tais como a exclusão de defeitos e falhas, a adição de novas funcionalidades, a adaptação a novas plataformas ou sistemas operacionais, entre outras possibilidades. As subcaracterísticas são delineadas da seguinte maneira:

- **Modificabilidade:** aborda a capacidade do software de ser modificado por razões ou necessidades específicas.
- **Estabilidade:** preocupa-se em verificar se ocorrerão falhas após as modificações.
- **Escalabilidade:** avalia a capacidade de crescimento do software para atender a uma demanda maior.

A manutenção de softwares é uma preocupação constante para gerentes de projetos, uma vez que falhas na abordagem dessas subcaracterísticas têm um impacto direto na qualidade dos serviços prestados.

## Portabilidade

Em 2020, o governo federal implementou auxílio financeiro à população por meio de um benefício, em resposta à elevada taxa de desemprego causada pela pandemia do novo coronavírus. Contudo, para ter acesso a esse benefício, era necessário instalar um aplicativo no smartphone. O problema surgiu quando ficou evidente que o aplicativo não havia sido projetado para atender a uma parcela tão extensa da população, comprometendo significativamente o seu funcionamento. A portabilidade torna-se um ponto de grande relevância com o advento dos dispositivos móveis, e suas características podem ser delineadas da seguinte forma (Wazlawick, 2013):

A portabilidade abrange um conjunto de características que indicam a capacidade do software de ser utilizado em outros sistemas, dispositivos e plataformas.

- **Adaptabilidade:** refere-se à capacidade de se adaptar a ambientes para os quais não foi originalmente projetado.
- **Analisabilidade:** envolvem a análise dos impactos positivos e negativos que a utilização do software em outros contextos pode acarretar.
- **Interoperabilidade:** demonstra a capacidade de interagir com outros sistemas, frequentemente desenvolvidos com diferentes tecnologias e arquiteturas.

Um exemplo emblemático e atual da importância da portabilidade surgiu com a popularização dos smartphones. Os sites, inicialmente projetados para monitores de tamanhos consideráveis, apresentaram comportamento inadequado nesses dispositivos, evidenciando a falta de preparo para a portabilidade.

## ISO 9000

Certamente, observou-se que os métodos abordados neste contexto são todos respaldados por normas. Muitos sistemas de gestão de qualidade têm como base as normas ISO 9000:2015 (ABNT, 2015), as quais têm como propósito:

- Descrever os fundamentos e princípios da gestão da qualidade.
- Compreender os processos de implementação da gestão da qualidade.
- Avaliar a conformidade dos produtos de software desenvolvidos.

De acordo com Carpinetti e Gerolamo (2019), a ISO 9000 incorpora oito princípios de gestão da qualidade, os quais visam orientar os gestores na melhoria de desempenho, especialmente em atividades relacionadas ao desenvolvimento de software. Esses princípios são os seguintes:

- **Foco no cliente:** adota uma abordagem que busca melhores práticas para entregar o melhor produto.
- **Liderança:** emprega metodologia e abordagens como meio de liderar.
- **Pessoas:** utiliza formas para incentivar o comprometimento das pessoas com os processos e a qualidade.
- **Processos:** verifica constantemente os processos e os repensa.
- **Inter-relacionamento:** promove o inter-relacionamento de atividades concorrentes.
- **Melhoria:** busca a melhoria contínua por meio de metodologias, normas e boas práticas.
- **Decisão:** utiliza os feedbacks gerados a favor da tomada de decisão.
- **Benefícios:** gera vantagens administrativas e operacionais por meio da adoção de boas práticas.

Para compreender como os princípios da ISO 9000 estão conectados às atividades de desenvolvimento no ambiente profissional, considere o exemplo a seguir. Imagine a existência de "ilhas" de desenvolvimento dentro de uma organização, onde as equipes são segregadas por conhecimento e habilidades. Contudo, em determinado momento do projeto, as funcionalidades desenvolvidas precisam ser integradas para que o sistema efetivamente ganhe existência. Como proceder?

A ISO 9000 oferece orientações específicas para abordar essa situação. Observe como a norma pode ser benéfica nesse exemplo, trazendo impactos positivos para os profissionais envolvidos. A norma fornece diretrizes sobre a abordagem a ser adotada com a equipe, liderança que orienta a integração das funcionalidades, e, sob uma perspectiva diferente, um olhar que mapeia os processos para propor melhorias e ajustes. Isso possibilita um planejamento mais adequado e gera benefícios a curto e longo prazo, conforme a maturidade aumenta.

## ISO 9001

Para concluir, é relevante destacar que a norma 9001:2015 adota uma abordagem com foco em tarefas administrativas no âmbito do Sistema de Gestão da Qualidade (SGQ). Conforme o catálogo da ISO 9001:2015 (ABNT, 2015), seus objetivos incluem:

- Gerenciar o controle documental.
- Realizar o controle de registros de qualidade.
- Padronizar a realização de auditorias internas.
- Efetuar o controle de produtos que não atendam às conformidades.
- Implementar ações corretivas.
- Implementar ações preventivas.

Ao longo das discussões apresentadas nesta unidade, você pôde observar como as normas se configuram como excelentes guias para as atividades de desenvolvimento de software, certo? Para alguns profissionais, a utilização dessas ferramentas pode parecer adicionar uma camada extra de atividade além daquelas já necessárias em um projeto. Entretanto, o propósito é, na realidade, o oposto. A intenção é que, por meio da adoção das normas, as atividades sejam realizadas de maneira mais precisa, resultando em uma menor necessidade de revisitar processos e atividades de desenvolvimento.

## Vamos Exercitar?

Ao mesmo tempo que a ISO 9126 ajudou essa empresa, ela tem se tornado um problema quando os colaboradores migram de uma equipe a outra, isso porque cada gerente de projetos tem utilizado uma forma de operacionalizar as normas ISO 9126. Para isso, o gerente geral solicitou que você desenvolvesse um relatório com um sistema de gestão de qualidade de produto.

Dessa forma, foi feita a seguinte sugestão: vamos adotar a ISO 9000, pois a sua estrutura possui oito princípios de gestão da qualidade, os quais têm como objetivo conduzir os gestores nas atividades de desenvolvimento de software. Perceba como nós conseguiremos a padronização de trabalho entre os gerentes de projetos ao observar os princípios dessa norma: foco no cliente, liderança, pessoas, processos, inter-relacionamento, melhoria, decisão e benefícios.

## Saiba mais

Acesse o artigo a seguir sobre a ISO 9126. [ISO 9126: O que é e por que é importante para a qualidade de software?](#)

Para ler mais sobre a Qualidade de Software, acesse o livro [Engenharia de Software - Pressman](#), Capítulo 15.

Quer saber mais sobre a qualidade de Software? Leia o Capítulo 24 de [Engenharia de Software - Sommerville](#).

## Referências

CARPINETTI, L. C. R.; GEROLAMO, M. C. **Gestão da Qualidade ISO 9000:2015**. São Paulo: Atlas, 2019.

MELLO, C. H. P. **Gestão da qualidade**. São Paulo: Pearson Education do Brasil, 2011.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

WAZLAWICK, R. S. **Engenharia de software**: conceitos e prática. Rio de Janeiro: Elsevier, 2013.

## Aula 3

Qualidade de processo

### Qualidade de processo

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você ampliará a sua compreensão sobre Qualidade do Processo, abordando os renomados modelos CMMI e MPS.BR. Explore desde os conceitos fundamentais até a aplicação prática dessas metodologias. Esses conteúdos são essenciais para aprimorar sua prática profissional, oferecendo ferramentas sólidas para avaliar, aperfeiçoar processos e atingir padrões internacionais de excelência. Esteja preparado para incorporar conhecimentos valiosos e elevar sua expertise na busca pela qualidade em seus projetos.

### Ponto de Partida

Caro estudante, muitas vezes nós, enquanto consumidores, estamos acostumados somente a ver o produto final e não os processos envolvidos no desenvolvimento dele. A ideia é a de que,

quando você vai a um fast-food, a única parte do processo com a qual você tenha contato seja o momento de fazer e receber o pedido. Seja em um fast-food, seja em um setor industrial ou, no nosso caso, na indústria do software, na grande maioria das vezes, avaliamos se o processo foi bem executado considerando o resultado que é entregue. Mas como são feitas as tratativas referentes à qualidade dos processos de desenvolvimento de software?

Inicialmente, serão discutidos os modelos de maturidade CMMI e MPS.BR, os quais são metodologias que visam, por meio de sua estrutura, posicionar a empresa em um nível de maturidade e, mediante o uso de técnicas, planejar uma evolução em termos organizacionais a fim de promover melhoria em seus processos.

Em seguida, haverá uma discussão acerca da ISO 9001:2015, cujo foco está em utilizar metodologias que permitam aos membros da organização conhecer os processos, os microprocessos e as interações entre os componentes em sua totalidade. Dessa forma, ao se conhecer bem os processos, a chance de erros e falhas é minimizada, proporcionando a melhoria dos processos e consequentes benefícios.

Em termos profissionais, esses temas são de extrema importância pois são largamente utilizados dentro de organizações que buscam técnicas para atingir resultados expressivos. Dessa maneira, conhecer as técnicas e compreender o momento de aplicá-las pode ser um interessante diferencial competitivo.

Para a nossa atividade, imagine que você trabalhe na PROC-TI, uma empresa que já atua no ramo de Tecnologia da Informação (TI) há mais de 25 anos. A PROC-TI foi contratada pelo governo federal para desenvolver uma solução de identificação via drones de áreas devastadas pelas queimadas na Amazônia.

Sabendo que a PROC-TI possui o interesse de implantar um modelo de referência em seus processos, a partir desta contratação do governo federal, explique qual seria o modelo mais adequado para a instituição e quais seriam as vantagens da implantação deste.

Bons estudos!

## Vamos Começar!

Quando você utiliza um software pode não perceber a complexidade dos processos necessários para garantir a confiabilidade, eficiência e segurança da aplicação. Para assegurar a qualidade dos processos de software, são empregados normas, métodos, ferramentas e metodologias.

De acordo com a definição de Sommerville (2018), no contexto do desenvolvimento de software, os processos são essenciais para organizar, gerenciar e compreender a sequência de atividades. Para proporcionar uma compreensão mais clara dos processos, tomemos como exemplo um projeto de desenvolvimento da página inicial de um site. Nesse caso, diversos processos podem ser identificados, tais como design de layout, tratamento de imagens, responsividade, entre

outros. É evidente que cada etapa do desenvolvimento pode ser orientada por normas específicas, visando atingir os objetivos desejados.

E por que a melhoria dos processos é importante para as atividades de desenvolvimento de software? À primeira vista, pode parecer que melhorar processos é aplicável apenas a atividades de linhas de produção, mas essa percepção não é precisa. Ao empregar ferramentas de qualidade de processos, o objetivo é corrigir erros e falhas, padronizar atividades e alinhar o trabalho com a equipe de desenvolvimento, entre outros benefícios. Antes de adotar qualquer metodologia de qualidade de processos, é essencial realizar o mapeamento detalhado desses processos (Sommerville, 2018).

O mapeamento de processos é uma ferramenta gerencial destinada a identificar a sequência na qual as atividades são executadas. Em uma perspectiva prática, o mapeamento pode proporcionar vantagens adicionais, tais como:

- **Compreensão dos processos:** possibilita a compreensão de todas as partes que os constituem.
- **Análise dos processos:** ao realizar o mapeamento, é viável refletir sobre as atividades que integram os processos.
- **Melhoria dos processos:** ao examinar minuciosamente as atividades que compõem os processos, é possível otimizá-los, ajustá-los ou substituí-los para aprimoramento.
- **Padronização dos processos:** as atividades alinhadas aos padrões de qualidade estabelecidos pelas políticas da empresa podem tornar-se padrão nos processos.
- **Documentação dos processos:** ao mapear as atividades que compõem os processos, é possível documentá-los, caso a equipe não tenha realizado essa etapa nas atividades anteriores à fase de desenvolvimento do projeto.

Percebeu que antes de qualquer coisa, devem-se mapear os processos para que se compreendam detalhadamente todas as atividades de desenvolvimento? Até aqui tudo certo. Contudo, qual é o nível de detalhamento adequado para o mapeamento? Conforme indicado por Sommerville (2018), há três níveis de detalhamento dos processos, como pode ser observado:

- **Nível 1 – Descritivo:** busca-se o alinhamento dos processos com as partes envolvidas no projeto de desenvolvimento por meio de uma descrição básica e abrangente.
- **Nível 2 – Analítico:** esta fase técnica detalha as atividades de desenvolvimento, os pontos de atenção e o tratamento de exceções.
- **Nível 3 – Executável:** oferece uma visão mais próxima aos dados e à aplicação em si. O objetivo aqui é detalhar as funcionalidades, os serviços e as saídas.

Dessa forma, após realizar o mapeamento dos processos, seja por meio de softwares ou através de descrições (como textos, tabelas ou quadros), é possível empregar ferramentas como modelos, normas e metodologias. Esses elementos constituem as formas pelas quais as equipes podem assegurar a qualidade dos processos de desenvolvimento de software. Entre essas ferramentas, abordaremos inicialmente os Modelos de Maturidade CMM e CMMI.

## Modelos de maturidade – CMMI

As conversas sobre a qualidade dos processos abrangem uma variedade significativa de perspectivas, uma vez que os processos de desenvolvimento podem ser analisados por diversos prismas. O Modelo de Maturidade de Capacidade (CMM) e o *Capability Maturity Model Integration* (CMMI) são modelos amplamente reconhecidos no âmbito do desenvolvimento de software, desempenhando papéis cruciais na avaliação e no aprimoramento dos processos organizacionais. Desenvolvido nos anos 1980, o CMM delineia cinco níveis de maturidade para representar a evolução progressiva das práticas em uma organização. Em contraste, o CMMI, introduzido em 2002, vai além dessa abordagem linear, apresentando constelações que integram diversas disciplinas, como desenvolvimento de sistemas, aquisição e serviços. Enquanto o CMM concentra-se em disciplinas separadas, o CMMI oferece uma visão mais abrangente e adaptável, permitindo que as organizações aprimorem continuamente suas práticas em sintonia com as demandas em constante evolução do cenário organizacional. Ambos os modelos representam ferramentas valiosas para guiar as organizações rumo à excelência nos processos e à entrega eficaz de produtos e serviços.

O CMMI apresenta duas abordagens principais: Representação por Estágios (*Staged*) e Representação Contínua (*Continuous*). A distinção fundamental entre essas duas abordagens reside na forma como as organizações implementam e progridem nos níveis de maturidade (Sommerville, 2018).

Na Representação por Estágios, a abordagem é sequencial, exigindo que uma organização alcance todos os objetivos de um determinado nível de maturidade antes de avançar para o próximo. A avaliação global é realizada com base na adoção e implementação integral de todos os processos associados ao nível de maturidade em questão. As organizações podem buscar certificação em um nível específico, demonstrando conformidade com todas as áreas de processo desse nível.

Por outro lado, na Representação Contínua, a abordagem é mais flexível. As organizações têm a liberdade de selecionar e implementar áreas de processo específicas de diferentes níveis de maturidade, de acordo com suas necessidades e prioridades. Isso permite que se concentrem em melhorar áreas específicas do processo sem a obrigação de atingir todos os objetivos de um nível antes de avançar. A avaliação é conduzida de maneira detalhada, avaliando cada área de processo individualmente para um controle mais granular do progresso e do aprimoramento (Sommerville, 2018).

Ambas as representações compartilham o objetivo final de promover a melhoria contínua dos processos e aumentar a maturidade organizacional. A escolha entre a abordagem por estágios ou pela contínua dependerá das necessidades específicas, dos objetivos e das capacidades da organização em questão.

Os cinco níveis no modelo CMMI por estágios são representados na Figura 1 e correspondem aos níveis de capacidade de um a cinco no modelo contínuo. Cada nível de maturidade possui um conjunto associado de áreas de processo e metas genéricas, refletindo as melhores práticas

em engenharia e gerenciamento de software, bem como a institucionalização da melhoria de processos. Os níveis de maturidade mais baixos podem ser alcançados introduzindo boas práticas; no entanto, os níveis mais elevados demandam um comprometimento com a medição e com o aprimoramento contínuo de processos.

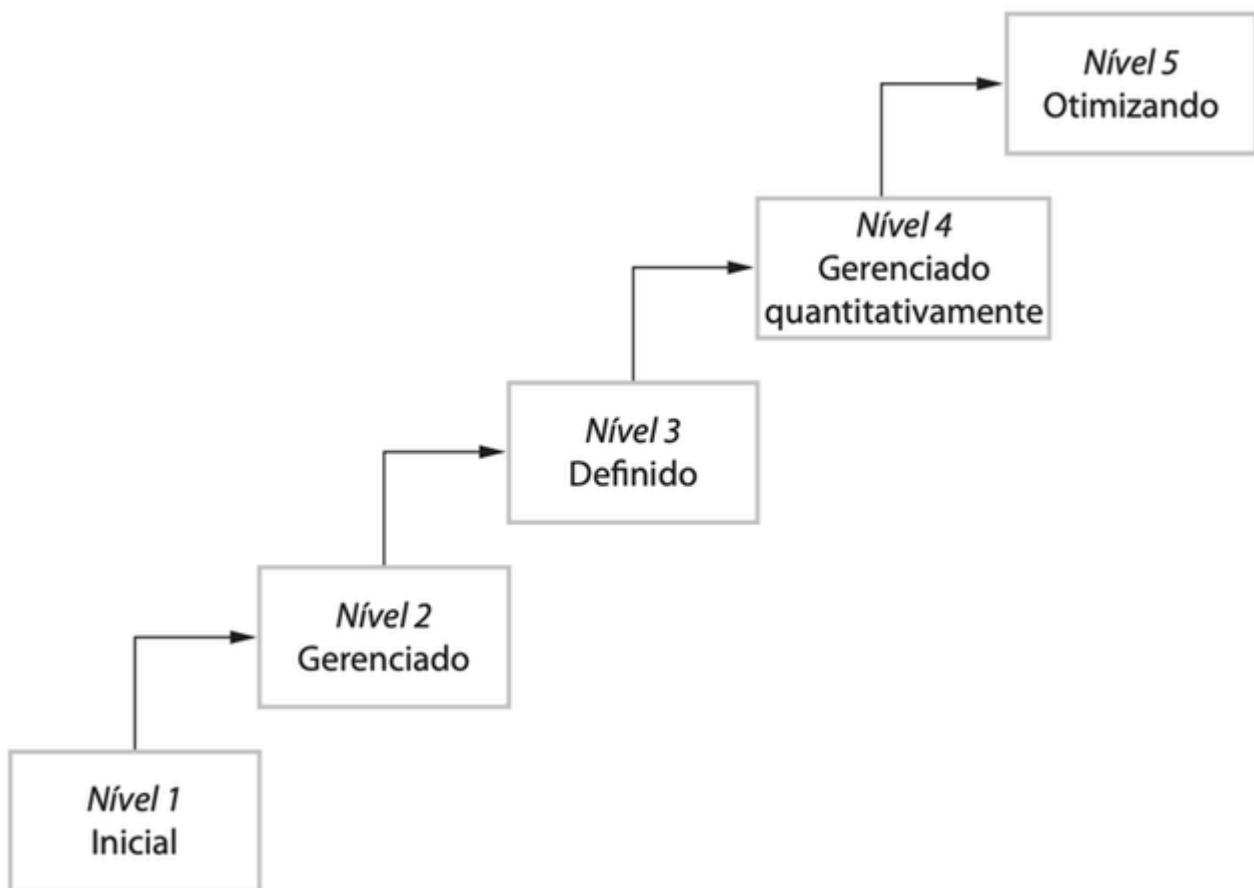


Figura 1 | O modelo de maturidade CMMI por estágios. Fonte: Sommerville (2018).

A principal vantagem do modelo CMMI por estágios é sua compatibilidade com o modelo de maturidade e capacidade de software proposto na década de 1980, o que facilita a aceitação e a adoção por muitas empresas em seus esforços de melhoria de processos. A transição do modelo original para o CMMI por estágios é relativamente simples para organizações que já se comprometeram com esse modelo. Além disso, o modelo por estágios estabelece um caminho de melhoria claro para as organizações, permitindo que planejem a transição de um nível para outro, como, por exemplo, do segundo para o terceiro nível.

No entanto, a principal desvantagem do modelo por estágios (e do CMM para Software) reside em sua natureza prescritiva. Cada nível de maturidade possui metas e práticas específicas, assumindo que todas devem ser implementadas antes da transição para o próximo nível. Entretanto, as circunstâncias organizacionais podem demandar uma abordagem mais flexível, tornando mais adequado implementar as metas e práticas em níveis superiores antes das

práticas de nível mais baixo. Quando uma organização opta por essa abordagem, uma avaliação de maturidade pode apresentar uma imagem distorcida de sua verdadeira capacidade.

Os modelos de maturidade contínuos não categorizam uma organização em níveis discretos; em vez disso, são abordagens mais refinadas que analisam práticas individuais ou conjuntos de práticas, avaliando a adoção de boas práticas em cada grupo de processos. Dessa forma, a avaliação de maturidade não se traduz em um único valor, mas em um conjunto de valores que indicam a maturidade da organização para cada processo ou grupo de processos.

No contexto do CMMI contínuo, são consideradas as áreas de processo e é atribuído um nível de avaliação de capacidade de zero a cinco para cada área de processo. É comum que organizações operem em diferentes níveis de capacidade para distintas áreas de processos. Portanto, o resultado de uma avaliação CMMI contínua é um perfil de capacidade que expõe cada área de processo e sua avaliação de capacidade correspondente. Um trecho de um perfil de capacidade, exemplificando processos em diferentes níveis de capacidade, é ilustrado na Figura 2. Nessa representação, é possível observar que o nível de capacidade em gerenciamento de configuração é elevado, enquanto a capacidade de gerenciamento de riscos é classificada como baixa. Empresas podem desenvolver perfis de capacidade reais e de metas, sendo o perfil-alvo uma expressão do nível de capacidade desejado para uma determinada área de processo.

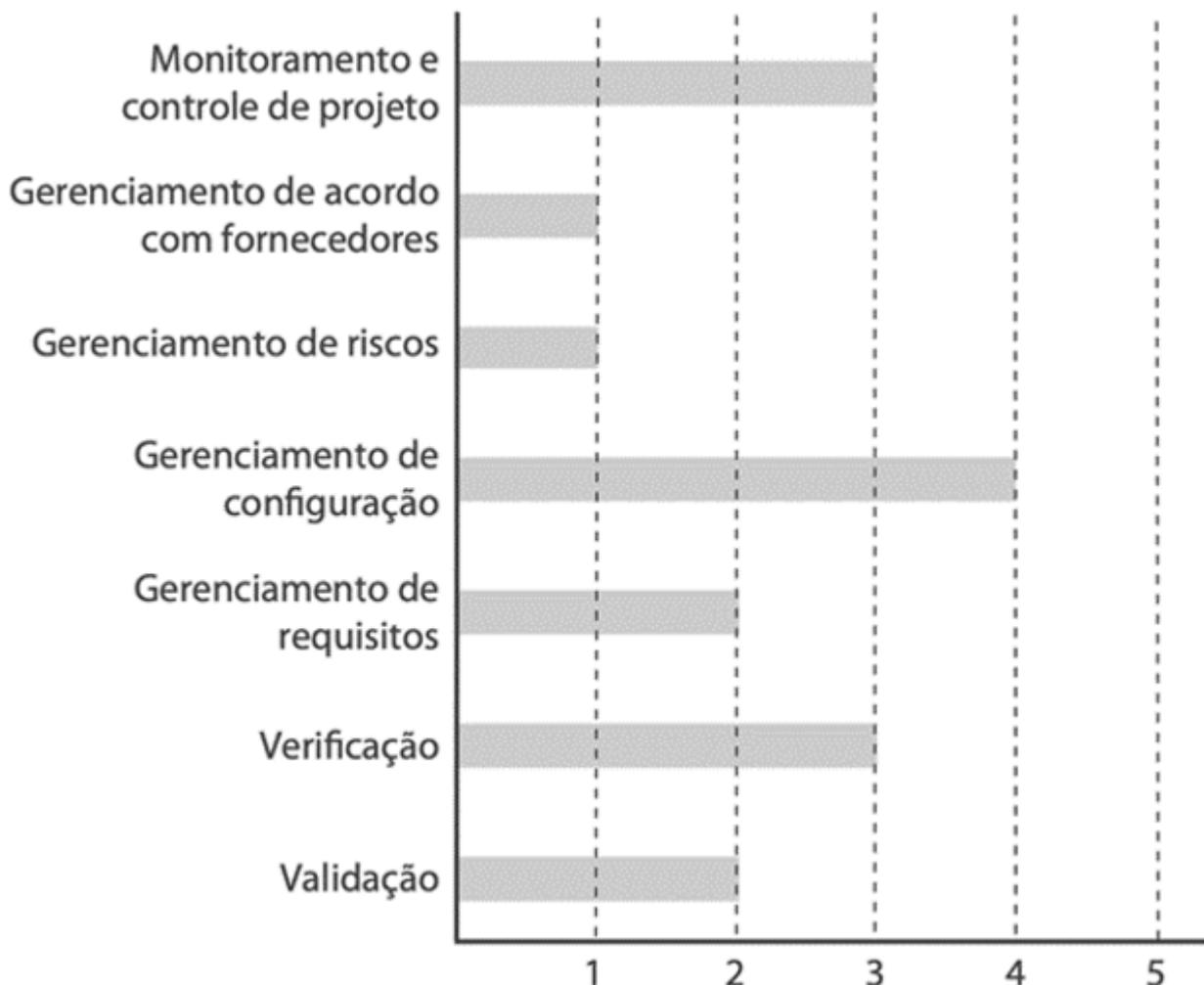


Figura 2 | Um perfil de capacidade de processos. Fonte: Sommerville (2018).

A principal vantagem do modelo contínuo reside na capacidade das empresas de selecionar os processos de melhoria de acordo com suas necessidades e seus requisitos específicos. Diferentes organizações apresentam distintos requisitos de melhoria de processos. Por exemplo, uma empresa dedicada ao desenvolvimento de software para a indústria aeroespacial pode direcionar seus esforços para melhorias na especificação de sistema, gerenciamento de configuração e validação, enquanto uma empresa de desenvolvimento web pode focar em processos relacionados ao atendimento ao cliente. Em contraste, o modelo por estágios impõe às empresas uma sequência específica de estágios a serem seguidos. No entanto, o CMMI contínuo oferece critérios e flexibilidade, permitindo que as empresas atuem dentro do framework de melhoria do CMMI de maneira mais adaptável e alinhada com suas necessidades particulares.

**Siga em Frente...**

## Melhorias de processo de Software brasileiro – MPS.BR

O Programa MPS-BR, estabelecido em dezembro de 2003 como uma iniciativa brasileira, busca aprimorar de forma contínua os processos de software. Uma das metas essenciais do MPS-BR é desenvolver e aperfeiçoar um modelo para a melhoria e avaliação de processos de software, com foco preferencial nas micro, pequenas e médias empresas, atendendo às suas necessidades de negócio e obtendo reconhecimento tanto nacional quanto internacional como um modelo aplicável à indústria de software.

Os componentes fundamentais do MPS-BR incluem o Modelo de Referência (MR-MPS), o Método de Avaliação (MA-MPS) e o Modelo de Negócio (MN-MPS), como apresentado na Figura 3.

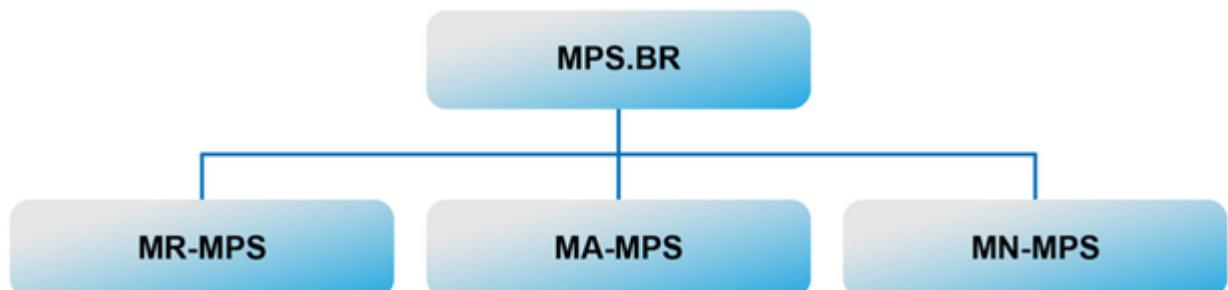


Figura 3 | Esquema MPS.BR. Fonte: Maitino (2021).

Para a compreensão das siglas que compõem o MPS.BR, acompanhe os tópicos a seguir:

- **Modelo de Referência (MR):** contém informações a forma como a organização deve conduzir os seus processos para atingir os resultados. Ainda é possível, nesse mesmo documento, determinar o nível de maturidade e da capacidade dos processos descritos na NBR ISO/IEC 12207.
- **Modelo de Avaliação (MA):** trata-se do processo no qual são determinados os parâmetros e requisitos para se aferir a qualidade do desenvolvimento. É baseado na norma ISO/IEC 15504.
- **Modelo de Negócio (MN):** determina o nível de maturidade dos processos, que podem ser: (A) Otimizado, (B) Gerenciado, (C) Definido, (D) Largamente definido, (E) Parcialmente definido, (F) Gerenciado e (G) Parcialmente gerenciado.

O MPS.BR é organizado em sete níveis de maturidade, cada um representando um estágio de aprimoramento nos processos. Os níveis são uma maneira de avaliar e guiar as organizações no desenvolvimento e na implementação de práticas mais maduras e eficientes. A seguir, estão os sete níveis do MPS.BR:

1. **Nível G (Inicial):** este é o nível inicial e básico. As práticas neste estágio são realizadas de maneira ad hoc, muitas vezes dependendo das habilidades individuais dos profissionais envolvidos.
2. **Nível F (Gerenciado):** neste estágio, a organização começa a adotar uma abordagem mais disciplinada para a gestão de processos. Há uma conscientização maior sobre a

importância de seguir processos definidos.

3. **Nível E (Definido):** Aqui, a organização estabelece processos padronizados e documentados. Há um esforço para definir e documentar processos para torná-los compreensíveis e consistentes.
4. **Nível D (Gerenciado Quantitativamente):** neste ponto, a organização começa a medir quantitativamente seus processos. Isso envolve a coleta e a análise de dados para entender melhor o desempenho do processo.
5. **Nível C (Definido Detalhado):** neste nível, os processos são refinados e otimizados. A organização trabalha em melhorias contínuas, identificando áreas específicas para aprimoramento.
6. **Nível B (Gerenciado Quantitativamente para o Negócio):** aqui, a organização expande a abordagem quantitativa para se alinhar melhor aos objetivos de negócios. A medição e a análise se tornam mais integradas aos objetivos organizacionais.
7. **Nível A (Em Otimização):** no nível mais alto, a organização busca a inovação e a otimização contínua. Os processos são ajustados e adaptados para atender às mudanças nas condições e nos objetivos organizacionais.

Esses níveis do MPS.BR fornecem um caminho claro para as organizações melhorarem seus processos de software, passando de abordagens ad hoc para práticas mais disciplinadas, mensuráveis e otimizadas. Cada nível representa um avanço na maturidade e na eficácia dos processos organizacionais.

Vale ressaltar que cada modelo apresenta o seu formulário específico: MR (Guia Geral e Guia de Aquisição), MA (Guia de Avaliação) e MN (Documentos do projeto). Segundo Rocha (2005), a construção das técnicas constituintes ao MPS.BR é composta pelas NBR ISO/IEC:

- NBR ISO/IEC 12207 para processo de ciclo de vida de software.
- ISO/IEC 15504 para avaliação de processo.
- ISO/IEC 15504-5 para modelo de avaliação de processo de software.

Na prática, como o MBS.BR deve ser iniciado no nível G, para o qual o manual determina 28 GPR, que são os objetivos que a equipe deve alcançar para atingir de maturidade. A seguir as três primeiras GPR do nível G, segundo MPS.BR (2012):

GPR 1. O escopo do trabalho para o projeto é definido.

GPR 2. As tarefas e os produtos de trabalho do projeto são dimensionados utilizando métodos apropriados.

GPR 3. O modelo e as fases do ciclo de vida do projeto são definidos.

Os GPRs necessitam de documentação, gestão e monitoramento. O manual sugere um modelo de documento, mas permite flexibilidade para que as empresas incluam ou removam campos conforme necessário.

## Normas ISO de qualidade de processos

A norma ISO 9001 pode ser uma ferramenta valiosa para os desenvolvedores de software que buscam aprimorar a qualidade de seus processos. De acordo com Valls (2003), a ISO 9001:2015 destaca-se por promover uma visão sistêmica que deve orientar as atividades de desenvolvimento. Isso implica que os profissionais devem possuir um conhecimento aprofundado dos processos, sendo essencial que esse entendimento seja uniforme entre os membros de toda a organização.

Valls (2003) argumenta ainda que a ISO 9001 representa um sistema de qualidade projetado para assegurar a otimização dos processos. Conhecida no âmbito profissional como Sistema de Gestão da Qualidade (SQA), essa norma é considerada pelos gestores de projetos de desenvolvimento de software como uma ferramenta robusta para a correção e para o aprimoramento contínuo dos processos.

Ao implementar o modelo ISO 9001:2015, é essencial estabelecer, manter e buscar aprimoramentos na gestão da qualidade, com foco nos processos e nas suas interações. Desta maneira, o objetivo claro é alcançar um entendimento aprofundado de cada componente dos processos e das relações entre eles, evitando qualquer dissociação entre as etapas.

Você notou como a compreensão das conexões entre as partes facilita a compreensão dos processos? Além dessa percepção, Valls (2003) destaca que, para a melhoria dos processos, as organizações devem ter uma compreensão nítida dos seguintes pontos:

- **Entradas e Saídas:** em todo software, é fundamental que as entradas e as saídas sejam explicitamente definidas desde a fase de levantamento de requisitos, contribuindo para o aprimoramento da maturidade nesse aspecto.
- **Sequência dos Processos:** a clareza e a definição precisam dos processos possibilitam que a equipe compreenda a sequência de trabalho, assegurando uma lógica bem estruturada em todas as etapas.
- **Interação dos Processos:** o conhecimento aprofundado dos processos viabiliza a compreensão das interações entre eles, permitindo a identificação de pontos críticos no projeto.
- **Recursos Disponíveis:** além dos recursos computacionais, a escassez, muitas vezes, está relacionada a prazos e à mão de obra especializada, sendo crucial considerar esses aspectos na gestão.
- **Responsabilidades:** cada membro da equipe deve ter clareza sobre suas atribuições no projeto, conhecendo as interações com colegas e reconhecendo as lideranças presentes.
- **Riscos:** uma compreensão aprofundada dos processos possibilita aos gestores identificar riscos e pontos críticos, permitindo a implementação de ações preventivas para garantir qualidade e cumprimento de prazos.

O processo de implementação da ISO 9001 é relativamente simples e aplicável a empresas de todos os portes. No entanto, obter o reconhecimento total por meio da certificação exige que a empresa já possua um sólido nível de maturidade em seus processos de desenvolvimento de

software, além de um conhecimento profundo das normas e das diretrizes que compõem a ISO 9001.

## Vamos Exercitar?

Nesta atividade, o mais indicado seria o MPS.BR, pois é um modelo brasileiro e possui características para pequenas empresas. Algumas características do MPS.BR são:

o MPS.BR é um modelo de maturidade brasileiro, o qual possui sete níveis de maturidade. Os níveis são:

Nível G:

1. GRE- Gerência de Requisitos.
2. GPR- Gerência de Projetos.

Nível F:

1. MED- Medição.
2. GQA- Garantia de Qualidade.
3. GCO- Gerência de Configuração.
4. AQU- Aquisição.
5. GPP- Gerência de Portfólio de Projeto.

Nível E:

1. GPR- Gerência de Projeto (evolução).
2. AMP- Avaliação e Melhoria do Processo Organizacional.
3. DFP- Definição do Processo Organizacional.
4. GRH- Gerência de Recursos Humanos.
5. GRU- Gerência de Reutilização.

Nível D:

1. DRE- Desenvolvimento de Requisitos.
2. ITP- Integração do Produto.
3. PCP- Projeto e Construção do Produto.
4. VAL- Validação.
5. Verificação.

Nível C:

1. DRU- Desenvolvimento para Reutilização.
2. GDE- Gerência de Decisões.

3. GRI- Gerência de Risco.

Nível B:

1. GPR- Gerência de Projetos

Nível A:

Não é necessário especificar os níveis, só foi utilizado no gabarito para ilustrar quais melhorias nos processos podem ocorrer com a utilização do MPS.BR.

## Saiba mais

Quer ler mais sobre o modelo de referência CMMI. Acesse o livro a seguir e leia o Capítulo 11.1. [Engenharia de Software - Hirama](#)

Quer ler mais sobre o modelo de referência MPS.BR. Acesse o livro a seguir e leia o Capítulo 11.2. [Engenharia de Software - Hirama](#)

No site oficial do [MPS.BR](#) você pode obter mais informações sobre o modelo de referência brasileiro.

## Referências

HIRAMA, K. **Engenharia de software**. Grupo GEN, 2011.

MAITINO NETO, R. **Engenharia de software**. Londrina: Editora e Distribuidora Educacional S.A., 2021.

SOFTEX. MPS BR. Softex, Brasília, [s. d.]. Disponível em: <https://softex.br/mpsbr/>. Acesso em: 22 abr. 2024.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

VALLS, V. M. A documentação na ISO 9001:2000. **Bases qualidade**, São Paulo, v. 12, n. 133, p. 100-105, jun. 2003.

## Aula 4

Auditoria de Sistemas

## Auditoria de sistemas



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você desvendará os fundamentos da Auditoria de Tecnologia da Informação. Nesta jornada, exploraremos os conceitos-chave, mergulhando nos Ciclos de Vida da Auditoria e na abordagem específica para Sistemas de Informações. Esses conhecimentos são essenciais para fortalecer sua prática profissional, capacitando-o a avaliar, garantir e aprimorar a segurança e a eficiência dos sistemas de informação. Prepare-se para adquirir habilidades cruciais no universo da auditoria tecnológica!

## Ponto de Partida

Você já ouviu falar sobre auditoria? Existem diversos segmentos que utilizam essa ferramenta de verificação a fim de gerar informações que permitem compreender as falhas de processos, de produtos e de gestão. As auditorias possibilitam o conhecimento, em detalhes, das atividades desenvolvidas, tornando possível a análise dos processos. Além disso, pode-se dizer que a auditoria não verifica somente falhas, mas também pontos de melhorias e potencialidades.

Você sabia que em Engenharia de Software também são utilizados os processos de auditoria? Evidentemente, aquela que é feita na área de desenvolvimento de sistemas possui algumas particularidades em relação à aplicada em outros segmentos, porém é preciso lembrar que o intuito comum a qualquer auditoria é compreender os processos, diagnosticar falhas ou pontos de melhorias e, posteriormente, buscar os ajustes necessários, ou seja, utilizá-la como instrumento de melhoria. Interessante, não é?

Sobre esse assunto, vamos supor que uma organização de renome no setor de tecnologia, está comprometida em garantir a segurança, integridade e eficácia operacional de seus sistemas de informação. Diante desse compromisso, a empresa decidiu realizar uma auditoria abrangente em seus sistemas, abordando áreas críticas de TI, práticas de segurança e conformidade com padrões regulatórios e internos.

O objetivo deste estudo de caso é explorar o ciclo de vida da auditoria de sistemas da empresa, destacando as fases de planejamento do cronograma, planejamento da auditoria, condução e relatório. Portanto, apresente uma compreensão detalhada de como a empresa estrutura e executa o processo de auditoria para assegurar a segurança de seus sistemas de informação.

Bons estudos!!!

## Vamos Começar!

### Conceitos de auditoria de tecnologia da informação

Normalmente, quando as pessoas abordam a experiência de passar por auditorias, os relatos frequentemente incluem momentos de tensão, pressão no trabalho, ajustes de conduta e revisões detalhadas das atividades profissionais.

Assim como em outras áreas do conhecimento, como fiscal, contábil e administrativa, a Engenharia de Software também incorpora processos de auditoria para avaliar o desenvolvimento de software. Dado que essas atividades requerem habilidades técnicas de programação, criatividade e aplicação precisa de processos, são naturalmente sujeitas a processos de auditoria.

Mas, de fato, o que é uma auditoria?

Conforme mencionado por Cardoso (2015), as atividades de auditoria têm como função principal analisar de forma parcial ou abrangente os processos, resultados e, em alguns casos, fornecer sugestões de ações corretivas ou melhorias. Essas atividades podem ser conduzidas por meio de diversos métodos, como questionários, testes práticos de uso, análise documental, entre outros, sendo essencial que os resultados obtidos não suscitem dúvidas.

Para ilustrar essa abordagem, consideremos o exemplo de uma empresa que necessita auditar o desenvolvimento de uma funcionalidade específica em um aplicativo de uma operadora de convênio odontológico, o qual possibilita o agendamento de consultas. A auditoria se tornou necessária devido a um substancial atraso na entrega do aplicativo. Esse processo visa identificar o momento exato do atraso e as razões que o causaram, informações cruciais para corrigir possíveis falhas no processo. Nesse cenário, a equipe de auditoria poderia analisar documentos ou realizar entrevistas com os envolvidos, obtendo dados suficientes para gerar informações valiosas na compreensão do problema.

É evidente que, apesar da existência de processos de auditoria para avaliar atividades em diversas áreas, a tecnologia da informação dispõe de métodos e normas específicas. As atividades de auditoria em Engenharia de Software são reconhecidas como auditoria em sistemas da informação. Essas atividades não se limitam à verificação de códigos, instruções e outras formas de codificação de uma aplicação computacional, mas também englobam a aplicação de esforços, conforme demonstrado a seguir (Cardoso, 2015):

- **Processos:** refere-se à maneira como são direcionados os métodos de execução das atividades de desenvolvimento. Em termos práticos, considere que a empresa adote o SCRUM com a finalidade específica de priorizar as atividades. A auditoria de processo poderia examinar detalhadamente se o SCRUM foi seguido ao longo do ciclo de vida do desenvolvimento do software.
- **Desenvolvimento:** envolvendo a análise dos scripts, essa fase consiste na verificação dos possíveis erros de escrita na linguagem de programação ou marcação utilizada. Um exemplo contemporâneo é a auditoria de responsividade em desenvolvimentos web, que examina a aplicação correta dos frameworks destinados a tornar a aplicação responsiva. Os processos de auditoria revelarão, por exemplo, quais dispositivos apresentam erros ou desconforto de navegação, oferecendo avaliações cruciais para a qualidade dos desenvolvimentos.
- **Testes:** nesta etapa, a auditoria busca verificar a eficácia dos testes realizados nos desenvolvimentos de software. Em termos profissionais, isso implica que as atividades executadas pelos testadores de software serão auditadas para assegurar se as funcionalidades desenvolvidas estão sendo testadas adequadamente ou se há falhas, vícios ou outras inconformidades.
- **Segurança e proteção dos dados:** envolve as ações realizadas para garantir a segurança e a proteção dos dados, visando evitar incidentes indesejados. Por exemplo, durante o processo de auditoria, a utilização de pentest (teste de invasão realizado por profissional de segurança da informação) pode ser empregada para verificar, na prática, a segurança de um sistema.
- **Estrutura de desenvolvimento:** concentra-se nas atividades administrativas e de recursos humanos, no ambiente de trabalho e nos recursos disponíveis para o desenvolvimento profissional. Por exemplo, durante suas análises, o auditor pode revisitar as atividades profissionais para avaliar se o ambiente de trabalho favorece um bom desempenho dos profissionais de desenvolvimento de software.

Sem dúvida, é evidente a amplitude das áreas em que auditorias podem ser conduzidas no contexto do desenvolvimento de software. Contudo, é crucial compreender as atribuições e as responsabilidades de um elemento fundamental nos processos de auditoria de software: o auditor.

Conforme destacado por Gallotti (2016), o auditor é um profissional que possui conhecimentos técnicos, gerenciais, operacionais e analíticos necessários para conduzir as atividades relacionadas ao desenvolvimento de auditorias em sistemas da informação.

Para compreender o papel do auditor, observe o Quadro 1.

TIPOS	ATRIBUIÇÃO
Inspeção	Efetuar inspeção dos processos e dos desenvolvimentos.

	Verificar onde ocorrem inconformidades e apresentar provar convincentes.
Controle	Efetuar a análise de controle dos processos, podendo ser operacionais, de desenvolvimento ou gerenciais.
Risco	Verificar quais riscos podem afetar o projeto, a fim de se indicar a equipe na qual existe um risco. Isso permite que o gerente de projetos possa tomar medidas corretivas/preventivas. Analisar de forma mais abrangente os riscos, apresentando consequências externas que possam ocorrer.
Contínua	Elaborar relatórios das análises em espaços curtos de tempo. Utilizar sistemas de verificação constantes para que se faça uma análise de diversos momentos. Contratar especialistas da área em que é necessária uma análise mais profunda, a fim de se permitir o aumento do detalhamento dos processos.

Quadro 1 | Atribuições e responsabilidades do auditor de tecnologia da informação. Fonte: adaptado de Gallotti (2016).

No que diz respeito às responsabilidades e atribuições de um auditor de desenvolvimento de software, consideremos um exemplo: suponha que uma empresa de desenvolvimento de jogos para dispositivos móveis optou por criar um jogo exclusivo para um público específico. Um processo de auditoria poderia identificar os impactos positivos e negativos de um jogo com um tema que exclui determinados jogadores. Entretanto, a decisão sobre o produto elaborado cabe aos níveis administrativos e gerenciais da empresa, os quais, com base nos resultados das auditorias, decidirão sobre o rumo a ser seguido. Embora uma auditoria aponte falhas, erros e pontos de atenção, ela não impõe à empresa a necessidade de adotar outras direções; essa decisão é de responsabilidade da própria empresa.

Além disso, é importante salientar que as atividades de auditoria de software são fundamentadas em testes, análises, avaliações e outras abordagens que evidenciem a existência de inconformidades ou a execução inadequada (Gallotti, 2016).

## Ciclos de vida da auditoria

Você percebeu a importância da auditoria na Engenharia de Software? Em nossas discussões, destacamos que as auditorias não devem ser encaradas como um processo doloroso e negativo, mas sim como uma ferramenta para identificar falhas, contribuindo para a qualidade do projeto, da equipe e da organização, refletindo positivamente no atendimento ao cliente. Contudo, na prática, em qual fase do projeto de desenvolvimento de software a auditoria deve ser empregada?

Os processos de auditoria em sistemas da informação podem ser aplicados ao longo de todo o ciclo de vida do projeto de desenvolvimento. O ciclo de vida da auditoria é composto por quatro processos, abrangendo desde as atividades preparatórias anteriores à auditoria propriamente dita até o seu acompanhamento (Weill e Ross, 2006). Esses processos de auditoria são sequenciais, e para entender a organização dessas atividades, consulte a Figura 1.



Figura 1 | Ciclos de vida da auditoria. Fonte: adaptada de Gallotti (2016).

A elaboração do cronograma de auditoria é flexível, adaptando-se ao nível de detalhamento exigido pelo projeto. A simplicidade, com informações básicas como atividade, datas inicial e final, e responsável, é suficiente, embora outras informações possam ser adicionadas. Em muitos casos, a aprovação do planejamento do cronograma pela contratante é crucial. Por exemplo, a auditoria de um aplicativo de votação on-line. Se a data final do relatório estiver marcada para após o dia da votação, a aprovação do cronograma é vital para evitar conflitos e garantir que a auditoria seja realizada conforme as necessidades do projeto (Vetorazzo, 2018).

O planejamento da auditoria, como primeira etapa, inicia-se com o mapeamento dos processos, identificando os agentes responsáveis por cada um. Durante esse processo, o auditor pode consultar documentos de auditorias anteriores para direcionar suas atividades com base em apontamentos feitos. Uma das atividades essenciais é a definição clara do objetivo da auditoria, permitindo que a empresa de desenvolvimento de software ajuste-se às necessidades do cliente. A divulgação desse objetivo é crucial, como destacado por Rainer (2016), pois fornece clareza sobre quem será auditado, o que será auditado e quando a auditoria ocorrerá. Essa transparência facilita a focalização da auditoria nas áreas específicas que podem impactar diretamente no

desempenho da aplicação, evitando auditorias em setores desnecessários, como exemplificado com equipes de *front end* que não impactariam diretamente a performance do sistema.

Vetorazzo (2018) ressalta que após o planejamento da auditoria ser elaborado e aprovado, inicia-se a fase de execução, na qual são aplicadas atividades de coleta de informações, como revisão documental, conversas, entrevistas, análise de dados de processos e observações, entre outras técnicas. O autor destaca a importância do checklist como um guia para o auditor, contendo os pontos a serem auditados. Esse checklist é fundamental para permitir análises abrangentes, e sua estrutura pode variar de acordo com o detalhamento necessário para o projeto, com informações básicas, como atividades, datas, horários, locais e responsáveis, podendo incluir outros dados conforme a necessidade.

De acordo com a fase de encerramento da auditoria consiste em reuniões, relatórios e apresentações que fornecem feedback sobre o que foi auditado. Essa etapa possibilita a discussão de melhorias e ajustes, dependendo do que foi acordado na contratação do serviço. No entanto, é ressaltado que a empresa ou equipe de auditoria não é responsável por implementar as mudanças sugeridas no relatório. Na prática, esse momento marca o fim de um período de auditoria, proporcionando à equipe auditada a oportunidade de conhecer os resultados, identificar acertos e erros, e projetar objetivos e melhorias futuras. Geralmente, é realizada uma reunião com a equipe auditada e os cargos administrativos para a apresentação dos resultados e a discussão de sugestões (Vetorazzo, 2018).

**Siga em Frente...**

## Abordagem de auditoria de sistemas de informações

As abordagens de auditoria de Tecnologia da Informação (TI) ou Sistemas de Informação (SI) fundamentam-se na avaliação dos riscos empresariais relacionados à validação das transações econômicas, financeiras e contábeis, considerando a perspectiva da efetividade, especialmente diante de recursos limitados. Ao realizar auditorias em ambientes de tecnologia da informação, o auditor tem a flexibilidade de escolher abordagens adequadas. As abordagens comuns incluem aquelas centradas no computador, aquelas realizadas através do computador e aquelas executadas com o computador. A auditoria tradicional, historicamente associada à verificação da confiabilidade dos registros em conformidade com documentos-fonte, evoluiu devido ao impacto contínuo da tecnologia na gestão, tornando essencial armazenar informações de forma acessível à auditoria. Com a complexidade crescente dos ambientes, especialmente com a expansão para intranets e internet, surgem desafios relacionados à vulnerabilidade dos computadores e casos potenciais de fraudes. Dependendo da complexidade do sistema computadorizado e das características do auditor de sistemas de informações, ele pode optar por uma das três abordagens mencionadas: (1) abordagem ao redor do computador; (2) abordagem através do computador; e (3) abordagem com o computador (Imoniana, 2016).

### Abordagem ao redor do computador

No passado, a auditoria ao redor do computador era uma abordagem muito solicitada pelos auditores, devido ao pouco envolvimento com a tecnologia de informação. A premissa subjacente a essa abordagem sugere que, se os dados de entrada estiverem corretos e os procedimentos correlatos estiverem em conformidade com os padrões operacionais e controles predefinidos, e se as saídas estiverem alinhadas com as expectativas, então os procedimentos lógicos de processamento tornam-se menos relevantes. Essa metodologia exige que o auditor avalie os níveis de concordância relacionados à implementação dos controles organizacionais no contexto da tecnologia da informação.

Isso implica conduzir uma auditoria nos documentos-fonte, examinando as funções de entrada subjacentes e compreendendo as funções de saída, as quais estão apresentadas em formatos de linguagem comprehensíveis para aqueles que não são especialistas em informática. Pouca ou nenhuma atenção é dedicada às operações lógicas de processamento. O sistema de processamento eletrônico de dados é utilizado apenas para tarefas mais simples, como a obtenção de níveis de estoque e sugestões para realimentação quando necessário. Operações simples, como isolar estoques com pouca movimentação, gerar estatísticas de *packing list* em relação à distribuição e identificar estoques obsoletos, bem como elaborar uma *aging list* de duplicatas por meio de transações computadorizadas, são realizadas principalmente nas funções de impressão de relatórios.

Existe frequentemente uma ponderação sobre a aplicação desse método, questionando sua eficácia como uma prática sólida de auditoria. No entanto, a hesitação em adotá-lo pode ser atribuída à capacidade desses sistemas de realizar operações contábeis simples, levando os auditores a considerá-los como tarefas concluídas, avaliando as entradas e as saídas básicas dos sistemas. É importante observar que, embora essa abordagem possa não ser a mais adequada para ambientes complexos, ela ainda é bastante conveniente para sistemas menores, nos quais a maioria das atividades de rotina é realizada manualmente. Essa abordagem é baseada na afirmação de que as entradas do sistema podem ser consideradas corretas se os resultados do sistema refletem com precisão os dados-fonte. Portanto, a saída também deve ser correta, e as maneiras pelas quais o sistema processou os dados têm pouca consequência. Vale ressaltar que, até o momento, ao adotar essa abordagem, os auditores têm pouco ou nenhum envolvimento direto com os registros gerados pelo computador, limitando-se a realizar consultas, rastrear dados, etc., sem acessar diretamente os programas do sistema.

Esta abordagem apresenta vantagens notáveis, como a não exigência de um amplo conhecimento em tecnologia de informação por parte do auditor, tornando as técnicas e as ferramentas de auditoria acessíveis e válidas. Além disso, sua aplicação é economicamente eficiente, envolvendo custos baixos e diretos. No entanto, as desvantagens são evidentes, incluindo uma restrição operacional devido à falta de compreensão sobre como os dados são atualizados, resultando em auditorias incompletas e inconsistentes devido à dinâmica do processo operacional. A avaliação da eficiência operacional também se torna mais desafiadora, pois não há parâmetros claros e padronizados. A abordagem pode levar a uma avaliação de risco inadequada, pois não exige que o auditor tenha uma capacidade profissional significativa em tecnologia de informação. Além disso, a exclusão das Unidades Centrais de Processamento (CPU) e funções aritméticas e lógicas em procedimentos de auditoria pode resultar em uma

representação não abrangente da tecnologia de informação da organização, distorcendo as decisões baseadas nos relatórios resultantes dessas auditorias.

## Abordagem através do computador

A aplicação dessa abordagem vai além da simples comparação entre documentos-fonte e resultados esperados, uma vez que os sistemas têm experimentado significativa evolução. No entanto, este método alerta para questões relacionadas ao manuseio de dados, à aprovação e ao registro de transações econômicas, financeiras e contábeis, sem deixar evidências documentais razoáveis por meio dos controles de programas incorporados nos sistemas. Por essa razão, o auditor necessita monitorar o processamento tanto através quanto dentro do computador. Essa abordagem aprimora a metodologia de auditoria centrada no computador. Ao utilizar este método de auditoria, uma pessoa pode solicitar, de diversas maneiras, semelhante à abordagem centrada no computador, a verificação dos documentos-fonte com dados intermediários. No entanto, destaca-se a ênfase do auditor em técnicas que empregam o computador como uma ferramenta para testar tanto a própria tecnologia quanto a entrada de dados.

Aqueles que apoiam o uso da abordagem centrada no computador favorecem a utilização da técnica de test data, que consiste no processamento de um dispositivo capaz de simular todas as transações possíveis com dados fictícios e reais.

Esta abordagem apresenta vantagens notáveis, incluindo a capacitação aprimorada do auditor em termos de habilidades profissionais relacionadas ao processamento eletrônico de dados. Além disso, permite ao auditor verificar com maior frequência as áreas que requerem revisão constante. No entanto, as desvantagens são consideráveis, destacando que, se a operação for realizada incorretamente, pode resultar em perdas incalculáveis, desaconselhando sua execução no ambiente de produção. O uso da abordagem pode ser dispendioso, especialmente no que se refere ao treinamento de auditores, aquisição e manutenção de pacotes de software. Considerando que os pacotes podem estar incorretos, técnicas manuais podem ser necessárias como complemento para assegurar a efetividade da abordagem. Além disso, há o risco de que os pacotes possam estar contaminados devido ao uso frequente na auditoria organizacional.

## Abordagem com o computador

A primeira alternativa, a abordagem centrada no computador, demonstra falta de eficiência devido à sua negligência em relação a algumas das características dos controles internos dos sistemas, resultando na ausência de testes substantivos convincentes que são essenciais para chegar a conclusões robustas sobre os sistemas. Por outro lado, a segunda opção, a abordagem através do computador, considerada superior à primeira, também pode levar à produção de registros incompletos. Em vez de realizar uma verificação equilibrada com as ferramentas disponíveis, essa abordagem tem a tendência de negligenciar os procedimentos manuais, deixando incompletas a maioria das tarefas que normalmente são executadas manualmente.

Diante dessas considerações, empresas de auditoria e pesquisadores no campo contábil têm proposto um método para auditar as tecnologias de informações e sistemas de maneira mais

aprimorada, empregando a abordagem completamente assistida pelo computador.

Realiza-se uma compilação do processo, aparentemente alcançando certos objetivos, tais como:

- Utilização das capacidades lógicas e aritméticas do computador para verificar a precisão dos cálculos em transações econômicas, financeiras e contábeis, bem como em responsabilidades como depreciações, impostos e taxas, multiplicação e contabilização (footings).
- Aplicação das capacidades de cálculos estatísticos e geração de amostras para facilitar confirmações de saldos necessárias para avaliar a integridade de dados em contas a receber, estoques, ativos fixos, advogados e circularização, entre outros testes com terceiros.
- Utilização das capacidades de edição e classificação do sistema computadorizado para ordenar e selecionar registros contábeis. Por exemplo, ao analisar a base de dados do sistema de estoque, um auditor pode identificar com precisão os itens de movimentação mais lenta ou obsoleta, isolando-os dos itens de movimentação rápida, facilitando análises mais complexas e substantivas.
- Aplicação das capacidades matemáticas do computador para analisar e fornecer listas de amostras de auditoria, seja por estratificação ou por unidade monetária de materialidade. Isso pode incluir a confirmação dos resultados de auditoria executados manualmente, como os cálculos globais.

Uma grande vantagem dessa abordagem é a disponibilidade e o uso eficaz de:

- Capacidades de auditoria para aplicar Técnicas de Auditoria Assistida por Computador (TAAC), também conhecidas como *Computer Assisted Audit Techniques* (CAAT).
- Possibilidades de desenvolver programas específicos para o uso do auditor ao evidenciar uma opinião sobre o processo contábil.
- Ganho de tempo com os passos aplicados no pacote generalizado de auditoria de tecnologia da informação.
- Integração de todos os processos de auditoria com os papéis de trabalho de auditoria, disponibilizando-os em uma rede de auditoria da firma, acessível via internet para toda a equipe, desde o trainee até o sócio.

## Vamos Exercitar?

Este estudo de caso tem como propósito investigar o ciclo de vida da auditoria de sistemas de uma empresa, focando as etapas de elaboração do cronograma, o planejamento da auditoria, a execução e a elaboração do relatório. A intenção é apresentar uma análise minuciosa da maneira como a empresa organiza e conduz o processo de auditoria para garantir a segurança de seus sistemas de informação. Portanto, uma possível solução para este estudo de caso a seguir:

### 1. Planejamento do Cronograma:

O departamento de TI é responsável por fornecer suporte essencial para os processos de negócios da empresa.

## Atividades:

- **Identificação de Prazos Cruciais:**
  - A equipe de auditoria, composta por auditores internos e externos, realiza uma reunião para identificar os prazos essenciais para a organização.
  - Estabelecimento de prazos para conclusão de projetos críticos, datas de lançamento de novos produtos e revisões regulares de segurança.
- **Análise de Disponibilidade de Recursos:**
  - Avaliação da disponibilidade de recursos, incluindo a equipe de auditoria, colaboradores internos, e ferramentas necessárias para conduzir a auditoria.
  - Planejamento para garantir que a auditoria não impacte adversamente as operações regulares da empresa.
- **Definição de Marcos e Metas:**
  - Estabelecimento de marcos de progresso e metas específicas para cada fase da auditoria.
  - Definição de objetivos claros para garantir que a auditoria esteja alinhada com os objetivos estratégicos da organização.
- Avaliação da disponibilidade de recursos, incluindo a equipe de auditoria, colaboradores internos, e ferramentas necessárias para conduzir a auditoria.
- Planejamento para garantir que a auditoria não impacte adversamente as operações regulares da empresa.
- Estabelecimento de marcos de progresso e metas específicas para cada fase da auditoria.
- Definição de objetivos claros para garantir que a auditoria esteja alinhada com os objetivos estratégicos da organização.

## 2. Planejamento da Auditoria:

Com base no cronograma estabelecido, a equipe de auditoria começa a planejar a abordagem detalhada para a auditoria.

## Atividades:

- **Identificação de Áreas Críticas:**
  - Revisão das áreas críticas de TI que requerem atenção especial, incluindo sistemas de banco de dados, servidores, controle de acesso e políticas de segurança.
  - Avaliação de riscos associados a cada área crítica identificada.
- **Seleção de Metodologia de Auditoria:**
  - Escolha da metodologia de auditoria mais apropriada, levando em consideração a estrutura regulatória, os padrões da indústria e os requisitos internos.
  - Planejamento detalhado de como cada área crítica será avaliada.

- **Design de Testes e Ferramentas:**
  - Desenvolvimento de testes específicos e seleção de ferramentas de auditoria adequadas para avaliar a eficácia dos controles existentes.
  - Preparação de questionários e entrevistas para coletar informações relevantes.
- Escolha da metodologia de auditoria mais apropriada, levando em consideração a estrutura regulatória, os padrões da indústria e os requisitos internos.
- Planejamento detalhado de como cada área crítica será avaliada.
- Desenvolvimento de testes específicos e seleção de ferramentas de auditoria adequadas para avaliar a eficácia dos controles existentes.
- Preparação de questionários e entrevistas para coletar informações relevantes.

## 3. Condução da Auditoria

Com o plano de auditoria em vigor, a equipe começa a executar as atividades planejadas.

### Atividades:

- **Coleta de Evidências:**
  - Realização de testes de segurança, análises de logs e revisão de políticas e procedimentos.
  - Entrevistas com pessoal-chave para validar práticas e controles.
- **Monitoramento em Tempo Real:**
  - Monitoramento contínuo de sistemas durante a auditoria para identificar qualquer atividade suspeita ou não autorizada.
  - Adaptação do plano conforme necessário com base em descobertas em tempo real.
- **Documentação Detalhada:**
  - Registro detalhado de todas as descobertas, as evidências coletadas e os resultados de testes.
  - Documentação de qualquer não conformidade ou área de melhoria identificada.
- Monitoramento contínuo de sistemas durante a auditoria para identificar qualquer atividade suspeita ou não autorizada.
- Adaptação do plano conforme necessário com base em descobertas em tempo real.
- Registro detalhado de todas as descobertas, as evidências coletadas e os resultados de testes.
- Documentação de qualquer não conformidade ou área de melhoria identificada.

## 4. Reporte:

Após a conclusão da auditoria, a equipe elabora um relatório detalhado para apresentar aos stakeholders.

## Atividades:

- **Análise de Resultados:**
  - Avaliação aprofundada das descobertas, destacando áreas de conformidade, não conformidade e recomendações para melhorias.
  - Classificação de riscos associados a cada descoberta.
- **Preparação do Relatório:**
  - Elaboração de um relatório de auditoria completo, incluindo uma visão geral do processo, da metodologia utilizada, das descobertas, da análise de riscos e das recomendações.
  - Apresentação clara de ações corretivas sugeridas e seus impactos.
- **Compartilhamento com Stakeholders:**
  - Apresentação do relatório aos principais stakeholders, incluindo a alta administração, equipe de TI e outros departamentos relevantes.
  - Discussão das descobertas, esclarecimento de dúvidas e obtenção de aprovação para implementação de ações corretivas.
- Elaboração de um relatório de auditoria completo, incluindo uma visão geral do processo, da metodologia utilizada, das descobertas, da análise de riscos e das recomendações.
- Apresentação clara de ações corretivas sugeridas e seus impactos.
- Apresentação do relatório aos principais stakeholders, incluindo a alta administração, equipe de TI e outros departamentos relevantes.
- Discussão das descobertas, esclarecimento de dúvidas e obtenção de aprovação para implementação de ações corretivas.

Este estudo de caso ilustra como o ciclo de vida da auditoria de sistemas pode ser aplicado em um ambiente de uma empresa de tecnologia, desde o planejamento do cronograma até o reporte final às partes interessadas. A abordagem estruturada contribui para garantir a eficácia das práticas de segurança e o alinhamento contínuo com os objetivos organizacionais.

## Saiba mais

O livro [Auditoria de Sistemas de Informação](#) apresenta de forma detalhada o tema abordado nesta aula.

Quer saber mais sobre as Auditoria de SI, acesse o artigo: [Auditoria de sistemas de informação: saiba o que e como é feita.](#)

O plano de contingência e de recuperação de desastres significa medidas operacionais estabelecidas e documentadas para serem seguidas, no caso de ocorrer alguma indisponibilidade dos recursos de informática, evitando-se que o tempo no qual os equipamentos fiquem parados acarrete perdas materiais aos negócios da empresa. Para entender mais como a

auditoria trabalha com essa questão, acesse o link a seguir: [AUDITORIA DE PLANO DE CONTINGÊNCIA E DE RECUPERAÇÃO DE DESASTRES.](#)

## Referências

CARDOSO, A. **Auditoria de sistema de gestão integrada.** São Paulo: Pearson Education do Brasil, 2015.

GALLOTTI, G. M. A. **Qualidade de software.** São Paulo: Pearson Education do Brasil, 2016.

IMONIANA, J. O. **Auditoria de sistemas de informação.** 3. ed. Grupo GEN, 2016.

VETORAZZO, A. de S. **Engenharia de software.** Porto Alegre: SAGAH, 2018.

WEILL, P.; ROSS, J. W. **Governança de TI:** como as empresas com melhor desempenho administram os direitos decisórios de TI na busca por resultados superiores. São Paulo: M. Books do Brasil Editora Ltda., 2006.

## Aula 5

Encerramento da Unidade

### Videoaula de Encerramento

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá a Qualidade de Software, Qualidade de Processo e Auditoria de Sistemas de Informação. Estes temas são fundamentais para aprimorar sua prática profissional, destacando a importância de entregar software confiável, otimizar processos e garantir a segurança dos sistemas de informação. Prepare-se para adquirir conhecimentos essenciais que impulsionarão sua excelência no universo da tecnologia.

## Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta unidade, que é saber aplicar os conceitos relacionados à qualidade de produto e de processo, nós precisamos começar com os conceitos da qualidade de software. Em linhas gerais, a qualidade de software implica a aplicação eficaz da gestão de qualidade para criar um produto útil com valor mensurável para desenvolvedores e usuários finais. Um software de alta qualidade deve atender às exigências explícitas e implícitas dos usuários, proporcionando confiabilidade e gerando benefícios, como a redução de manutenção e suporte, permitindo maior inovação, agilizando processos de negócio e resultando em maior receita, rentabilidade e disponibilidade de informações cruciais.

O termo "qualidade de software", frequentemente associado à conformidade funcional e desempenho conforme expectativas do patrocinador, segundo Sommerville (2018), aborda a qualidade em três níveis: organizacional, focado em estabelecer padrões para minimizar erros; de projeto, envolvendo o desenvolvimento conforme padrões estabelecidos pelos gestores de projetos; e de planejamento, exigindo a elaboração de um plano de qualidade e revisões para evitar falhas inadvertidas nos processos e produtos desenvolvidos.

Outro ponto importante é a Garantia de Qualidade de Software (SQA), que é uma abordagem sistemática e proativa para assegurar a qualidade dos processos e produtos relacionados ao desenvolvimento de software. Seu objetivo principal é garantir que as atividades de software sejam planejadas, implementadas e monitoradas de maneira eficaz, alinhando-se aos padrões e procedimentos estabelecidos. A SQA abrange uma ampla gama de atividades, desde a definição de processos até a realização de auditorias e revisões, visando identificar e corrigir desvios em relação às normas estabelecidas.

No contexto da SQA, são implementados práticas e métodos para melhorar continuamente a qualidade do software, antecipando e prevenindo defeitos em vez de corrigi-los após a implementação. Isso envolve a criação de planos de qualidade, a realização de revisões de código, o estabelecimento de métricas e indicadores de desempenho, além da promoção de uma cultura organizacional que valorize a qualidade em todas as fases do ciclo de vida do software. A SQA desempenha um papel crucial no fornecimento de confiança aos stakeholders, garantindo que os produtos de software atendam aos requisitos e padrões estabelecidos, resultando em sistemas mais confiáveis, eficientes e alinhados com as expectativas dos usuários.

Um ponto importante da qualidade de software é a medição. Essa prática envolve a obtenção de valores numéricos ou perfis para atributos de componentes, sistemas ou processos de software, comparando esses valores entre si e com padrões para avaliar a eficácia de métodos, ferramentas e processos. O objetivo a longo prazo é substituir revisões para julgar a qualidade do software, embora avaliações automatizadas ainda não tenham atingido esse ideal. Métricas de software, como tamanho do código, índice Fog, número de defeitos e complexidade ciclomática, desempenham um papel crucial na tomada de decisões de gerenciamento. As métricas de controle guiam mudanças nos processos, enquanto as métricas de previsão auxiliam na estimativa do esforço necessário para implementar alterações no software. Há duas abordagens para o uso de medições em um sistema de software: atribuir valores aos atributos de qualidade

do sistema ou identificar componentes que não atendem aos padrões de qualidade, destacando características individuais que se desviam da norma.

As métricas de produto são medidas de previsão que avaliam atributos internos de um sistema de software, como tamanho do código e complexidade ciclomática. No entanto, essas características facilmente mensuráveis não têm uma relação clara e consistente com atributos de qualidade, como compreensibilidade e manutenibilidade, devido às variações nos processos de desenvolvimento, tecnologias e tipos de sistemas. As métricas de produto se dividem em métricas dinâmicas, coletadas durante a execução do programa, e métricas estáticas, obtidas a partir de representações estáticas como projeto, código-fonte ou documentação. Esses tipos de métricas estão associados a diferentes atributos de qualidade, em que métricas dinâmicas avaliam eficiência e confiabilidade, enquanto métricas estáticas ajudam a avaliar complexidade, compreensibilidade e manutenibilidade do sistema ou de seus componentes. Métricas dinâmicas, como tempo de execução e número de falhas, possuem uma relação mais evidente com as características de qualidade do software em comparação com métricas estáticas.

Quando falamos em qualidade, o ISO é fundamental para este tema. A norma NBR 13596, também conhecida como ISO/IEC 9126, estabelece padrões e diretrizes para a avaliação de qualidade de software. Essa norma internacional fornece um framework abrangente para medir a qualidade do software em termos de suas características e subcaracterísticas. A ISO/IEC 9126 define seis atributos principais de qualidade do software, que incluem funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. Cada atributo é subdividido em diversas subcaracterísticas que abrangem aspectos específicos da qualidade do software.

Além de fornecer uma estrutura para avaliação, a norma ISO/IEC 9126 é uma ferramenta valiosa para desenvolvedores, testadores e stakeholders envolvidos no ciclo de vida do software. Ao adotar essas diretrizes, as organizações podem medir, avaliar e melhorar a qualidade de seus produtos de software, promovendo assim a entrega de sistemas mais confiáveis, eficientes e alinhados às expectativas dos usuários.

A ISO 9000 e a ISO 9001 são normas internacionais que estabelecem diretrizes e requisitos para sistemas de gestão da qualidade. A ISO 9000 fornece uma visão geral dos conceitos fundamentais e do vocabulário relacionados à gestão da qualidade, enquanto a ISO 9001 é uma norma específica que define os critérios para um sistema de gestão da qualidade eficaz. A ISO 9001 estabelece requisitos detalhados que as organizações devem atender para obter a certificação de seu sistema de gestão da qualidade. Essas normas visam garantir a consistência, eficiência e melhoria contínua nos processos de uma organização, proporcionando confiança aos clientes e às partes interessadas de que os produtos e serviços atendem aos padrões de qualidade estabelecidos. A implementação e certificação de acordo com as normas ISO 9000 e ISO 9001 são reconhecidas internacionalmente e contribuem para aprimorar a reputação e a competitividade das organizações no mercado global.

A qualidade do produto está diretamente ligada com o processo de desenvolvimento, portanto ter qualidade no processo é indispensável. O *Capability Maturity Model Integration* (CMMI) é um modelo de melhoria de processos que fornece diretrizes detalhadas para organizações

aprimorarem seus processos de desenvolvimento e manutenção de software. Desenvolvido pelo *Software Engineering Institute* (SEI) da Universidade Carnegie Mellon, o CMMI visa promover a maturidade dos processos, fornecendo uma estrutura para avaliação e melhoria contínua. Ele se baseia em cinco níveis de maturidade, que vão desde o inicial até o otimizado, cada um representando um estágio específico de maturidade dos processos. Ao adotar o CMMI, as organizações podem avaliar e aprimorar a eficácia de seus processos, aumentar a qualidade do software e a satisfação do cliente, além de proporcionar uma base sólida para a implementação de práticas de gerenciamento de projetos mais eficientes.

A implementação bem-sucedida do CMMI geralmente resulta em benefícios tangíveis, como aumento na eficiência operacional, redução de defeitos, melhoria da previsibilidade e melhor alinhamento com as metas estratégicas da organização. Além disso, o CMMI é aplicável a diversas áreas além do desenvolvimento de software, incluindo engenharia de sistemas, aquisição e serviços. Sua abordagem flexível e abrangente permite que as organizações adaptem e personalizem as práticas de acordo com suas necessidades específicas, tornando-o um guia valioso para alcançar a excelência nos processos.

O Modelo de Referência para Melhoria de Processo de Software (MPS.Br) é uma iniciativa brasileira que visa promover a melhoria contínua dos processos de desenvolvimento e manutenção de software nas organizações. Inspirado em modelos internacionais, como o CMMI, o MPS.Br é adaptado para atender às necessidades específicas do contexto brasileiro. Ele oferece um conjunto de práticas e critérios que auxiliam as empresas a avaliar e aprimorar seus processos, visando maior eficiência, qualidade e satisfação do cliente.

O MPS.Br é organizado em níveis de maturidade, desde o G (Gestão) até o A (Otimizado), proporcionando uma trajetória evolutiva para as organizações aprimorarem seus processos. A obtenção de uma certificação MPS.Br reconhecida internacionalmente demonstra o comprometimento da organização com a excelência e a melhoria contínua. Além disso, o modelo tem se consolidado como uma referência importante para o setor de tecnologia no Brasil, contribuindo para elevar os padrões de qualidade e competitividade das empresas no cenário global.

A auditoria de sistemas desempenha um papel essencial ao avaliar a conformidade, integridade e eficácia dos processos de desenvolvimento de software, contribuindo para garantir e aprimorar a qualidade dos produtos e serviços entregues. A auditoria desempenha um papel essencial no desenvolvimento de software, desafiando-se a analisar e aprimorar processos e resultados, identificando oportunidades para correções e melhorias. Essas atividades, conforme explicado por Cardoso (2015), empregam métodos diversos, incluindo questionários, testes práticos de uso e análise documental, visando fornecer resultados claros e sem ambiguidades. Um exemplo prático desse processo ocorre em uma empresa de convênio odontológico, em que a equipe de auditoria busca entender as razões e o momento exato de um significativo atraso na entrega de uma funcionalidade no aplicativo, fornecendo informações valiosas para corrigir possíveis falhas.

No campo da Engenharia de Software, as atividades de auditoria em sistemas de informação ultrapassam a mera verificação de códigos e instruções, abrangendo processos,

desenvolvimento, testes, segurança de dados e estrutura de desenvolvimento. Por exemplo, na auditoria de processo, a equipe pode analisar a conformidade com metodologias específicas, como o Scrum, ao longo do ciclo de vida do desenvolvimento do software. A auditoria de desenvolvimento verifica erros na linguagem de programação, enquanto a auditoria de testes avalia a eficácia dos testes realizados nos desenvolvimentos. A segurança dos dados é validada por meio de técnicas como o *pentest*, e a auditoria da estrutura de desenvolvimento examina aspectos administrativos, de recursos humanos e do ambiente de trabalho. A abrangência dessas áreas destaca a importância do papel do auditor em garantir a qualidade e eficácia no desenvolvimento de software.

As abordagens de auditoria de Tecnologia da Informação (TI) ou Sistemas de Informação (SI) visam avaliar os riscos empresariais associados à validação de transações econômicas, financeiras e contábeis, priorizando a efetividade diante de recursos limitados. No contexto da auditoria em ambientes de tecnologia da informação, os auditores têm flexibilidade para escolher entre abordagens centradas no computador, realizadas através do computador e executadas com o computador. A evolução da auditoria tradicional, anteriormente focada na verificação da confiabilidade de registros em conformidade com documentos-fonte, é impulsionada pelo impacto contínuo da tecnologia na gestão, exigindo a acessibilidade adequada às informações para auditoria. Com a crescente complexidade dos ambientes, incluindo intranets e internet, surgem desafios relacionados à segurança dos computadores e possíveis casos de fraudes, levando os auditores de sistemas de informações a escolherem entre três abordagens: ao redor do computador, através do computador e com o computador (Imoniana, 2016).

No passado, a abordagem de auditoria ao redor do computador era amplamente adotada devido à limitada familiaridade com a tecnologia da informação. Essa metodologia se baseia na suposição de que, se os dados de entrada estiverem corretos e os procedimentos relacionados estiverem em conformidade, e as saídas estiverem alinhadas com as expectativas, os procedimentos lógicos de processamento tornam-se menos relevantes. O auditor concentra-se em examinar a implementação dos controles organizacionais na tecnologia da informação, realizando auditorias nos documentos-fonte e nas funções de entrada e saída, com pouca atenção às operações lógicas de processamento. Apesar de sua conveniência em ambientes menores, essa abordagem enfrenta críticas quanto à sua eficácia, pois pode resultar em auditorias incompletas e inconsistentes devido à dinâmica operacional, além de limitar a avaliação da eficiência operacional e a compreensão abrangente da tecnologia de informação da organização.

Apesar de apresentar vantagens, como a acessibilidade para auditores com pouco conhecimento em tecnologia de informação e custos baixos, essa abordagem possui desvantagens notáveis. A falta de compreensão sobre a atualização de dados pode levar a auditorias incompletas, enquanto a exclusão de funções cruciais e a falta de parâmetros padronizados dificultam a avaliação eficaz. A abordagem também pode resultar em avaliações de risco inadequadas e distorções nas decisões organizacionais, uma vez que não exige um profundo conhecimento em tecnologia de informação por parte do auditor e exclui aspectos essenciais da infraestrutura tecnológica.

A abordagem ao redor do computador vai além da mera comparação entre documentos-fonte e resultados esperados, considerando a evolução significativa dos sistemas. No entanto, destaca a preocupação com o manuseio de dados, a aprovação e o registro de transações sem evidências documentais adequadas pelos controles de programas nos sistemas. O auditor é orientado a acompanhar o processamento dentro e através do computador, melhorando a metodologia de auditoria centrada no computador. Apesar de permitir verificações semelhantes à abordagem centrada no computador, destaca-se a ênfase em técnicas que usam o computador para testar a tecnologia e a entrada de dados.

Aqueles que apoiam a abordagem centrada no computador favorecem a técnica de test data, que simula transações com dados fictícios e reais. Embora forneça benefícios, como aprimoramento das habilidades do auditor em processamento eletrônico de dados e revisões mais frequentes, apresenta desvantagens significativas. A execução incorreta pode resultar em perdas, sendo desaconselhável no ambiente de produção. O custo associado ao treinamento de auditores, aquisição e manutenção de pacotes de software, além do risco de contaminação, são considerações importantes. Técnicas manuais podem ser necessárias como complemento para garantir a efetividade da abordagem.

A abordagem com o computador na auditoria destaca-se pela sua ênfase no uso das capacidades do computador para aprimorar as atividades de análise e verificação. Nesse método, o auditor se beneficia das capacidades lógicas, aritméticas e matemáticas do computador para avaliar a precisão de cálculos em transações econômicas, financeiras e contábeis. Além disso, são aplicadas técnicas estatísticas e de geração de amostras para confirmar saldos e avaliar a integridade de dados em diversas áreas, como contas a receber, estoques e ativos fixos. A edição e classificação de registros contábeis são otimizadas pelo uso do sistema computadorizado, proporcionando uma análise mais eficaz. A abordagem com o computador também se destaca pela aplicação de Técnicas de Auditoria Assistida por Computador (TAAC), desenvolvimento de programas específicos, economia de tempo e integração de processos de auditoria em uma rede acessível online para toda a equipe auditora. Essa metodologia representa uma evolução na forma como a auditoria aproveita a tecnologia para alcançar eficiência e precisão nas suas atividades.

## É Hora de Praticar!



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Apresente uma breve explicação sobre o MPS.BR (Melhoria de Processo do Software Brasileiro) e o *Capability Maturity Model Integration* (CMMI). Destaque suas principais características, seus objetivos e a importância na gestão de processos de software.

Explore as semelhanças e as diferenças entre o MPS.BR e o CMMI. Analise as áreas de processo, os níveis de maturidade e como cada modelo aborda a melhoria contínua nos processos de software. Destaque as peculiaridades que cada um traz para aprimorar a qualidade e eficiência.

Discuta as vantagens de integrar o MPS.BR e o CMMI na melhoria de processos de software.

Considere como a combinação dessas abordagens pode resultar em benefícios sinérgicos, otimizando a gestão de projetos e promovendo a qualidade do produto final.

- Como a implementação consistente de práticas de garantia de qualidade pode influenciar diretamente na satisfação do usuário e na eficácia de um software?
- De que maneira a implementação do MPS.Br (Melhoria de Processo do Software Brasileiro) pode promover a excelência nos processos de desenvolvimento de software e aprimorar a qualidade dos produtos em organizações brasileiras?
- Como a auditoria de sistemas pode contribuir para fortalecer a segurança da informação e garantir a conformidade em um ambiente empresarial cada vez mais dinâmico e suscetível a ameaças cibernéticas?

O MPS.BR é uma iniciativa brasileira que visa a melhoria de processos de software em organizações. Desenvolvido e mantido pela Associação para Promoção da Excelência do Software Brasileiro (Softex) em parceria com a comunidade de software brasileira, o MPS.BR é baseado em normas e padrões internacionais, adaptados à realidade do setor de software no Brasil. Seus principais objetivos incluem promover a melhoria contínua nos processos, aumentar a maturidade das organizações e fortalecer a competitividade no mercado.

O CMMI é um modelo global que oferece uma abordagem integrada para o desenvolvimento de software, sistemas e serviços. Desenvolvido pelo Software Engineering Institute (SEI) da Universidade Carnegie Mellon, o CMMI visa melhorar a eficácia e a eficiência organizacional, fornecendo um conjunto de práticas e diretrizes que abrangem o ciclo de vida completo do desenvolvimento de produtos e serviços.

## Semelhanças e Diferenças:

### Semelhanças:

- Ambos os modelos visam a melhoria contínua dos processos.
- Ambos são baseados em boas práticas e padrões internacionais.
- Ambos oferecem um conjunto de áreas de processo e níveis de maturidade.

### Diferenças:

- O MPS.BR foi desenvolvido especificamente para atender às necessidades do setor de software brasileiro, enquanto o CMMI é aplicável globalmente.
- O CMMI é mais abrangente, cobrindo não apenas o desenvolvimento de software, mas também sistemas e serviços.

- O MPS.BR é estruturado em três níveis (A, B e C), enquanto o CMMI utiliza cinco níveis de maturidade.

## Vantagens da Integração:

- Complementaridade: a integração permite aproveitar as fortalezas de cada modelo, cobrindo lacunas específicas e proporcionando uma visão mais abrangente.
- Reconhecimento Internacional: a integração possibilita que as organizações atendam a requisitos globais de qualidade e melhoria de processos.
- Sinergia na Melhoria: a combinação de abordagens promove benefícios sinérgicos, resultando em uma gestão de projetos mais eficiente e produtos finais de maior qualidade.

Este gabarito serve como um guia para discussão e análise, podendo ser expandido com exemplos específicos e casos práticos durante a atividade discursiva.

Explore o mapa mental que traz de forma concisa informações sobre o MPS.BR, CMMI, ISO 9126 e ISO 9000. Uma referência direta para compreender os padrões e modelos essenciais em qualidade e gestão.

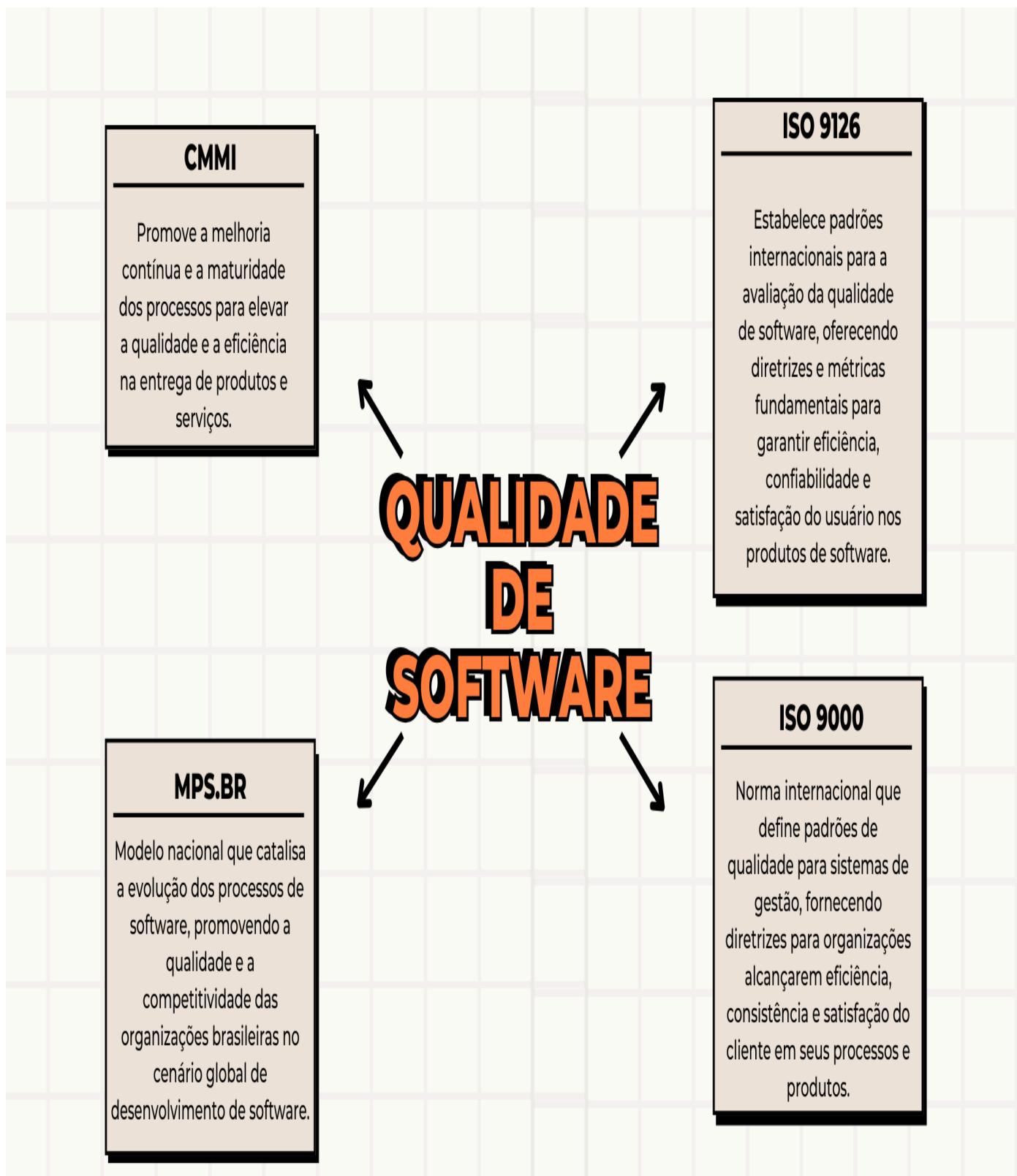


Figura | Qualidade de software.

CARDOSO, A. **Auditoria de sistema de gestão integrada**. São Paulo: Pearson Education do Brasil, 2015.

IMONIANA, J. O. **Auditoria de sistemas de informação**. 3 ed. Grupo GEN, 2016.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Unidade 3

### Teste de Software

#### Aula 1

Conceitos de testes de software

#### Conceitos de testes de software

##### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá os conceitos de Verificação e Validação de software. Abordaremos desde os conceitos fundamentais até a prática de testes de software e elaboração de casos de teste. Esses conteúdos são essenciais para a sua prática profissional, fornecendo as habilidades necessárias para garantir a qualidade e confiabilidade dos produtos de software que você desenvolve. Esteja pronto para aprimorar sua expertise e se destacar no mercado de tecnologia.

#### Ponto de Partida

Não é de hoje que a elevada qualidade de um produto, além de alguns outros fatores de natureza mais subjetiva, é o que conta para fazer dele um produto de sucesso em seu segmento. Um carro, um aparelho de televisão ou um telefone celular, por exemplo, tendem a seguir como produtos comercialmente viáveis se atenderem a certos requisitos de qualidade identificados com os seus propósitos e com o nicho a que se destinam. Naturalmente essa premissa também continuará sendo verdadeira se usarmos um software como exemplo e, nesse contexto específico, as providências para se garantir bons níveis de qualidade vêm sendo aprimoradas, especialmente aquela que é o tema central desta unidade: o teste de software.

Na aula de hoje, exploraremos os fundamentos essenciais de Verificação e Validação de software, examinando suas diferenças e sua importância na garantia da qualidade do produto final. Além disso, abordaremos os conceitos fundamentais dos testes de software, desde suas definições até suas abordagens práticas na detecção de falhas e na melhoria da confiabilidade do software. Em particular, focaremos na elaboração eficaz de casos de teste, destacando sua relevância na verificação do comportamento do software e na validação de suas funcionalidades em relação aos requisitos estabelecidos.

Para exercitarmos o que foi aprendido, vamos imaginar o seguinte problema: um sistema de inscrição on-line permite que os usuários se inscrevam em eventos, preenchendo um formulário que inclui um campo de data para inserir a data de nascimento. Alguns usuários reportaram problemas ao inserir suas datas de nascimento, no qual o sistema rejeita datas válidas ou aceita datas inválidas. O sistema precisa validar corretamente as datas, considerando anos bissextos, a quantidade correta de dias em cada mês e anos históricos válidos.

Por isso, você deve criar um conjunto de casos de teste focados na validação do campo de data de nascimento no formulário de inscrição do sistema. Seus casos de teste devem explorar diferentes cenários de entrada de data para identificar problemas na lógica de validação do sistema.

Para cada caso de teste, você deve especificar:

1. **Nome do Caso de Teste:** uma descrição breve e descriptiva.
2. **Pré-condições:** qualquer configuração ou estado do sistema necessário antes da execução do teste.
3. **Passos:** passos detalhados para executar o teste.
4. **Dados de Teste:** a data específica que será testada.
5. **Resultado Esperado:** o que se espera que aconteça se a validação da data estivesse funcionando corretamente (aceitação da data válida ou rejeição da data inválida).
6. **Critérios para Falha:** descrição específica do que seria considerado uma falha no teste (por exemplo, aceitação de uma data inválida).

Vamos começar?!

Bons Estudos!!!

## Vamos Começar!

### Verificação e Validação

Houve um tempo em que os sistemas computacionais estavam confinados aos desktops, principalmente para propósitos corporativos como controle de vendas e estoques. Hoje em dia, a gama de dispositivos que dependem de sistemas para operar é vasta, oferecendo flexibilidade e confiabilidade muito além do que seria possível sem aplicativos computacionais. Nossos carros e televisores são apenas dois exemplos disso, com programas em execução em praticamente todos os lugares, controlando dispositivos essenciais em nosso dia a dia. Mas o que aconteceria se esses programas falhassem?

Para evitar que essa pergunta tenha que ser respondida da pior maneira, a Engenharia de Software desenvolveu mecanismos para garantir que os produtos de software – tanto os comuns quanto aqueles que conferem "inteligência" aos nossos equipamentos – passem por processos que garantam sua capacidade de executar suas funções de maneira adequada e com altos padrões de qualidade. Estes processos incluem verificação, validação e testes de software. Embora possam parecer semelhantes à primeira vista, uma definição precisa servir para esclarecer e diferenciar esses conceitos.

No entanto, antes de individualizarmos esses termos, vale a pena posicioná-los em um contexto mais amplo, relacionado à qualidade de um produto, e conhecido por Gerenciamento da Qualidade do Software (ou *Software Quality Management*). Nesse contexto, a verificação e a validação são colocadas como ações intimamente relacionadas, destinadas à averiguação da conformidade do produto com as necessidades do cliente e comumente referenciadas em conjunto, sob a sigla V&V.

Conforme uma relevante publicação na área de Engenharia de Software (IEEE, 2014), o propósito da Verificação e Validação (V&V) é auxiliar a organização de desenvolvimento de software a integrar qualidade ao sistema ao longo do ciclo de vida, por meio de avaliações objetivas dos produtos e processos. Essas avaliações determinam se os requisitos são precisos, completos, consistentes e testáveis, demonstrando se estão corretos. Os processos de V&V também avaliam se os produtos desenvolvidos em uma atividade específica estão em conformidade com seus requisitos e se satisfazem sua finalidade pretendida.

As atividades de V&V não se limitam apenas a programas ou funções, mas se estendem a qualquer artefato resultante de uma etapa do ciclo de vida de um produto. Requisitos, projetos e implementações são exemplos de artefatos que devem ser submetidos a verificações e validações.

Seguindo essa linha, a IEEE (2014) define que a verificação busca garantir a construção correta do produto, assegurando que ele atenda às especificações estabelecidas nas fases anteriores do processo. Por outro lado, a validação busca garantir que o produto certo seja construído, ou seja, que ele cumpra sua finalidade específica.

Os conceitos de verificação e de validação podem ser relacionados a momentos específicos do ciclo de vida de um produto de software. Nesses termos a verificação se refere ao processo de determinar se um fluxo de trabalho foi executado corretamente ou não, e ela ocorre ao final de cada fluxo de trabalho. Já a validação é o processo intensivo de avaliação que ocorre imediatamente antes de o produto ser entregue ao cliente; seu propósito é o determinar se o produto como um todo satisfaz ou não às suas especificações (Schach, 2009).

Embora frequentemente confundidos e contextualmente interligados, verificação e validação não são conceitos intercambiáveis. A verificação diz respeito à confirmação das especificações do software em uma fase específica do desenvolvimento, enquanto a validação abrange a garantia do produto de software como um todo, confrontando-o com as expectativas do cliente.

Apesar de expressões como "incorporar qualidade ao sistema" e "verificar a conformidade de uma atividade com os requisitos" transmitirem uma aparente simplicidade procedural, a implementação de processos de verificação e a validação demanda um planejamento meticuloso e execução precisa. O objetivo do planejamento de V&V é assegurar que cada recurso, função e responsabilidade estejam claramente definidos.

Durante esta fase de planejamento, é essencial especificar os recursos, suas funções e atividades, assim como as técnicas e ferramentas a serem empregadas. Uma compreensão abrangente dos distintos propósitos de cada atividade de V&V facilita o planejamento cuidadoso das técnicas e dos recursos necessários para alcançar seus objetivos. Além disso, o planejamento deve abordar aspectos de gestão, comunicação, políticas e procedimentos relacionados às atividades de V&V.

Como já mencionado, verificação e validação estão inseridas em um escopo mais amplo, ligadas à Garantia da Qualidade do Software (*Software Quality Assurance* ou SQA). Segundo a perspectiva de Pressman (2021), verificação e validação abrangem uma variedade de atividades de SQA, tais como revisões técnicas, auditorias de qualidade, monitoramento de desempenho, simulação, estudo de viabilidade, teste de usabilidade, teste de aceitação e teste de instalação. Os autores também destacam que, embora os testes desempenhem um papel crucial em V&V, muitas outras atividades são necessárias.

Falando em testes, é relevante discutir seu papel nesse contexto. O teste representa a última linha de defesa na garantia da qualidade, porém, não pode ser considerado como uma garantia total de que o produto entregue pelas equipes está isento de falhas. Os testes fornecem o último meio pelo qual a qualidade pode ser avaliada e, de maneira prática, as falhas podem ser identificadas. A qualidade é incorporada ao software por meio da aplicação adequada das técnicas de Engenharia de Software e é validada durante os testes (Pressman, 2021).

## Conceitos de testes de software

Um teste não se trata de um procedimento isolado, concluído por um único membro da equipe. Embora comumente referido como "teste", sua realização depende de uma série de ações e procedimentos executados por diversos membros da equipe de desenvolvimento. Por essa

razão, seria mais apropriado chamá-lo de processo de teste, visto que formalmente ele representa uma sequência de ações executadas com o propósito de identificar problemas no software, ampliando assim a percepção geral de sua qualidade e garantindo que o produto final atenda às necessidades do usuário.

É importante ressaltar, no entanto, que o objetivo do teste não é assegurar que um programa esteja completamente livre de defeitos, mas sim identificar problemas no software. Apesar de essa premissa soar estranha (e um tanto frustrante), ela decorre do fato de que nenhum teste é capaz de garantir 100% de ausência de defeitos em um sistema. Portanto, se o processo de teste não revelar defeitos, é necessário aprimorar o processo como um todo. Não se pode presumir que o sistema esteja livre de problemas apenas porque o teste não os identificou.

Mas, afinal, o que é teste de software? Como ele se efetiva? Há um plano a ser executado?

O teste de software consiste na verificação dinâmica de que um programa, de fato, fornece comportamentos esperados em um conjunto finito de casos de teste adequadamente selecionados do domínio de execução geralmente infinito (IEEE, 2014).

É comum que uma definição extraída de uma publicação de natureza estritamente técnica requeira uma explicação que a torne mais compreensível para o público em geral em relação ao conceito de teste.

O termo "verificação dinâmica" sugere que a aplicação de um teste implica, necessariamente, a execução do programa. Além disso, essa execução deve se basear em entradas selecionadas previamente. Como discutiremos em maior detalhe adiante, os casos de teste consistem em entradas fornecidas ao programa e nas saídas correspondentes esperadas. Teoricamente, as possíveis entradas para um programa são infinitas, o que justifica a necessidade de restringi-las a um "conjunto finito de casos de teste" escolhidos cuidadosamente.

Bem, mas como os testes são feitos? Há uma ferramenta computacional que os execute? Nas seções futuras desta unidade, teremos a oportunidade de abordar esses assuntos com mais riqueza de detalhes, mas convém termos agora uma rápida visão de um programa sendo testado. Para que isso seja possível, algumas premissas devem ser apresentadas:

- O programa em teste foi criado e está sendo executado no Eclipse, um importante ambiente integrado de desenvolvimento (ou *Integrated Development Environment – IDE*) utilizado principalmente para criação e para teste de aplicações Java.
- O tipo de teste em questão é o de unidade. Por meio dele, apenas uma unidade do programa (uma função ou uma classe, por exemplo) é testada e não o programa todo.
- A ferramenta utilizada para a efetivação do teste é o JUnit, que já se encontra instalada nativamente no Eclipse.

Na Figura 1, você vê uma classe – chamada CalculoTest –, que representa um caso de teste. Note que há variáveis com os valores 10 e 5 e uma terceira variável que contém o valor esperado para a execução da unidade que, no caso, realiza uma operação de soma.

```

1 package processos;
2 import junit.framework.TestCase;
3 public class CalculoTest extends TestCase{
4     public void testeExecutaCalculo() {
5         //Define os valores a serem calculados e testados
6         float PassaValor1 = 10;
7         float PassaValor2 = 5;
8         float RetornoEsperado = 15;
9         //Executa o método "ExecutaCalculo" da classe Calculo e
10        //armazena o resultado em uma variável
11        float RetornoFeito = Calculo.ExecutaCalculo(PassaValor1, PassaValor2);
12        //compara o valor retornado com o valor esperado
13        assertEquals (RetornoEsperado, RetornoFeito, 0);
14    }
15
16
17 }

```

Figura 1 | Classe de teste escrita em Java. Fonte: Medeiros (2009, [s. p.]).

A classe a ser testada para este caso de teste está descrita na Figura 2 e se chama Cálculo.

```

1 package processos;
2 public class Calculo {
3     public void testeExecutaCalculo() {
4         public static float ExecutaCalculo (float Valor1, float Valor2) {
5             float Soma = Valor1 + Valor2;
6             return Soma;
7
8         }
9     }
10
11
12 }

```

Figura 2 | Classe de teste escrita em Java. Fonte: Medeiros (2009, [s. p.]).

A execução da classe CalculoTest através do JUnit retornará sucesso para o caso de teste em questão. Dessa forma, uma função pode ser testada por meio dessa ferramenta.

Bem, uma vez que entendemos o conceito de teste, pode-se supor erroneamente que sua realização depende unicamente da vontade do desenvolvedor em executar o programa com base em alguns casos de teste. No entanto, responder afirmativamente a essa questão seria ignorar uma etapa crucial neste contexto: o planejamento.

Conforme ensinado por Pressman e Maxim (2018), o teste é um conjunto de atividades que requer planejamento antecipado e execução baseada em procedimentos padronizados, o que justifica a criação de um modelo para o teste. Segundo os autores, esse modelo deve incluir o uso de técnicas específicas no design dos casos de teste e no método de teste.

Uma parte essencial do plano de testes é definir quem será responsável por executá-los. Como pondera Schach (2008), a realização dos testes pode ser vista como um processo destrutivo, e é

compreensível que um programador não queira "destruir" seu próprio trabalho. Portanto, atribuir a atividade de teste à mesma equipe que desenvolveu o produto pode não ser uma decisão sábia, pois há uma grande probabilidade de a equipe sentir a necessidade de proteger seu programa e, consequentemente, não criar testes que revelem problemas no código.

Outra razão para evitar a designação dos criadores do programa como seus testadores é o fato de que um terceiro poderá detectar uma falha no entendimento dos requisitos que passou despercebida ao programador e que foi implementada incorretamente. Um teste feito por outras pessoas pode aumentar a chance de descoberta do problema antes que ele tenha reflexos na operação do cliente.

Com isso em mente, é importante ressaltar que um plano de teste geralmente é dividido em quatro principais etapas:

- **Planejamento:** nesta fase, é essencial determinar quem será responsável pela execução dos testes, quando serão realizados, quais recursos serão necessários (como ferramentas e computadores), e qual técnica será empregada (por exemplo, técnica estrutural ou funcional).
- **Projeto de casos de teste:** aqui são estabelecidos os casos de teste que serão utilizados no processo. Este conceito será detalhado no próximo item.
- **Execução do programa com os casos de teste:** nesta etapa, os testes são efetivamente realizados, executando-se o programa com os casos de teste selecionados.
- **Análise dos resultados:** nesta fase, é feita uma análise dos resultados obtidos nos testes para verificar se foram satisfatórios.

A Figura 3 ilustra um modelo de plano de teste.

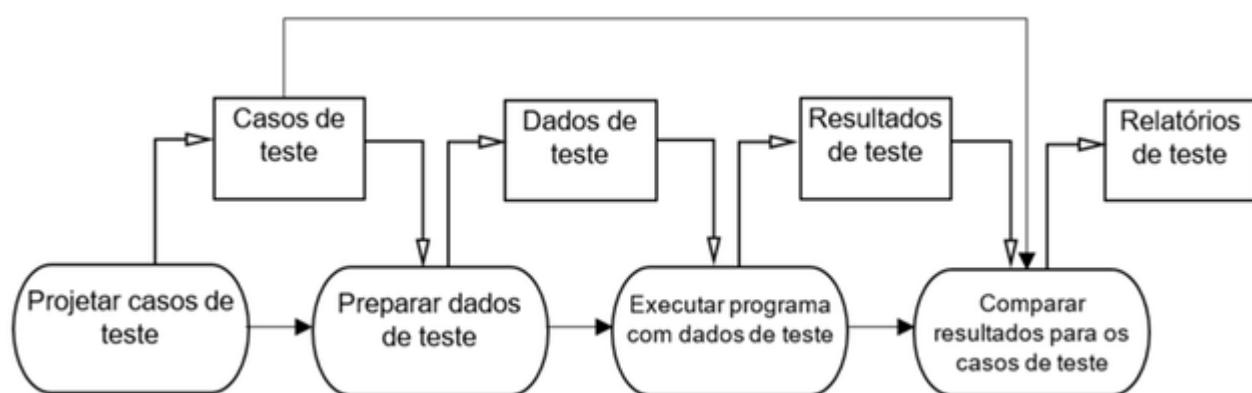


Figura 3 | Modelo de processo de teste. Fonte: Sommerville (2018).

Bem, aparentemente os casos de teste são, de fato, o elemento central no processo de teste.

**Siga em Frente...**

## Casos de Testes

Um caso de teste consiste em um conjunto de dados de entrada que pode ser fornecido ao programa, juntamente com a saída esperada que o programa deve produzir em conformidade com os requisitos estabelecidos previamente. Aqui, a entrada refere-se aos dados necessários para a execução do programa, enquanto a saída esperada representa o resultado dessa execução ou de uma função específica. Podemos esclarecer esses conceitos com um exemplo: suponha que você esteja testando um programa (ou função) projetado para validar datas inseridas pelo usuário. Um caso de teste típico seria composto pelo par (31/12/2020; válido). Ao fornecer a entrada 31/12/2020, a função de validação deveria retornar "data válida".

A boa escolha dos casos de teste é fator crítico para o sucesso da atividade. Um conjunto de casos de teste de baixa qualidade pode não exercitar partes críticas do programa e, em consequência disso, acabar não revelando defeitos no código. O que aconteceria, por exemplo, se o responsável pelos testes usasse apenas datas válidas como entradas? No mínimo, a parte do programa que trata das datas inválidas não seria executada, o que prejudicaria a confiabilidade do processo de teste e do produto testado.

Para exemplificar essa abrangência de datas, observe os casos de teste a seguir:

```
t1= {(15/2/1946; data válida), (30/2/2022; data inválida); (15/13/2023; data inválida);  
(29/2/2016; data válida), (29/2/2015; data inválida), (##/1/1985; data inválida)}.
```

Vamos analisar cada um dos casos de teste do conjunto t1:

- A entrada do primeiro caso de testes (15/2/1946) é formada por um dia válido (ou seja, contido no intervalo entre 1 e 31), um mês válido (contido no intervalo entre 1 e 12) e um ano válido. Essa entrada, que deverá provocar o retorno da mensagem "data válida" irá, portanto, exercitar trechos do programa criados para tratamento de dia, mês e ano simultaneamente válidos.
- A entrada 30/2/2022 também é formada por dia, mês e anos situados em intervalos válidos. No entanto, para o mês 2, o dia 30 é considerado inválido, pois o mês de fevereiro possui 28 ou 29 dias apenas. Assim, embora uma análise isolada de cada unidade da data possa indicar sua validade, a combinação do dia com o mês a torna inválida. O trecho de programa a ser exercitado, portanto, será diferente daquele exercitado no caso de teste anterior.
- Já a entrada 15/13/2023 também deverá retornar a mensagem "data inválida", pois o mês está fora do intervalo permitido, independentemente do ano e do dia escolhidos. Mais uma vez, o trecho do programa de validação de data a ser exercitado será distinto dos anteriores.
- Prosseguimos com as entradas 29/2/2016 e 29/2/2015. Elas retornam, respectivamente, "data válida" e "data inválida". Embora os dias e meses dessas datas sejam idênticos, 2016 foi ano bissexto, o que torna válida a primeira das datas. Já no caso de 2015, o dia 29 de

fevereiro não fez parte do calendário daquele ano, o que torna a data inválida. Novamente o fluxo do programa em questão seguirá trechos específicos para realizar as validações.

- Por fim, a última data de entrada deverá retornar a mensagem “data inválida” devido a um caractere não numérico na unidade do dia.

Embora exista um conjunto muito maior de entradas possíveis, esse conjunto de casos de teste será capaz de exercitar grande parte do código, o que aumentará a chance de descoberta de defeitos.

No entanto, será que os casos de teste se limitam apenas a esse aspecto? Vamos considerar outro exemplo: suponha que o cenário de teste seja a verificação da funcionalidade de login em um sistema de reserva de passagens para agentes de viagens. A avaliação é baseada na resposta do sistema a diferentes padrões de entrada de nome de usuário e senha. Selecionamos três cenários para ilustração:

1. Verificação da resposta do sistema quando o agente insere um nome de usuário e uma senha válidos.
2. Verificação da resposta do sistema quando o agente insere um nome de usuário e/ou uma senha inválidos.
3. Verificação da resposta do sistema quando o agente pressiona a tecla Enter com o campo de nome de usuário vazio.

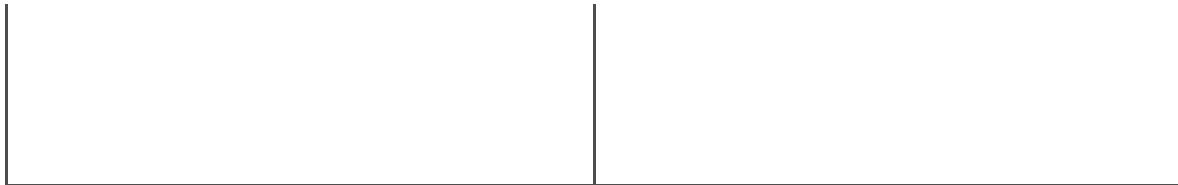
Para que possamos tornar mais específico nosso cenário, trataremos apenas do primeiro item e, com base nele, apresentamos o Quadro 1, que descreve um caso de teste relacionado.

#### Bloco 1

Cenário	Caso de teste	Passos de teste	Dados de entrada
Checagem da funcionalidade de login.	Checagem da resposta ao se inserir nome de usuário e senha válidos.	<ul style="list-style-type: none"> <li>- Executar a aplicação.</li> <li>- Informar o nome do agente.</li> <li>- Informar a senha.</li> <li>Acionar o botão "ok".</li> </ul>	Nome do agente: Marcio Senha: 5555

#### Bloco 2

Resultado esperado	Resultado obtido
O login deve ser bem-sucedido.	Login bem-sucedido.



Quadro 1 | Cenário detalhado do teste e um caso de teste possível. Fonte: Maitino (2021).

Assim, o caso de teste se torna mais detalhado, e todas as condições para sua verificação são especificadas de forma completa. Observa-se que até os passos para a realização do teste – que podem parecer óbvios – estão descritos no caso de teste. Fica evidente que o sucesso no procedimento de testes está diretamente relacionado à escolha adequada e ao uso correto dos casos de teste. Idealmente, cada conjunto de casos de teste deve estar associado a um requisito significativo a ser testado. Para evitar definições incorretas, é essencial planejamento e um bom entendimento da aplicação. Uma abordagem eficaz para resolver esse problema, é definir o ambiente no qual o teste será realizado, estabelecer as entradas do caso de teste, determinar a saída esperada para cada entrada e definir os passos a serem executados para realizar os testes.

Quando um caso de teste é executado, o resultado deve ser registrado. Existem várias abordagens para definir o resultado da aplicação de um caso de teste específico. A mais comum inclui as seguintes opções:

- **Passou:** todos os passos do caso de teste foram concluídos com sucesso para todas as entradas.
- **Falhou:** nem todos os passos foram concluídos com sucesso para uma ou mais entradas.
- **Bloqueado:** o teste não pôde ser executado devido à impossibilidade de configurar o ambiente.

## Vamos Exercitar?

Para a atividade de montagem de caso de teste individual para a validação de data em um formulário, aqui está um gabarito sugerido com alguns cenários de teste baseados no tipo de dados de teste fornecido:

### Gabarito: Casos de Teste para Validação de Data de Nascimento

#### Caso de Teste 1: Data Válida Não Bissexto

- **Nome do Caso de Teste:** Validar Data de Nascimento Comum Não Bissexto
- **Precondições:** O formulário de inscrição está carregado e o campo de data de nascimento está vazio.
- **Passos:**
  1. Clique no campo de data de nascimento.

2. Insira a data "15/2/1946".

3. Envie o formulário.

- **Dados de Teste:** 15/2/1946.

- **Resultado Esperado:** o sistema deve aceitar a data como válida e permitir a submissão do formulário.

- **Critérios para Falha:** o sistema rejeita a data ou impede a submissão do formulário.

## Caso de Teste 2: Data Inválida com Dia Inexistente

- **Nome do Caso de Teste:** validar Data de Nascimento com Dia Inexistente

- **Precondições:** o formulário de inscrição está carregado e o campo de data de nascimento está vazio.

- **Passos:**

1. Clique no campo de data de nascimento.

2. Insira a data "30/2/2022".

3. Envie o formulário.

- **Dados de Teste:** 30/2/2022.

- **Resultado Esperado:** o sistema deve rejeitar a data como inválida e impedir a submissão do formulário.

- **Critérios para Falha:** o sistema aceita a data e permite a submissão do formulário.

## Caso de Teste 3: Data Inválida com Mês Inexistente

- **Nome do Caso de Teste:** validar Data de Nascimento com Mês Inexistente

- **Precondições:** o formulário de inscrição está carregado e o campo de data de nascimento está vazio.

- **Passos:**

1. Clique no campo de data de nascimento.

2. Insira a data "15/13/2023".

3. Envie o formulário.

- **Dados de Teste:** 15/13/2023.

- **Resultado Esperado:** o sistema deve rejeitar a data como inválida e impedir a submissão do formulário.

- **Critérios para Falha:** o sistema aceita a data e permite a submissão do formulário.

## Caso de Teste 4: Data Válida Bissexta

- **Nome do Caso de Teste:** validar Data de Nascimento em Ano Bissesto

- **Precondições:** o formulário de inscrição está carregado e o campo de data de nascimento está vazio.

- **Passos:**

1. Clique no campo de data de nascimento.
2. Insira a data "29/2/2016".
3. Envie o formulário.

- **Dados de Teste:** 29/2/2016.
- **Resultado Esperado:** o sistema deve aceitar a data como válida e permitir a submissão do formulário.
- **Critérios para Falha:** o sistema rejeita a data ou impede a submissão do formulário.

## Caso de Teste 5: Data Inválida Não Bissexto

- **Nome do Caso de Teste:** validar Data de Nascimento Não Bissexto Inexistente
- **Precondições:** o formulário de inscrição está carregado e o campo de data de nascimento está vazio.
- **Passos:**

1. Clique no campo de data de nascimento.
2. Insira a data "29/2/2015".
3. Envie o formulário.

- **Dados de Teste:** 29/2/2015.
- **Resultado Esperado:** o sistema deve rejeitar a data como inválida e impedir a submissão do formulário.
- **Critérios para Falha:** o sistema aceita a data e permite a submissão do formulário.

Você pode elaborar mais casos de testes ou até mesmo ter casos de testes diferentes dos apresentados. O importante é lembrar de como criar casos de testes relevantes para o sistema.

## Saiba mais

Para ler mais sobre o Teste de Software, acesse o livro [Engenharia de Software - Pressman](#), Capítulo 19.

O livro de testes do Delamaro, [Introdução ao Teste de Software](#), oferece uma abordagem abrangente e prática para dominar as técnicas essenciais de verificação e validação de software.

Os casos de teste são ferramentas fundamentais para garantir a qualidade e confiabilidade de um software, fornecendo um roteiro preciso para avaliar seu funcionamento em diversas situações. Leia mais no artigo indicado: [Casos de Teste: Entenda a importância e porque são fundamentais para a área de QA](#).

## Referências

IEEE Computer Society. **Guide to the Software Engineering Body of Knowledge**. Piscataway: The Institute of Electrical and Electronic Engineers, 2014.

MAITINO NETO, R. **Engenharia de software**. Londrina: Editora e Distribuidora Educacional S.A., 2021.

MEDEIROS, M. P. **JUnit Tutorial**. DevMedia. [S./], 2009. Disponível em: <https://www.devmedia.com.br/junit-tutorial/1432>. Acesso em: 22 abr. 2024.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 9. ed. Porto Alegre: AMGH, 2021.

SCHACH, S. R. **Engenharia de software: os paradigmas clássicos e orientados a objetos**. 7. ed. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Aula 2

Estratégias de testes

### Estratégias de testes

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você conhecerá as Estratégias de Testes, incluindo Teste de Caixa Branca e Teste de Caixa Preta. Esses conteúdos são cruciais para sua prática profissional, capacitando-o a identificar e corrigir falhas de maneira eficiente. Compreender as diferentes estratégias de teste permite uma abordagem abrangente na verificação da funcionalidade e

eficácia do software desenvolvido. Prepare-se para adquirir habilidades valiosas que impulsionarão sua carreira como profissional de testes de software.

## Ponto de Partida

Após nosso primeiro contato com o assunto, é hora de avançarmos para temas mais específicos relacionados às técnicas de teste. Em termos simples, uma técnica representa uma abordagem para realizar testes e, durante nosso estudo, você compreenderá por que existem diversas maneiras de conduzir os testes. A aplicação prática das técnicas ajudará a entender o propósito de cada uma delas. Após esta introdução, abordaremos os testes para aplicações móveis, web e orientadas a objetos para esclarecer suas especificidades.

Durante nosso estudo da técnica funcional, descreveremos alguns tipos de testes e desenvolveremos um exemplo prático de teste de componente. No teste estrutural, detalharemos a transformação de um código-fonte simples em um grafo, um elemento gráfico que o representará e permitirá a realização de uma simulação de cobertura de testes. Nas seções sobre testes para aplicações móveis, web e orientadas a objetos, forneceremos detalhes específicos dessas aplicações, que orientarão a condução dos procedimentos de teste. Com isso, esperamos capacitá-lo a tomar decisões sobre as técnicas e tipos de teste a serem aplicados em diferentes situações.

Com o objetivo de expandir suas atividades no desenvolvimento e teste de sistemas, os gestores de uma empresa de software decidiram oferecer serviços de teste para terceiros. Dessa forma, desenvolvedores interessados em terceirizar os testes de seus produtos poderiam recorrer a essa empresa, e os testes seriam realizados mediante contrato e remuneração financeira.

Na condição de um dos melhores profissionais dessa empresa, você foi destacado para assumir o primeiro projeto de teste e logo se deparou com um obstáculo: por força de cláusula contratual, você não teria acesso ao código-fonte daquela aplicação, que fora projetada para ser executada em computadores pessoais conectados em rede local.

O desenvolvedor que confiou os testes à empresa em que você atua mostrou-se absolutamente disponível para fornecer todos os elementos do sistema a você, exceto o código-fonte. Considerando esse cenário, você foi chamado a planejar o procedimento de teste e, com a finalidade de direcionar seus esforços, seu gestor apontou quatro itens que deveriam obrigatoriamente compor o planejamento:

- O documento que deverá ser solicitado à organização desenvolvedora: considere que as funções a serem testadas devem estar descritas em algum documento e é exatamente este que você deverá solicitar ao seu cliente.
- A técnica de teste que deverá ser utilizada: das técnicas abordadas, você deve escolher uma delas e explicar o motivo da escolha.
- Critério que utilizará para a escolha dos casos de teste: sua descrição neste tópico deve se basear na melhor escolha de casos de teste para o perfil do sistema em teste.

- Definição de quem deverá participar dos procedimentos de teste: especifique neste item quem deverá participar dos testes, considerando o conhecimento do sistema que um ou outro elemento pode possuir.

Bons estudos!

## Vamos Começar!

### Estratégias de Testes

Um produto de software necessita de inúmeros elementos para oferecer ao seu usuário um funcionamento pleno. A adequação do hardware e da infraestrutura de rede são apenas dois desses elementos, e o dimensionamento incorreto de um deles terá o potencial de arruinar a experiência do usuário. Há, no entanto, um aspecto que sempre terá papel decisivo na percepção do usuário em relação ao sistema que utiliza: as suas funções. Afinal, será por meio delas que ele interagirá com o programa e alcançará seus objetivos traçados lá na fase de requisitos. Funções mal projetadas ou defeituosas podem, inclusive, desestimular o uso do sistema. Com tamanha importância, a área de testes não poderia deixar de direcionar muita atenção às funções do sistema e o faz por meio da aplicação da Técnica de Teste Funcional e do Teste de Funcionalidades, assuntos que desenvolveremos na sequência.

No entanto, antes de nos aprofundarmos nas técnicas específicas de teste, é importante situá-las dentro do contexto da estratégia de teste. Essa estratégia corresponde a uma prática cujo objetivo é estabelecer um plano detalhado descrevendo os passos a serem executados no processo de teste, ou seja, um modelo para conduzir os testes. Pressman (2021) fornece uma visão genérica dos elementos que estão presentes em todas as estratégias adotadas para os testes:

- **Revisões de Software:** revisões eficazes eliminam problemas no código antes da aplicação efetiva dos testes.
- **Progressão do Teste:** os testes devem iniciar em unidades individuais do software e progredir em direção à integração do sistema como um todo.
- **Depuração:** embora seja uma atividade distinta, a depuração deve estar associada ao processo de teste.
- **Técnicas:** diferentes técnicas de teste são adequadas para diferentes tipos de sistemas e são aplicadas conforme os recursos disponíveis à equipe de teste.

O último componente da estratégia de teste embasa nosso próximo assunto. Para compreendermos as razões por trás de uma técnica, é útil fazer uma breve digressão: os testes são conduzidos com objetivos específicos em mente e são planejados para verificar diversas propriedades de um sistema. Dependendo da técnica selecionada, os casos de teste podem ser elaborados para verificar se as especificações funcionais estão implementadas corretamente, ou então são escolhidos para avaliar o desempenho, a confiabilidade e a usabilidade do sistema,

entre outras propriedades. Dessa forma, abordagens distintas de teste justificam a aplicação de técnicas igualmente diversas (Pressman, 2021).

Embora as técnicas de teste sejam altamente relevantes por estarem mais diretamente associadas às metodologias de aplicação de testes, esta não é a única dimensão que podemos identificar neste contexto. Na verdade, podemos estabelecer outros tipos de relações entre um teste e seu objetivo simplesmente fixando o momento, o objeto e, como já mencionado, a maneira (ou metodologia) de aplicação.

Parece complicado? Observe a Figura 1: ela posiciona os vários tipos de teste nas três dimensões que acabamos de mencionar: o “quando”, o “que” e o “como”.

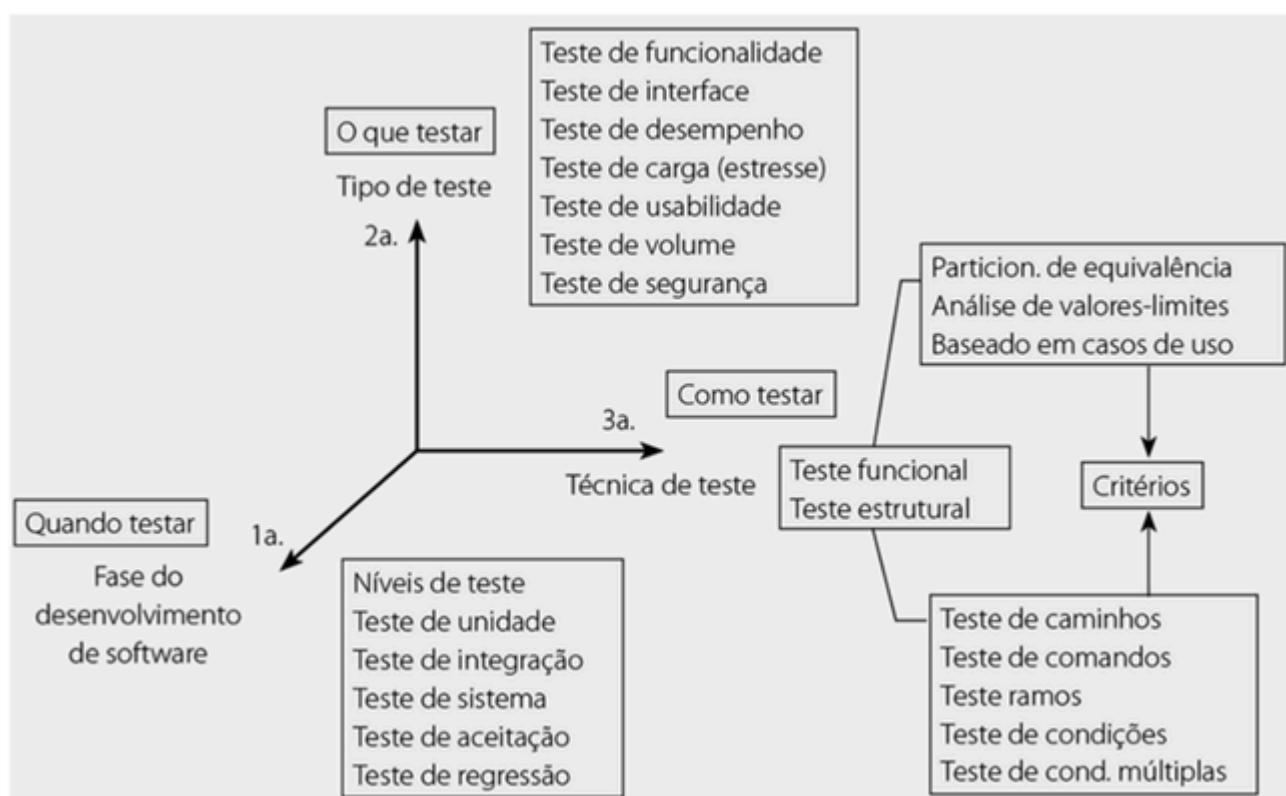


Figura 1 | As três dimensões dos testes. Fonte: Braga (2016).

Apesar de não termos tido contato com os testes apontados na figura, é possível destacar que o teste funcional está relacionado a uma maneira de se realizar um teste, daí ter sido posicionado na dimensão “Como testar”. Já o teste de funcionalidade – que também será abordado na sequência – guarda associação com “o que testar”. Feita essa contextualização, passamos à abordagem inicial da técnica de teste funcional e do teste de funcionalidade.

## Técnica de Teste Funcional

Essa abordagem se fundamenta nas especificações do software para extrair os requisitos de teste. O teste é conduzido nas funcionalidades do programa, daí o termo "funcional". Seu objetivo não é examinar os processamentos internos, mas sim verificar se o algoritmo inserido produz os resultados esperados (Bartié, 2002).

Uma das vantagens dessa estratégia de teste é que não demanda conhecimento detalhado sobre a implementação do programa. Nem mesmo o acesso ao código-fonte é necessário. Observe uma representação dessa técnica na Figura 2:

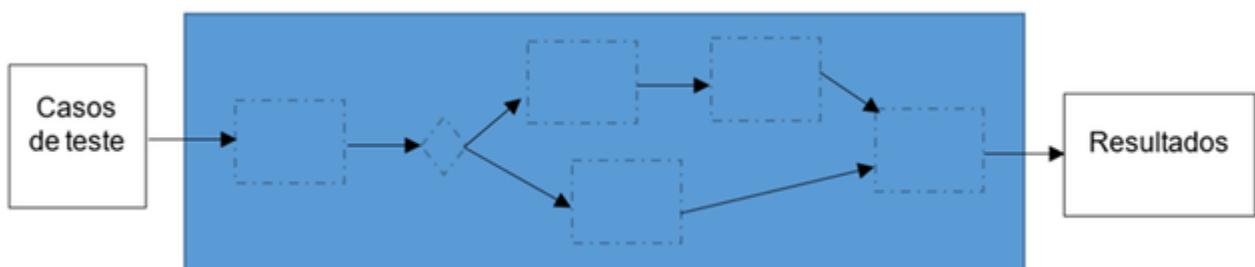


Figura 2 | Visão do teste de caixa preta. Fonte: Bartié (2002).

A ilustração na figura nos leva a compreender que o testador não possui conhecimento dos detalhes internos do sistema e baseia sua avaliação apenas nos resultados obtidos para cada entrada fornecida. Nesse sentido, Pressman (2021) afirma que um produto de software pode ser testado desde que o responsável conheça a função específica para a qual o software foi projetado e, com esse entendimento, possa realizar testes que demonstrem a operacionalidade de cada uma das funções, embora o objetivo final do teste seja encontrar defeitos no produto. Os autores concluem que essa abordagem de teste adota uma perspectiva externa do produto e não se concentra na lógica interna do software, a qual permanece opaca ao testador. Por esse motivo, a técnica funcional também é conhecida como teste de caixa preta.

O planejamento do teste funcional comprehende dois passos principais: a identificação das funções que o software deve executar (por meio da especificação dos requisitos) e a criação de casos de teste para verificar se essas funções estão sendo desempenhadas corretamente. Apesar da simplicidade dessa técnica e de sua aplicabilidade a todos os programas cujas funções são conhecidas, é importante considerar uma dificuldade inerente: não é possível garantir que partes essenciais ou críticas do software serão testadas, mesmo com um conjunto robusto de casos de teste.

Um teste funcional não deve ser aplicado simultaneamente a todas as funções do sistema ou em uma única ocasião. Em vez disso, ele deve examinar elementos específicos em ocasiões predefinidas, o que levou à divisão desse tipo de teste em subtipos. Para elucidar essa circunstância, alguns exemplos são úteis: quando aplicado para verificar as funções de um módulo, função ou classe do sistema, ele é chamado de teste de unidade. O teste de integração, por outro lado, é conduzido para avaliar como as unidades funcionam em conjunto ou integradas, enquanto o teste de regressão, um subtipo do teste funcional, tem como objetivo garantir que uma alteração feita em uma função não tenha introduzido outros problemas no código (Pressman, 2021).

A menção ao quarto subtipo de teste funcional nos dará a oportunidade para o desenvolvimento de um exemplo prático. Quando testamos um componente do sistema de modo independente para verificar sua saída esperada, chamamos tal procedimento de teste de componentes. Geralmente, ele é executado para verificar a funcionalidade e/ou usabilidade de componentes, mas não se restringe apenas a eles. Um componente pode ser qualquer coisa que receba entradas e forneça alguma saída, incluindo uma página web, telas e até mesmo um sistema dentro de um sistema maior.

Como aplicação prática desse teste, imaginemos um sistema de vendas pela internet. Em uma análise mais minuciosa, seria possível identificar muitos componentes nessa aplicação, mas escolheremos apenas alguns, os quais estão representados na Figura 3.

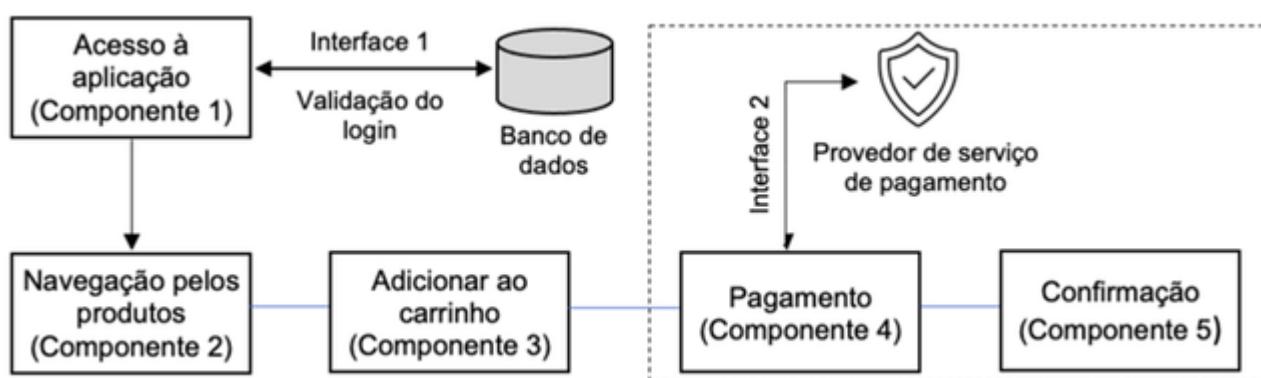


Figura 3 | Componentes de um sistema de vendas pela internet. Fonte: Maitino (2021).

O componente de login (identificado como componente 1) efetiva a validação de acesso do usuário ao sistema, usando, para isso, uma interface específica, que realiza a conexão com o banco de dados. Esse componente dá acesso à navegação pelos produtos, à inclusão de itens no carrinho, ao pagamento da compra e à sua confirmação. Para efetivação do pagamento, há uma interface que conecta o sistema em teste a um provedor de serviço de pagamentos. Considerando esse cenário, vejamos o que deve ser testado em específico no componente 1 (login) (Maitino, 2021):

- A interface de usuário nos itens de usabilidade e acessibilidade.
- O carregamento da página, como forma de garantir bom desempenho da aplicação.
- A suscetibilidade a ataques ao banco de dados por meio de elementos da interface de usuário.
- A resposta da funcionalidade de login por meio do uso de credenciais falsas e válidas.

Se considerarmos o componente 3 (adição do produto ao carrinho), o testador deverá verificar o que ocorre, por exemplo, quando o usuário coloca um item, o qual supostamente deseja comprar, e fecha aplicação em seguida. Um cuidado especial também deverá ser tomado com a comunicação entre o componente 4 e o provedor de serviço de pagamento.

As características do teste de componente nos preparam para o próximo tipo de teste. Se a técnica de teste funcional se concentra nas funcionalidades do sistema, qual seria então a diferença em relação ao teste de funcionalidade? Os testes de funcionalidade priorizam as interações com o usuário e a navegação no sistema, sendo conduzidos para verificar se um aplicativo de software opera corretamente de acordo com as especificações do projeto, especialmente no que se refere à entrada e validação de dados, às operações de menu e todas as interações do usuário com as interfaces. Além das verificações mencionadas, um teste de funcionalidade também deve abordar a funcionalidade de "copiar e colar" e a conformidade dos ajustes de padrões regionais com a localidade na qual o software está sendo utilizado.

Dentro do contexto da qualidade de software, o termo "funcionalidade" não se limita apenas às características do sistema que são objetos de teste. Há também uma abordagem que considera a funcionalidade como um atributo fundamental da qualidade. Nesse segundo caso, refere-se ao grau em que o software atende às necessidades declaradas pelo cliente, conforme indicado pelos subatributos de adequação, exatidão, interoperabilidade, conformidade e segurança (Pressman, 2021). Em outras palavras, a funcionalidade proporcionada pelo sistema não deve ser confundida com o nível em que um programa satisfaz requisitos de adequação ao propósito, precisão e facilidade de operação com outros sistemas.

## Siga em Frente...

## Técnica de Teste Estrutural

Se voltarmos um pouco no texto e observarmos novamente a Figura 1, notaremos que o teste estrutural está situado na mesma categoria do teste funcional. Esta categoria, denominada "Como testar", engloba técnicas (ou abordagens) para a execução de testes. Os testes estruturais (também conhecidos como caixa branca) são assim denominados porque se baseiam na arquitetura interna do programa. Com acesso ao código-fonte e à estrutura do banco de dados, o testador pode submeter o programa a uma ferramenta automatizada de teste. É importante mencionar que um teste funcional também pode (e deve) ser realizado de forma automatizada.

Existem várias ferramentas capazes de conduzir testes estruturais, porém, não será selecionada nenhuma em particular para os propósitos de detalhamento deste teste. Em vez disso, vamos nos apoiar em uma possível forma de representação do código do programa para elaborar um método de teste. Suponhamos que, ao analisar o código do programa, nossa ferramenta construa uma representação conhecida como grafo. Conforme Delamaro (2004), em um grafo, os nós correspondem a blocos indivisíveis de código, o que significa que não há desvio de fluxo do programa dentro do bloco e, uma vez que o primeiro comando é executado, os demais comandos são executados sequencialmente. Outro componente presente em um grafo são as arestas (ou os arcos), que representam o fluxo entre os nós. Se considerarmos o código de uma aplicação que calcula o fatorial de um valor, mostrado na Figura 4, podemos identificar que o trecho entre a linha 1 e a linha 8 atende aos requisitos para se transformar em um nó do grafo.

```
1 #include <stdio.h>
2 main()
3 {
4     int i = 0;
5     valor = 0;
6     fatorial = 1;
7     printf("Programa que calcula o fatorial de um valor informado pelo usuario\n");
8
9     do {
10         printf("\nInforme um valor entre 1 e 10: ");
11         scanf("%d",&valor);
12         } while ((valor<1) || (valor>10));
13         for (i=1; i<=valor; i++)
14             fatorial=fatorial*i;
15         printf("\nO Fatorial de %d = %d", valor, fatorial);
16         printf("\n");
17 }
```

Figura 4 | Programa que calcula o fatorial de um número fornecido. Fonte: Maitino (2021).

O programa que usaremos para ilustrar um teste estrutural será aquele cujo código pode ser visto na Figura 5. Sua função é a de verificar a validade de um nome de identificador fornecido com base nas seguintes regras:

- Tamanho t do identificador entre 1 e 6 ( $1 \leq t \leq 6$ ): condição válida.
- Tamanho t do identificador maior que 6 ( $t > 6$ ): condição inválida.
- Primeiro caractere c é uma letra: condição válida.
- Primeiro caractere c não é uma letra: condição inválida.
- O identificador possui apenas caracteres válidos: condição válida.
- O identificador possui um ou mais caracteres inválidos: condição inválida.

```
/* 01 */ {
/* 01 */ char achar;
/* 01 */ int length, valid_id;
/* 01 */ lenght = 0;
/* 01 */ printf ("Identificador");
/* 01 */ achar = fgetc(stdin);
/* 01 */ valid_id = valid_s(achar);
/* 01 */ if (valid_id)
/* 02 */     lenght = 1;
/* 03 */ achar = fgetc (stdin);
/* 04 */ while (achar != '\n')
/* 05 */ {
/* 05 */     if (!(valid_f(achar)))
/* 06 */         valid_id = 0;
/* 07 */     lenght++;
/* 07 */     achar = fgetc (stdin);
/* 07 */ }
/* 08 */ if (valid_id && (length >= 1) && (lenght < 6))
/* 09 */     printf ("Valido\n");
/* 10 */ else
/* 10 */     printf ("Invalido\n");
/* 11 */ }
```

Figura 5 | Código em C que apura classes válidas e inválidas de identificadores. Fonte: Delamaro (2004).

Quando analisado pela ferramenta de teste, o Código apresentado na Figura 5, será transformado no grafo da Figura 6. Note que cada trecho identificado com um número no código é representado em um nó no grafo gerado.

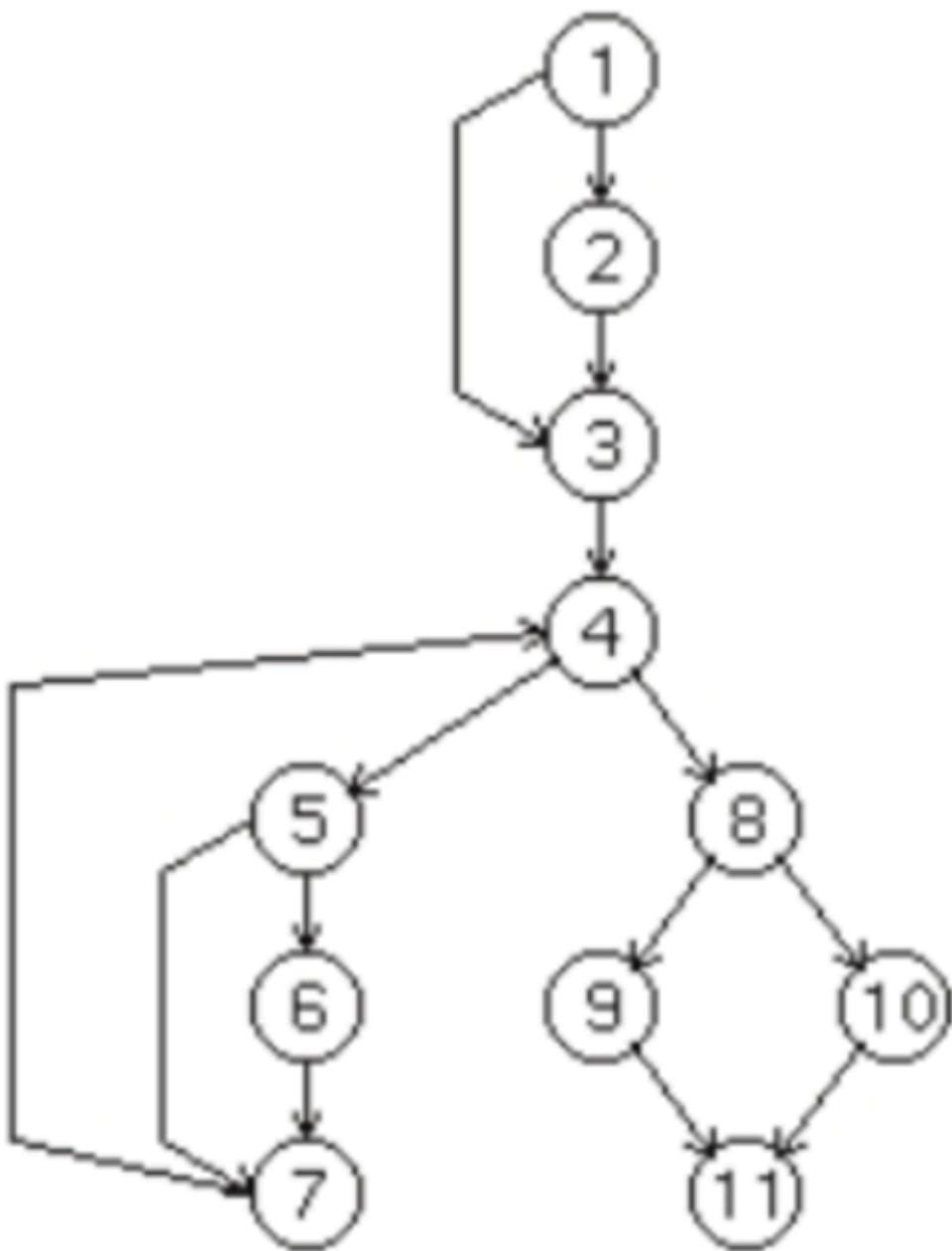


Figura 6 | Grafo gerado pela ferramenta de teste estrutural. Fonte: Delamaro (2004).

Ao testador cabe selecionar casos de teste capazes de percorrer os nós, os arcos e os caminhos representados no grafo do programa. Apenas para fins de exemplificação, um caminho que pode ser percorrido pelo fluxo do programa é o formado pela sequência de nós 2,3,4,5,6 e 7. Casos de testes diferentes tenderão a exercitar sequências diferentes do grafo e, se considerarmos a existência de um conjunto infinito desses casos, teremos que as escolhas incorretas podem arruinar o teste. Ainda, se considerarmos que os casos de teste são potencialmente infinitos, alguns critérios de cobertura do código devem ser previamente definidos.

## Vamos Exercitar?

Um procedimento de teste que não é precedido por um planejamento criterioso pode perder o rumo ao longo do caminho e não ter uma conclusão satisfatória. Nesse sentido, esta situação-problema procura conduzi-lo à criação de um planejamento coerente com as possibilidades e as restrições impostas pelo cenário. Apenas para resgatarmos o contexto em que a narrativa se dá, você foi designado para testar um sistema convencional (ou seja, não web e não móvel) e, de antemão, sabe que não contará com o código-fonte dele.

Como um documento de planejamento sempre refletirá características, estilo e outras subjetividades de quem o criou, o que forneceremos aqui serão as linhas gerais que nele deverão estar presentes. A primeira providência será a de prever o acesso ao documento em que os requisitos do sistema estão especificados. Será por meio dele que você conhecerá as funções do sistema e poderá criar casos de testes específicos para cada uma. Lembre-se: não há código-fonte disponível e, portanto, a técnica de testes mais imediata para aplicação é a funcional.

O efeito da ausência do código-fonte responde ao segundo item do planejamento. Como já mencionado, a técnica a ser das funções derivadas dos requisitos, o testador poderá aplicar testes em cada uma delas. Esta circunstância utilizada é a funcional e, por meio dela nos apresenta então outro desdobramento: as características dos casos de teste. Eles deverão ser selecionados de modo que consigam reproduzir o uso corriqueiro das funções, além de serem capazes de verificar itens de usabilidade das interfaces de usuário.

Por fim, o procedimento de teste deve ser acompanhado por ao menos um desenvolvedor do software. Como esse profissional provavelmente atua na organização desenvolvedora, o planejamento deverá prever ocasiões em que ele deverá se deslocar ao local em que os testes serão feitos. A presença do desenvolvedor garantirá que um conhecedor dos detalhes do sistema atuará durante os procedimentos de teste.

O que segue é uma solução viável para este desafio:

### Documento de planejamento inicial de teste

- **Objetivo:** este documento tem o objetivo de descrever quatro elementos do planejamento de um teste, considerando o cenário que será resgatado em seu corpo.
- **Documento de requisitos de software:** a fim de que a equipe conheça as funções do produto a ser testado, deverá ser solicitado ao desenvolvedor (também chamado cliente) o documento de requisitos do software.
- **Restrição:** nenhuma restrição se aplica a este item.
- **Técnica de teste:** considerando a ausência do código-fonte, a técnica a ser utilizada é a funcional. Cada função do produto deve ser conhecida e analisada por meio do documento que agrupa seus requisitos.
- **Casos de teste:** os casos de teste devem ser selecionados de modo que possam exercitar, da forma mais completa possível todas as funções do produto. Eles devem conseguir

reproduzir o uso corriqueiro das funções e verificar as condições de usabilidade das interfaces de usuário.

- **Participantes do teste:** será requisitada a participação de ao menos um desenvolvedor do produto no procedimento de teste, em intervalos regulares ou sempre que for necessária sua intervenção. A presença desse desenvolvedor garantirá que um conhecedor dos detalhes do sistema esteja orientando o procedimento de teste.

## Saiba mais

O livro de Pressman oferece insights valiosos sobre a importância dos testes de software, destacando técnicas eficazes para garantir a qualidade e robustez dos sistemas desenvolvidos, acesse o Capítulo 19 do livro indicado [Engenharia de Software - Pressman](#).

Os testes funcionais e estruturais são duas abordagens essenciais para garantir a qualidade e robustez do software. Enquanto os testes funcionais avaliam o comportamento do sistema com base nas especificações e requisitos funcionais, os testes estruturais examinam a estrutura interna do código-fonte, identificando possíveis falhas e vulnerabilidades. Para ler mais sobre estes tipos de testes, leia os Capítulos 1 e 2 [Introdução ao Teste de Software](#).

Para saber mais sobre os testes de desenvolvimento, leia o Capítulo 8 de [Engenharia de Software - Sommerville](#).

## Referências

BARTIÉ, A. **Garantia da qualidade de software:** as melhores práticas de Engenharia de Software aplicadas à sua empresa. Rio de Janeiro: Elsevier, 2002.

BRAGA, P. H. **Testes de software.** São Paulo: Pearson Educational do Brasil, 2016.

DELAMARO, M. E. **Introdução ao teste de software.** Rio de Janeiro: Elsevier, 2004.

MAITINO NETO, R. **Engenharia de software.** Londrina: Editora e Distribuidora Educacional S.A., 2021.

PRESSMAN, R. S. **Engenharia de software:** uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

## Aula 3

Tipos de testes

## Tipos de testes



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você explorará os fundamentos dos testes de software, e estudaremos as técnicas essenciais, incluindo Teste do Caminho Básico, Testes de Estrutura de Controle e Testes Baseados em Modelos. Esses conteúdos são cruciais para a prática profissional, capacitando os alunos a identificar, avaliar e corrigir falhas de maneira eficiente, garantindo a qualidade e confiabilidade dos sistemas desenvolvidos. Prepare-se para adquirir habilidades valiosas para sua carreira na área de desenvolvimento de software.

## Ponto de Partida

Nesta aula, serão apresentadas três importantes técnicas de teste de software: Teste do Caminho Básico, Testes de Estrutura de Controle e Testes Baseados em Modelos. O Teste do Caminho Básico consiste em identificar e testar todos os caminhos possíveis em um programa, garantindo uma cobertura abrangente e minimizando a probabilidade de erros não detectados. Por sua vez, os Testes de Estrutura de Controle focam na avaliação da lógica de controle do programa, verificando as condições, os loops e as ramificações para garantir seu funcionamento adequado. Já os Testes Baseados em Modelos envolvem a criação de modelos que representam o comportamento esperado do sistema, permitindo simular diferentes cenários e identificar possíveis falhas.

Essas técnicas são essenciais para garantir a qualidade e a confiabilidade do software. Ao aplicar o Teste do Caminho Básico, os desenvolvedores podem identificar e corrigir falhas em diferentes partes do código, garantindo que todas as funcionalidades se comportem como esperado. Os Testes de Estrutura de Controle permitem verificar se as decisões lógicas do programa estão sendo tomadas corretamente, enquanto os Testes Baseados em Modelos ajudam a prever o comportamento do sistema em diferentes situações. Assim, ao dominar essas técnicas, os alunos estarão mais preparados para desenvolver software de qualidade e com menor incidência de bugs.

Para exercitar os testes de caminho básico, análise o trecho de código a seguir:

```
public boolean saque(double valor){  
    boolean status = false;  
    if(valor <= saldo){  
        saldo = saldo - valor;  
        status=true;  
    }  
  
    else{  
        if(valor <= limiteCredito){  
            limiteCredito = limiteCredito - valor;  
            status = true;  
        }  
    }  
  
    return status;  
}
```

Figura 1 | Código

A partir deste código, crie um grafo de fluxo de programa para o código anterior.

Bons estudos!

## Vamos Começar!

## Teste do Caminho Básico

O teste de caminho básico é uma técnica de teste de caixa-branca. Essa abordagem permite que o projetista de casos de teste derive uma medida da complexidade lógica de um projeto procedural e utilize essa medida como guia para definir um conjunto-base de caminhos de execução. Os casos de teste desenvolvidos para exercitar esse conjunto-base garantem que todas as instruções de um programa sejam executadas pelo menos uma vez durante o teste.

Antes de adentrarmos no método do caminho básico, é importante introduzir uma notação simples para representar o fluxo de controle, denominada grafo de fluxo (ou grafo de programa). Esse tipo de representação deve ser utilizado apenas quando a estrutura lógica de um

componente for complexa. O grafo de fluxo facilita o acompanhamento dos caminhos de um programa, tornando mais simples a compreensão de sua lógica (Pressman, 2021).

Para exemplificar o uso de um grafo de fluxo, consideremos a representação do projeto procedural apresentada na Figura 2a. Nessa representação, utilizamos um fluxograma para demonstrar a estrutura de controle do programa. Posteriormente, na Figura 2b, esse fluxograma é mapeado em um grafo de fluxo correspondente, levando em conta que os losangos de decisão do fluxograma não possuem condições compostas.

Na Figura 2b, cada círculo, designado como nó do grafo de fluxo, representa um ou mais comandos procedurais. Uma sequência de retângulos de processamento e um losango de decisão podem ser mapeados em um único nó. As setas presentes no grafo de fluxo, conhecidas como arestas ou ligações, representam o fluxo de controle e são análogas às setas presentes no fluxograma. É importante observar que uma aresta deve terminar em um nó, mesmo que este não represente nenhum comando procedural específico (por exemplo, o símbolo do diagrama de fluxo para a construção se-então-senão [*if-then-else*]). As áreas delimitadas pelas arestas e pelos nós são denominadas regiões. Ao contabilizarmos as regiões, incluímos a área fora do grafo como uma região adicional.

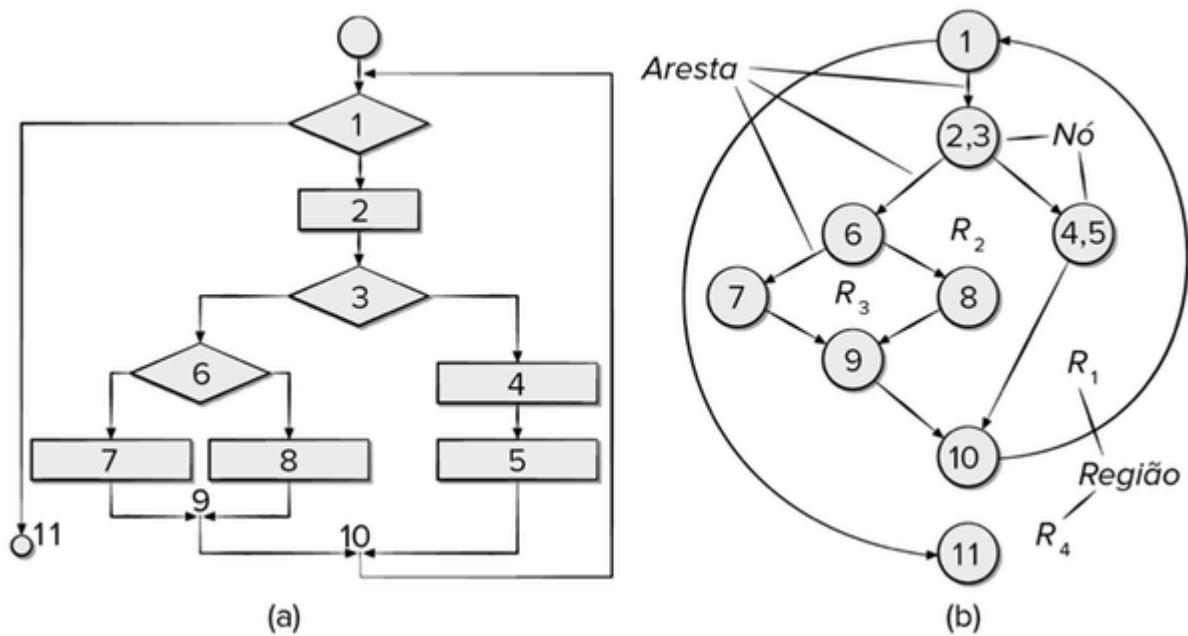


Figura 2 | (a) Fluxograma e (b) grafo de fluxo. Fonte: Pressman (2021).

Um caminho independente refere-se a qualquer trajeto através do programa que introduza pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando expresso em termos de um grafo de fluxo, um caminho independente deve incorporar pelo menos uma aresta que não tenha sido percorrida anteriormente antes de definir o caminho. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo ilustrado na Figura 2b seria:

Caminho 1: 1-11

Caminho 2: 1-2-3-4-5-10-1-11

Caminho 3: 1-2-3-6-8-9-10-1-11

Caminho 4: 1-2-3-6-7-9-10-1-11

Note que cada novo caminho introduz uma nova aresta. O caminho

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 não é considerado um caminho independente porque é simplesmente uma combinação dos caminhos já especificados e não atravessa nenhuma nova aresta.

Os caminhos de 1 a 4 constituem um conjunto básico para o grafo de fluxo apresentado na Figura 2b. Em outras palavras, se testes forem planejados para cobrir a execução desses caminhos (conjunto básico), cada comando do programa será executado pelo menos uma vez, e cada condição será testada tanto em sua condição verdadeira quanto falsa. É importante observar que o conjunto básico não é único. Na verdade, diferentes conjuntos básicos podem ser derivados para um mesmo projeto procedural.

Mas como determinamos quantos caminhos devemos buscar? A resposta vem do cálculo da complexidade ciclomática. A complexidade ciclomática é uma métrica de software que oferece uma medida quantitativa da complexidade lógica de um programa. Quando utilizada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto básico de um programa, estabelecendo um limite superior para a quantidade de testes que devem ser realizados para garantir que todos os comandos sejam executados pelo menos uma vez.

A complexidade ciclomática é fundamentada na teoria dos grafos e representa uma métrica essencial para avaliação de software. Existem três formas de calcular a complexidade ciclomática (Pressman, 2021):

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.

2. A complexidade ciclomática  $V(G)$  para um grafo de fluxo  $G$  é definida como:

$$V(G) = E - N + 2$$

onde  $E$  é o número de arestas do grafo de fluxo e  $N$  é o número de nós do grafo de fluxo.

3. A complexidade ciclomática  $V(G)$  para um grafo de fluxo  $G$  é definida como:

$$V(G) = P + 1$$

onde  $P$  é o número de nós predicação contidos no grafo de fluxo  $G$ .

Examinando mais uma vez o diagrama de fluxo da Figura 2b, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos citados anteriormente (Pressman, 2021):

O grafo de fluxo tem quatro regiões.

$$V(G) = 11 \text{ arestas} - 9 \text{ nós} + 2 = 4.$$

$$V(G) = 3 \text{ nós predicados} + 1 = 4.$$

Portanto, a complexidade ciclomática para o grafo de fluxo da Figura 2b é 4.

O valor de  $V(G)$  é crucial, pois fornece um limite superior para o número de caminhos independentes que constituem o conjunto-base de testes. Isso implica um limite superior para o número de testes necessários para garantir que todos os comandos do programa sejam exercitados. Neste caso específico, seriam necessários no máximo quatro casos de teste para abranger cada caminho lógico independente.

## Testes de Estrutura de Controle

A técnica de teste de caminho base é uma das várias abordagens para testar a estrutura de controle de um programa. Embora o teste de caminho base seja simples e altamente eficaz, ele por si só não é suficiente. Nesta seção, exploramos outras variações do teste de estrutura de controle, que ampliam sua abrangência e melhoram a qualidade do teste caixa-branca.

O teste de condição é um método para projetar casos de teste que exercitam as condições lógicas presentes em um módulo de programa. Por outro lado, o teste de fluxo de dados, seleciona caminhos de teste com base na localização de definições e usos de variáveis no programa (Pressman, 2021).

Já o teste de ciclo é uma técnica de teste caixa-branca que se concentra exclusivamente na validação das estruturas de ciclo. Podem ser identificadas duas classes distintas de ciclos, ciclos simples e ciclos aninhados, conforme apresentado na Figura 3.

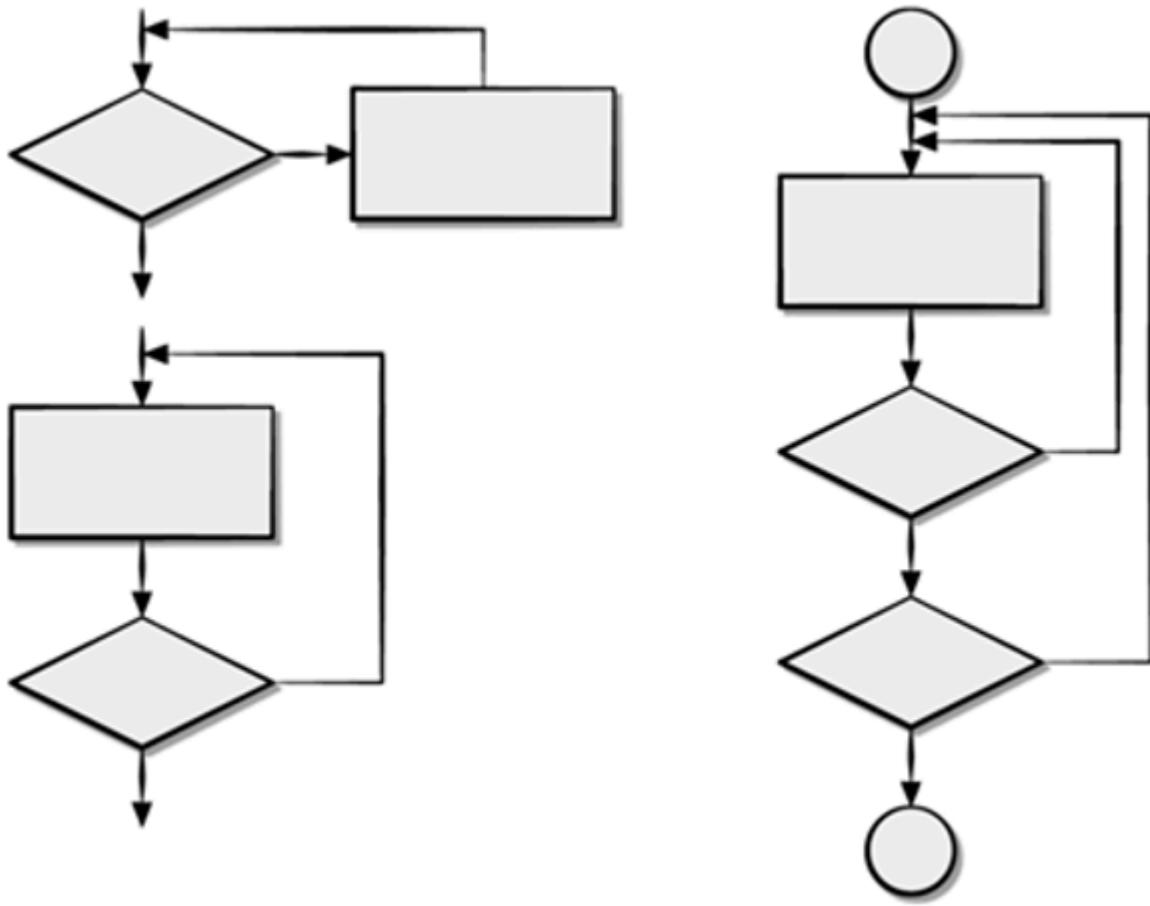


Figura 3 | Classes de Ciclos. Fonte: Pressman (2021).

**Ciclos simples.** O seguinte conjunto de testes pode ser aplicado a ciclos simples, onde  $n$  é o número máximo de passadas permitidas através do ciclo (Pressman, 2021).

- Pular o ciclo inteiramente.
- Somente uma passagem pelo ciclo.
- Duas passagens pelo ciclo.
- $m$  passagens através do ciclo onde  $m < n$ .
- $n - 1, n, n + 1$  passagens através do ciclo.

**Ciclos aninhados.** Se quisermos aplicar a abordagem de teste de ciclos simples para ciclos aninhados, o número potencial de testes aumentaria exponencialmente à medida que o nível de aninhamento aumentasse. Isso resultaria em um número excessivo de testes. Uma estratégia para reduzir a quantidade de testes (Sommerville, 2018):

- Comece pelo ciclo mais interno e defina todos os outros ciclos para seus valores mínimos.
- Execute os testes de ciclo simples para o ciclo mais interno, mantendo os ciclos externos com seus parâmetros mínimos de iteração. Adicione testes para valores fora do intervalo

ou excepcionais.

- Prossiga para o próximo ciclo externo, mantendo todos os outros ciclos externos em seus valores mínimos e os ciclos aninhados com valores "típicos".
- Continue esse processo até que todos os ciclos tenham sido testados.

Essa abordagem permite testar os ciclos aninhados de maneira eficiente, reduzindo o número total de testes necessários enquanto ainda garante uma cobertura adequada dos casos de teste.

## Siga em Frente...

## Testes Baseados em Modelos

Uma especificação pode ser elaborada de várias formas, incluindo descrições textuais em linguagem natural ou modelos formais. A ambiguidade nas descrições textuais pode levar a inconsistências na especificação, destacando a importância de abordagens mais precisas. Modelagem permite capturar e reutilizar conhecimento sobre o sistema, facilitando a compreensão do que o sistema deve fazer. Durante os testes, determinar os objetivos de teste pode ser um desafio, especialmente quando as especificações não são claras. A falta de definição precisa poder levar a interpretações diferentes e consequentemente a testes improdutivos. É crucial uma especificação clara para orientar o desenvolvimento e os testes, evitando discrepâncias na implementação e nas expectativas do teste.

A habilidade de criar modelos não é uma novidade a ser adquirida. Testadores sempre desenvolvem algum tipo de modelo, mesmo que informalmente; caso contrário, seria impossível determinar se o comportamento é aceitável. A questão crucial é o formato desse modelo. Um modelo informal ou apenas mental dificulta a automação dos testes. Para elaborar um script de teste ou um plano de teste, o testador precisa compreender os passos fundamentais necessários para utilizar o sistema (Delamaro, 2018).

As possíveis sequências de ações durante o uso do sistema são delineadas. Geralmente, existem múltiplas opções de "próximas ações" em um ponto específico do processo. Algumas técnicas permitem descrever essas próximas ações em um grafo, nos quais os nós representam as configurações (ou estados) e as arestas representam as transições entre essas configurações. Essas técnicas são coletivamente chamadas de "Máquinas de Transição de Estados". Além disso, algumas técnicas permitem que os modelos sejam decompostos hierarquicamente, o que simplifica comportamentos complexos em comportamentos mais simples e de nível mais baixo.

As capacidades adicionais incluem o uso de variáveis e condicionais, permitindo que as transições dependam de variáveis ou do contexto atual do sistema. Existem várias técnicas baseadas em Máquinas de Transição de Estados, cada uma diferindo em como certos elementos são explicitamente ou implicitamente representados. O modelo mais simples é o das Máquinas de Estados Finitos (MEFs).

Em uma MEF, as configurações e as transições são representadas explicitamente. No entanto, para sistemas com muitas configurações possíveis, uma MEF pode não ser adequada devido à possibilidade de um número excessivamente grande de estados. Para contornar esse problema, outras técnicas foram propostas. Por exemplo, uma MEF estendida inclui conceitos de variáveis de contexto e transições parametrizadas, eliminando a necessidade de representar explicitamente uma configuração específica do sistema no modelo (Delamaro, 2018). Em vez disso, uma configuração é definida pelo estado atual da MEF e pelos valores das variáveis de contexto.

Os métodos TT, UIO, W e DS são técnicas específicas de Testes Baseados em Modelo (TBM) que foram desenvolvidas para auxiliar no processo de geração de casos de teste. Cada um desses métodos possui características distintas e é aplicável em diferentes contextos. Vou explicar brevemente cada um deles (Delamaro, 2018):

- **Método TT (Teste de Transição):** o Método TT é uma técnica que se concentra na cobertura de transições entre estados em um modelo de Máquina de Estados Finitos (MEF). Ele visa garantir que todas as transições no modelo sejam percorridas durante a execução dos testes. O método TT é relativamente simples de ser implementado e é eficaz para identificar falhas relacionadas à lógica de transição entre estados.
- **Método UIO (*Unique Input/Output*):** o Método UIO é uma técnica de teste baseada em sequências de entrada e saída únicas. Ele é especialmente útil para sistemas que envolvem lógica de controle complexa e é projetado para identificar problemas relacionados à combinação de entradas e saídas. o Método UIO busca gerar sequências de teste que revelem possíveis erros no comportamento do sistema de forma eficiente.
- **Método W:** o Método W é uma técnica de teste baseada em MEF que visa maximizar a cobertura de estados no modelo. Ele utiliza um algoritmo de busca para encontrar sequências de teste que percorram o maior número possível de estados no modelo. O Método W é particularmente útil para sistemas com muitos estados, ajudando a garantir uma cobertura abrangente durante o teste.
- **Método DS (Domínio de Sequências):** o Método DS é uma técnica de teste baseada na identificação e na utilização de sequências de teste que abrangem diferentes domínios de entrada. Ele busca identificar classes de equivalência para os dados de entrada e gera sequências de teste que cubram essas classes de forma abrangente. O Método DS é eficaz para garantir uma cobertura adequada dos diferentes cenários de entrada em um sistema.

Esses métodos são apenas algumas das técnicas disponíveis em Testes Baseados em Modelo e cada um deles tem suas próprias vantagens e limitações. A escolha do método mais adequado depende das características do sistema a ser testado e dos objetivos do teste. No entanto, todos esses métodos compartilham o objetivo comum de utilizar modelos para orientar e otimizar o processo de teste de software.

## Vamos Exercitar?

O grafo de fluxo de programa é uma representação do código do programa para construir um método de teste estrutural. Uma possível solução para o exercício pedido é:

A marcação do código é:

```
public boolean saque(double valor){  
    boolean status = false; //1  
    if(valor <= saldo){ //1  
        saldo = saldo//2 valor; //2  
        status=true;  
    }  
    else{ //3  
        if(valor <= limiteCredito){ //3  
            limiteCredito = limiteCredito - valor; //4  
            status = true; //4  
        }  
    }  
    return status; //5  
}
```

Figura 4 | Código com marcação

O grafo resultante é:

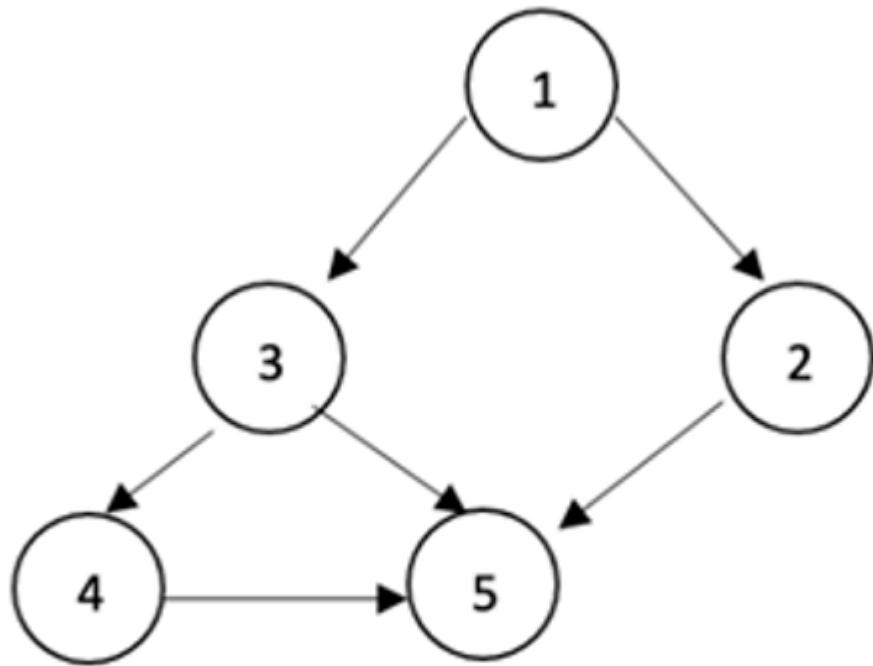


Figura 5 | Grafo resultante

## Saiba mais

Para ler mais sobre o Teste de caminho básico, acesse o livro do Pressman, Capítulo 19.4.1. [Engenharia de Software - Pressman](#).

O livro de testes do Delamaro oferece uma abordagem abrangente e prática para dominar as técnicas essenciais de verificação e validação de software. Para ler mais sobre os Testes baseados em modelo, leia o Capítulo 3 [Introdução ao Teste de Software](#).

Para ler mais sobre o Teste de estrutura de controle, acesse o livro do Pressman, Capítulo 19.4.2. [Engenharia de Software - Pressman](#).

## Referências

DELAMARO, M. **Introdução ao teste de software**. Rio de Janeiro, Elsevier, 2016.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Aula 4

Desenvolvimento orientado a testes

### Desenvolvimento orientado a testes

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

##### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Na aula de hoje você explorará o Desenvolvimento Orientado a Testes (TDD), Gerenciamento de Testes e Testes Automatizados de Software. Esses temas são essenciais para a prática profissional, capacitando os alunos a criarem software robusto, gerenciar eficientemente os processos de teste e automatizar tarefas repetitivas. Prepare-se para adquirir habilidades fundamentais que impulsionarão sua carreira no desenvolvimento de software de forma eficaz e eficiente.

## Ponto de Partida

Nesta aula serão introduzidos três conceitos fundamentais relacionados ao teste de software: Desenvolvimento Orientado a Testes (TDD), Gerenciamento de Testes e Testes Automatizados de Software. O TDD é uma abordagem que enfatiza escrever testes automatizados antes mesmo de escrever o código de produção, garantindo que o código seja mais robusto e tenha uma cobertura de testes abrangente desde o início do processo de desenvolvimento. O

Gerenciamento de Testes é crucial para organizar e coordenar todas as atividades relacionadas aos testes, incluindo planejamento, execução e relatórios de resultados, garantindo que os testes sejam realizados de forma eficiente e eficaz. Por fim, os Testes Automatizados de Software são essenciais para aumentar a velocidade e a confiabilidade dos testes, permitindo que sejam executados de forma rápida e repetitiva, o que é especialmente importante em projetos de desenvolvimento ágil e contínuo.

Esses conceitos são de extrema importância para garantir a qualidade do software e para promover práticas de desenvolvimento eficientes. O TDD promove uma abordagem proativa em relação aos testes, o que resulta em código mais limpo, modular e passível de manutenção. O Gerenciamento de Testes ajuda a garantir que os recursos de teste sejam alocados de forma adequada, priorizando testes críticos e maximizando o retorno sobre o investimento em testes. Já os Testes Automatizados de Software permitem uma rápida detecção de regressões e problemas de integração, proporcionando um feedback instantâneo aos desenvolvedores e reduzindo o tempo necessário para identificar e corrigir falhas. Em conjunto, esses conceitos formam uma base sólida para a implementação de processos de teste eficazes e eficientes em qualquer projeto de desenvolvimento de software.

Pensando em soluções com TDD, vamos para a seguinte situação hipotética: a Biblioteca Municipal de LivroVille decidiu modernizar seu sistema de gerenciamento, substituindo o antigo sistema manual por um novo sistema informatizado. O objetivo principal do novo sistema é automatizar as funções de cadastro de livros, empréstimo, devolução e busca de livros disponíveis. A equipe de desenvolvimento escolheu usar a metodologia TDD para garantir a qualidade do software desde o início e facilitar a manutenção futura do sistema.

O primeiro desafio enfrentado pela equipe foi o desenvolvimento do módulo de cadastro de livros. O módulo precisava permitir que os bibliotecários adicionassem novos livros ao sistema, incluindo informações como título, autor, ano de publicação e categoria. Sugira uma solução utilizando o TDD como norteador.

Bons estudos!

## Vamos Começar!

## Desenvolvimento Orientado a Testes (TDD)

Considerando o momento da aplicação dos testes, a fase em que ela ocorre é, tradicionalmente, uma das últimas do processo de software, logo após o completo desenvolvimento do produto e antes da entrega ao cliente. Via de regra, qualquer atraso ocorrido durante o projeto impactará no tempo disponível para os testes. Haveria então outro momento mais adequado para aplicá-los? Será que a prática do teste também não teria experimentado uma evolução, impulsionada pelo aprimoramento das metodologias de desenvolvimento? Felizmente a resposta é sim para ambas as questões.

A chegada das metodologias ágeis, em particular o *Extreme Programming* (XP), trouxe inovações na forma como conduzimos nossos testes, com o *Test Driven Development* (TDD) destacando-se como a mais significativa delas. Segundo Aniche (2015), o cerne do TDD é a prática de escrever testes automatizados de forma contínua ao longo do processo de desenvolvimento e antes mesmo da implementação do código. Essa mudança no ciclo de desenvolvimento incentiva o desenvolvedor a produzir um código de qualidade superior, uma vez que a escrita de testes de unidade eficazes exige a correta utilização dos recursos da orientação a objetos.

Antes de prosseguirmos com a exploração deste tópico, é necessário revisitar alguns termos já mencionados para delimitar nosso tema. O Desenvolvimento Orientado a Testes será tratado aqui como uma das práticas do XP, e, nesse contexto, focalizaremos o teste de unidade como objeto de análise. A inclusão do TDD no âmbito do XP justifica nossa referência ao paradigma de Orientação a Objetos mencionado anteriormente. Por fim, é importante lembrar que o teste de unidade é realizado em cada classe do sistema, enquanto os testes de aceitação são aplicados a cada funcionalidade ou história do usuário (Teles, 2004).

Então, como exatamente o Desenvolvimento Orientado a Testes funciona? Vamos descobrir isso ao longo desta primeira parte da seção, começando pela introdução de uma figura que se tornou uma referência universal quando se trata desse assunto. A aplicação do Desenvolvimento Orientado a Testes é exemplificada pelo ciclo Vermelho-Verde-Refatorar, que é apresentado na Figura 1.

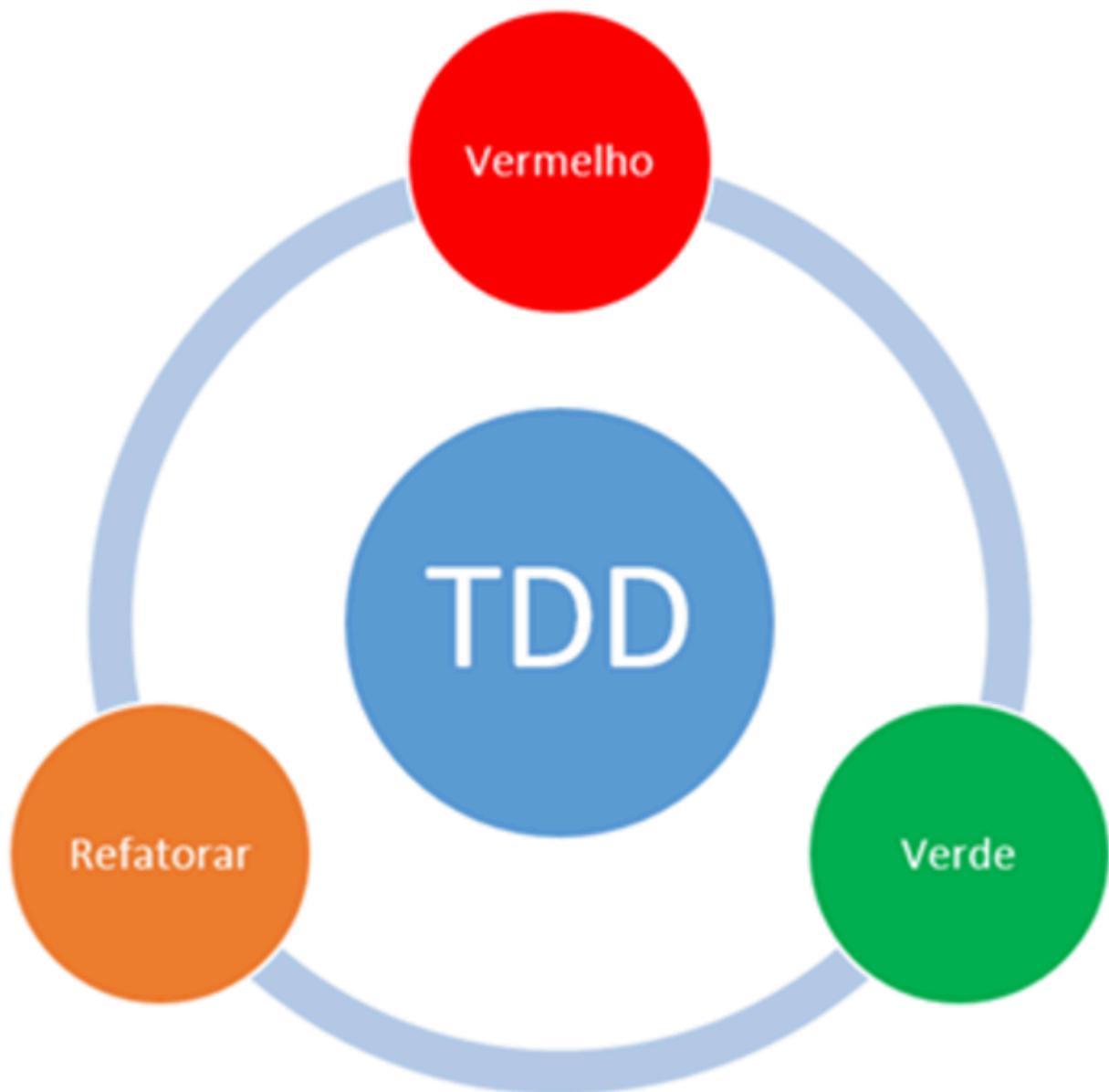


Figura 1 | Ciclo Vermelho-Verde-Refatorar. Fonte: adaptada de Qiu (2018).

Essa representação pode ser expressa textualmente da seguinte maneira:

- Na primeira etapa do ciclo, o desenvolvedor escreve um teste que está destinado a falhar deliberadamente. O código em que o teste é incorporado provavelmente não irá compilar.
- Em seguida, o desenvolvedor escreve o código referente a uma nova funcionalidade do sistema, com o objetivo de fazer o teste escrito na etapa anterior passar com sucesso.
- Como última etapa dessas três fases, o desenvolvedor realiza a refatoração no código, eliminando possíveis duplicidades, melhorando a estrutura do código, alterando o nome de variáveis, entre outras ações.

Apesar dessa ilustração nos fornecer uma ideia do ciclo, ela não esclarece completamente como ele se desenrola, especialmente para aqueles que estão utilizando o TDD pela primeira vez. A Figura 2 oferece uma visão detalhada do procedimento. Na realidade, o TDD consiste em duas partes: implementação rápida e refatoração, e na prática, o teste para a implementação rápida não se limita apenas ao teste de unidade, podendo incluir também testes de aceitação.

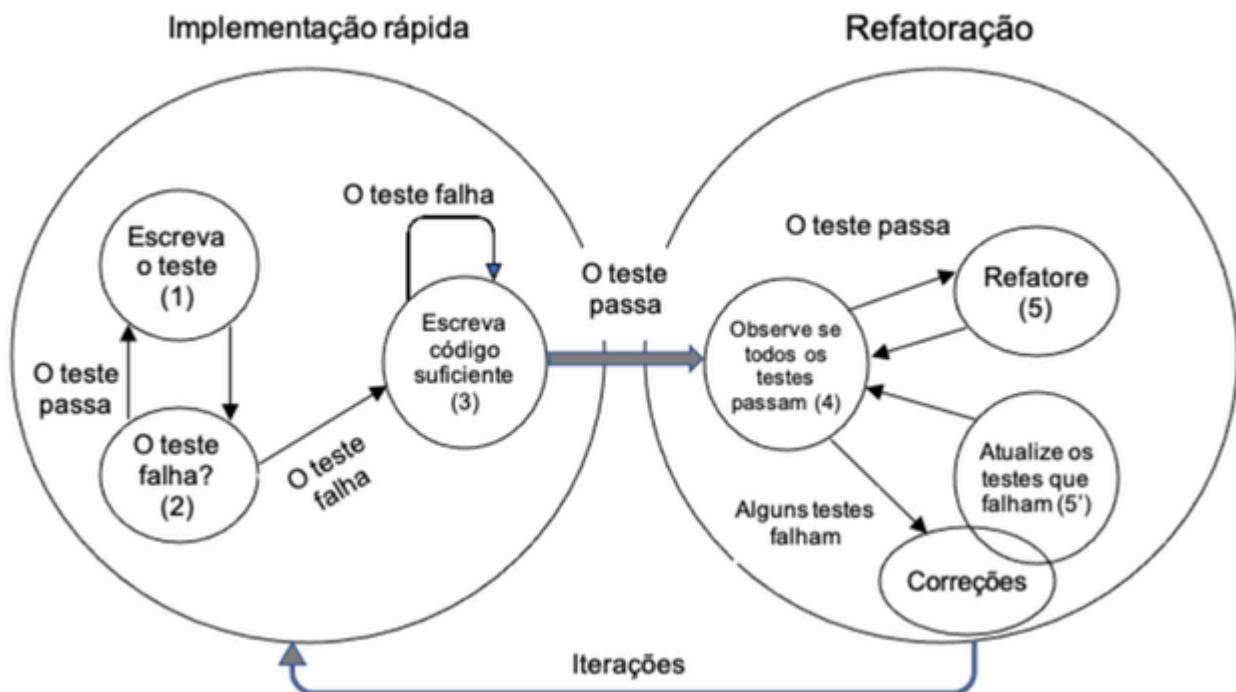


Figura 2 | O procedimento de TDD em detalhes. Fonte: adaptada de Qiu (2018).

Observe que na parte de implementação rápida, o processo conduzido pelo desenvolvedor é visualmente retratado da mesma maneira que descrevemos textualmente, ou seja, (1) o desenvolvedor escreve um teste destinado a falhar e (2) verifica se, de fato, o teste falha. Se o teste não falhar, ele deve ser reescrito. Se falhar, (3) o desenvolvedor procede escrevendo o código para que o teste seja bem-sucedido. Neste ponto, uma observação importante deve ser feita: a expressão "código suficiente" indica que o desenvolvedor só deve codificar o necessário para que o teste seja aprovado, sem exageros. Quando o teste é aprovado, a etapa de refatoração inicia e, como primeira ação dessa etapa, (4) o desenvolvedor verifica se todos os testes passam e (5) realiza a refatoração no código. Se um ou mais testes falharem, (5') eles devem ser atualizados e quaisquer correções necessárias devem ser feitas. A seta com a legenda "Iterações" indica que este processo deve ser repetido até a conclusão do procedimento.

Neste ponto cabe um exemplo da aplicação do TDD e novamente usaremos a loja de comércio eletrônico para desenvolvê-lo, segundo Aniche (2015). A classe exibida no código da Figura 3 aponta o produto de maior valor e o produto de menor valor no carrinho e o faz percorrendo a lista de compras e comparando valores.

```
1  public class MaiorEMenor {  
2      private Produto menor;  
3      private Produto maior;  
4      public void encontra (CarrinhoDeCompras carrinho) {  
5          for (Produto produto : carrinho.getProdutos()) {  
6              if (menor == null || produto.getValor() < menor.getValor()) {  
7                  menor = produto  
8              }  
9              else if (maior == null || produto.getValor() > maior.getValor()) {  
10                  maior = produto;  
11              }  
12          }  
13      }  
14      public Produto getMenor() {  
15          return menor;  
16      }  
17      public Produto getMaior() {  
18          return maior;  
19      }  
20  }  
21 }  
22 }
```

Figura 3 | Classe que retorna o produto de maior valor e de menor valor. Fonte: Aniche (2015).

O método `encontra()` recebe um `CarrinhoDeCompras` e percorre a lista de produtos desse carrinho, comparando sempre o produto atual com o menor e maior já encontrados. Ao final da execução, temos nos atributos `maior` e `menor` os produtos desejados. A Figura 4 exemplifica o uso da classe `MaiorEMenor`. Observe que o carrinho contém três itens: o que tem preço menor é o jogo de pratos e o que tem preço maior é a geladeira. Ao executar essa aplicação, a saída será:

O menor produto: *jogo de pratos*

O maior produto: *geladeira*.

```
1 public class TestaMaiorEMenor {  
2     public static void main(String[] args) {  
3         CarrinhoDeCompras carrinho = new CarrinhoDeCompras();  
4         carrinho.adiciona(new Produto("Geladeira", 450.0));  
5         carrinho.adiciona(new Produto("Liquidificador", 250.0));  
6         carrinho.adiciona(new Produto("Jogo de pratos", 70.0));  
7  
8         MaiorEMenor algoritmo = new MaiorEMenor();  
9         algoritmo.encontra(carrinho);  
10    System.out.println("O menor produto: " + algoritmo.getMenor().getNome());  
11    System.out.println("O maior produto: " + algoritmo.getMaior().getNome());  
12    }  
13    }  
14 }
```

Figura 4 | Classe que utiliza a classe MaiorEMenor. Fonte: Aniche (2015).

Sendo assim, podemos concluir que a aplicação funciona, certo? Ainda não. Precisamos verificar se ela funciona também em outro cenário. Se o código do desenvolvedor fizesse a inserção dos mesmos produtos em ordem diferente (geladeira, liquidificador e jogo de pratos), a saída gerada seria a seguinte:

```
menor produto: jogo de pratos.  
Exception in thread "main" java.lang.NullPointerException at  
TestaMaiorEMenor.main(TestaMaiorEMenor.java:5).
```

É evidente que se os produtos forem adicionados em ordem decrescente, a classe não produzirá a saída esperada, dificultando a realização da compra pelo cliente. Essa circunstância nos leva a duas conclusões:

- Testar constantemente, considerar vários cenários e repetir o procedimento a cada mínima alteração são medidas essenciais em um processo de teste.
- Realizar o teste manualmente (ou seja, sem o auxílio de uma ferramenta de teste) conforme indicado no item anterior é inviável.

Além disso, nesta seção abordaremos os testes automatizados de software como uma solução possível para esse problema. Antes disso, porém, discutiremos algumas questões sobre o teste de unidade e seu gerenciamento, sempre dentro do contexto do Desenvolvimento Orientado a

Testes. Em seguida, apresentaremos uma solução viável para a situação mencionada anteriormente.

## Siga em Frente...

### Gerenciamento de Testes

Em muitos casos, adotar a prática de criar testes antes de codificar não é o modelo de desenvolvimento ao qual um profissional de TI está habituado. Além da falta de familiaridade com esse estilo, existem questões relacionadas ao teste de unidade que exigem um gerenciamento adequado por parte dos responsáveis. É importante destacar que o teste de unidade é o componente central do Desenvolvimento Orientado a Testes (TDD), embora também possa abranger o teste de aceitação. As situações a seguir demandam um gerenciamento eficiente para garantir a condução satisfatória do TDD (Teles, 2004).

#### ***Como escrever testes quando o sistema faz acesso a um banco de dados?***

Nesta perspectiva, o desempenho emerge como um aspecto crucial a ser cuidadosamente considerado durante o procedimento de teste. É imperativo que os testes de unidade sejam executados de forma rápida e eficiente, garantindo assim que o ciclo "testar-codificar-refatorar" seja concluído em um período adequado. O desempenho dos testes torna-se ainda mais relevante quando múltiplas classes estão envolvidas no acesso ao banco de dados. O gerenciamento desta situação envolve a implementação de estratégias para que uma quantidade substancial de testes que dependem de dados do banco seja adaptada para acessar arquivos na memória volátil, em vez de utilizar o registro em disco. Uma abordagem viável seria desenvolver uma classe simulada que funcione como um banco de dados, interceptando os comandos SQL enviados pela aplicação. Embora essa solução possa ser eficaz, é importante ressaltar que, em algum momento, será necessário testar o acesso ao banco de dados real. No entanto, uma gestão cuidadosa da situação deve garantir que tais acessos sejam realizados com a menor frequência possível.

#### ***O que o desenvolvedor deve fazer quando não tiver ideia de como testar uma classe?***

Embora a abordagem dessa situação esteja diretamente ligada à classe específica em questão, é natural suspeitar que estamos lidando com uma classe problemática. Se a instância da classe é feita de maneira complexa, é provável que um padrão de codificação viável não tenha sido seguido, tornando aconselhável uma revisão do código. Se a classe sob teste interage com várias outras classes de complexidade elevada, a solução pode envolver a criação de mock objects, que são objetos simulados que imitam o comportamento de objetos reais mais complexos.

O termo "*mock objects*" (ou objetos simulados, em português) é usado para descrever uma categoria específica de objetos que simulam objetos reais para fins de teste. Eles podem ser criados utilizando frameworks que facilitam consideravelmente esse processo. Quase todas as

principais linguagens de programação possuem frameworks disponíveis para a criação de *mock objects* (Medeiros, 2014).

O gerenciamento dessa situação envolve a realização de reuniões com a equipe de desenvolvedores para discutir os desafios encontrados ao adotar o TDD.

### ***Como saber se foi testado tudo o que poderia dar errado?***

As características de bom senso e experiência desempenham um papel fundamental nessa questão. Quando o sistema apresenta um erro durante o teste de aceitação, é provável que esteja faltando um teste de unidade. Ao escrever esse teste, o desenvolvedor deve ser incentivado a lembrar-se de outros testes que possam não ter sido escritos, o que o ajudará a garantir a abrangência dos testes.

Os desafios associados ao TDD não se limitam aos mencionados anteriormente. É comum encontrar indivíduos que acreditam que essa metodologia aumenta a carga de trabalho da equipe de desenvolvimento. No entanto, como o TDD enfatiza a escrita dos testes antes da codificação efetiva, a simplificação da implementação das classes é um dos benefícios desse estilo de desenvolvimento. Além disso, como o TDD é uma prática do XP, é importante ressaltar que a programação em pares é um aspecto extremamente positivo em termos de eficiência na criação dos testes, devido à colaboração intensa entre os membros da dupla.

## **Testes Automatizados de Software**

Testar uma unidade em vários cenários e repetidamente é uma etapa fundamental para garantir a qualidade desejada do produto. No entanto, essa necessidade não pode ser atendida apenas com o esforço humano; é essencial automatizar o teste. Teles (2004) explica que os testes de unidade são automatizados na forma de classes do sistema, que têm como finalidade testar outras classes. O autor ressalta que, em geral, para cada classe do sistema, deve existir outra classe cujo único propósito seja testá-la, sendo este o primeiro passo para automatizar os testes de unidade.

De fato, a automação do processo reduz o custo e o tempo dos testes, que são fatores interligados. Aniche (2015) argumenta que escrever um teste automatizado não é uma tarefa complexa, pois se assemelha a um teste manual. Para ilustrar esse ponto, consideremos um exemplo de uma aplicação de comércio eletrônico. Digamos que um desenvolvedor precise testar o carrinho de compras de uma loja virtual que contém dois produtos cadastrados. Ao simular o comportamento de um cliente, o desenvolvedor seleciona os dois produtos, adiciona-os ao carrinho e verifica a quantidade de itens presentes no carrinho, assim como o valor total da compra. O resultado esperado seria uma quantidade de itens igual a dois e o valor total da compra sendo a soma dos valores dos dois produtos.

O que o desenvolvedor fez foi criar um cenário (a compra de dois produtos), realizar uma ação (adicionar os ao carrinho) e verificar o resultado (verificar a quantidade e o valor dos itens comprados). Em um teste automatizado de software, a sequência de ações deve seguir

exatamente esse padrão, com a intervenção humana limitada à verificação do valor obtido em comparação com o valor esperado. Ao revisitar o Código presente na Figura 4, percebemos que a classe constrói um cenário (um carrinho de compras com três produtos), executa uma ação (chamada do método `encontra()`) e valida a saída, que consiste em imprimir o maior e o menor produto. A execução simples da classe monta o cenário e executa a ação, sem intervenção direta do desenvolvedor (Aniche, 2015).

Embora já tenhamos automatizado duas das três etapas da sequência, ainda será necessário tornar a conferência da saída independente da intervenção humana. A plena automatização do teste virá com a utilização do JUnit, o framework de testes de unidade do Java, que funciona em conjunto com o IDE Eclipse. A adaptação do nosso código ao JUnit será simples e atingirá a parte da validação, na qual os métodos do framework serão invocados para que comparem o resultado esperado com o obtido. O método `Assert.assertEquals()` fará esse trabalho. O Código, apresentado na Figura 5, mostra a classe `TestaMaiorEMenor` ajustada para funcionamento do JUnit. Como sabemos, esse teste falhará, pois há um defeito na classe `MaiorEMenor`, instanciada pela `TestaMaiorEMenor`. A presença do `else` naquele código não permite que o segundo `if` seja executado e, portanto, o maior elemento nunca é verificado. Com esse defeito corrigido, o teste passará.

```
1 import org.junit.Assert;
2 import org.junit.Test;
3 public class TestaMaiorEMenor {
4     @Test
5     public void ordemDecrescente() {
6         CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
7         carrinho.adiciona(new Produto("Geladeira", 450.0));
8         carrinho.adiciona(new Produto("Liquidificador", 250.0));
9         carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
10
11         MaiorEMenor algoritmo = new MaiorEMenor();
12         algoritmo.encontra(carrinho);
13         Assert.assertEquals("Jogo de pratos", algoritmo.getMenor().getNome());
14         Assert.assertEquals("Geladeira", algoritmo.getMaior().getNome());
15     }
16 }
```

Figura 5 | Código adequado para teste no JUnit. Fonte: Aniche (2015).

Dessa forma é que um teste automatizado se efetiva. Naturalmente que a complexidade, a quantidade e o tamanho das classes tornarão os testes mais ou menos complexos, mas a forma geral não se altera significativamente. A agilidade no procedimento, a redução de custos e a

capacidade de o teste ser repetido quantas vezes forem necessárias são apenas algumas poucas vantagens da automação.

## Vamos Exercitar?

O Desenvolvimento Orientado por Testes (*Test-Driven Development – TDD*) é uma metodologia de desenvolvimento de software que enfatiza a escrita de testes antes de escrever o código propriamente dito. O processo do TDD pode ser resumido em três etapas principais: escrever um teste que falha, escrever o código mínimo necessário para fazer o teste passar e, por fim, refatorar o código para melhorar a estrutura e a clareza, garantindo que todos os testes continuem passando. Vamos examinar um estudo de caso fictício que ilustra a aplicação do TDD em um projeto de desenvolvimento de software.

### Estudo de Caso: Sistema de Gerenciamento de Biblioteca

#### Contexto

A Biblioteca Municipal de LivroVille decidiu modernizar seu sistema de gerenciamento, substituindo o antigo sistema manual por um novo sistema informatizado. O objetivo principal do novo sistema é automatizar as funções de cadastro de livros, empréstimo, devolução e busca de livros disponíveis. A equipe de desenvolvimento escolheu usar a metodologia TDD para garantir a qualidade do software desde o início e facilitar a manutenção futura do sistema.

#### Desafio

O primeiro desafio enfrentado pela equipe foi o desenvolvimento do módulo de cadastro de livros. O módulo precisava permitir que os bibliotecários adicionassem novos livros ao sistema, incluindo informações como título, autor, ano de publicação e categoria.

#### Solução com TDD

- 1. Escrever Testes Antes do Código:** antes de escrever o código para o módulo de cadastro de livros, a equipe desenvolveu testes para as funcionalidades esperadas. Por exemplo, um teste foi criado para verificar se, ao adicionar um novo livro, os detalhes dele eram corretamente armazenados no sistema.
- 2. Implementação do Código para Passar os Testes:** com os testes em mãos, a equipe começou a implementar o código do módulo de cadastro de livros. O foco era escrever o mínimo de código necessário para fazer os testes passarem. Isso resultou em uma implementação inicial que satisfazia os requisitos básicos do cadastro de livros.
- 3. Refatoração:** após fazer o código passar nos testes, a equipe procedeu à refatoração do código para melhorar sua estrutura, legibilidade e eficiência, sempre garantindo que os testes continuassem passando. Durante a refatoração, a equipe melhorou a organização do código, eliminou duplicações e otimizou o armazenamento dos dados.

## Resultados e Benefícios

- **Qualidade do Software:** a adoção do TDD ajudou a equipe a desenvolver um módulo de cadastro de livros robusto e confiável desde o início. Os testes garantiram que o software funcionasse conforme o esperado e ajudaram a identificar e corrigir bugs precocemente no ciclo de desenvolvimento.
- **Manutenção Facilitada:** a base de código bem testada e refatorada facilitou a manutenção e a adição de novas funcionalidades ao sistema no futuro, pois qualquer alteração que quebrasse a funcionalidade existente seria rapidamente identificada pelos testes existentes.
- **Melhoria Contínua:** a metodologia TDD incentivou a equipe a pensar cuidadosamente sobre o design do software e a implementação das funcionalidades antes de escrever o código, levando a melhorias contínuas na qualidade do software.

## Conclusão

O estudo de caso do Sistema de Gerenciamento de Biblioteca de LivroVille demonstra como o TDD pode ser efetivamente aplicado para desenvolver software de alta qualidade. A metodologia ajudou a equipe a enfrentar desafios de desenvolvimento com confiança, garantindo um sistema robusto, fácil de manter e adaptável às necessidades futuras da biblioteca.

## Saiba mais

O método ágil XP, utiliza o desenvolvimento orientado a teste como uma característica do processo. Para saber mais sobre o TDD, leia o capítulo 8.2 de [Engenharia de Software - Sommerville](#)

Quer saber como aplicar o TDD, acesse o artigo [Entendendo e Aplicando o Test Driven Development \(TDD\)](#) e leia mais sobre o tema.

## Referências

ANICHE, M. **Testes automatizados de software:** um guia prático. [S. I.]: Casa do Código, 2015.

MEDEIROS, H. Mocks: introdução à automatização de testes com mock object. **DevMedia**, [S. I.], 2014.

TELES, V. M. **Extreme programming:** aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. São Paulo: Novatec, 2004.

QIU, J. **Acceptance test driven development with react.** [S. I.: s. n.], 2018.

## Aula 5

Encerramento da Unidade

### Videoaula de Encerramento



#### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

##### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Bem-vindo à videoaula essencial sobre Testes de Software! Exploraremos testes funcionais e estruturais, além do Desenvolvimento Orientado a Testes (TDD). Estes conteúdos são fundamentais para sua prática profissional, garantindo a qualidade e confiabilidade do software que você desenvolve. Compreender e dominar essas técnicas não só melhora a eficiência do processo de desenvolvimento, mas também eleva sua credibilidade como profissional na área de desenvolvimento de software. Prepare-se para aprimorar suas habilidades e se destacar no mercado!

### Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é aplicar os principais conceitos de testes de software que é você deverá primeiramente conhecer os conceitos fundamentais da verificação e da validação. Para garantir confiabilidade de sistemas computacionais, a Engenharia de Software desenvolveu mecanismos de Verificação e Validação (V&V), essenciais para garantir que os produtos de software cumpram suas funções com alta qualidade. A V&V, parte do Gerenciamento da Qualidade do Software, busca confirmar se os requisitos estão corretos e se os produtos atendem às expectativas do cliente, sendo a verificação relacionada à construção correta do produto e a validação à garantia de que o produto certo foi construído.

A verificação e a validação estão integradas em um escopo mais amplo, ligado à Garantia da Qualidade do Software (SQA), e abrangem várias atividades, incluindo revisões técnicas, monitoramento de desempenho e testes. Embora os testes desempenhem um papel crucial na identificação de falhas, eles representam apenas a última linha de defesa na garantia da

qualidade (SOMMERVILLE, 2018). A qualidade do software é incorporada e validada durante todo o ciclo de desenvolvimento, sendo essencial aplicar corretamente as técnicas de Engenharia de Software.

Os testes, embora fundamentais, não garantem que o produto entregue esteja livre de falhas. Eles fornecem um meio prático para identificar problemas, mas a qualidade é assegurada ao longo do processo de desenvolvimento, com a aplicação rigorosa de técnicas de Engenharia de Software. Portanto, a Verificação e Validação são partes essenciais do ciclo de vida do software, garantindo que os produtos atendam aos requisitos e padrões de qualidade estabelecidos.

Já os testes de software representam um processo dinâmico, envolvendo uma série de ações e procedimentos realizados por diversos membros da equipe de desenvolvimento para identificar problemas no software. Essencialmente, o teste busca ampliar a percepção da qualidade do produto final, garantindo que atenda às necessidades do usuário. Embora não possa garantir a ausência completa de defeitos, o teste visa identificar problemas, promovendo melhorias contínuas no processo como um todo.

O teste de software consiste na verificação dinâmica do comportamento esperado de um programa em um conjunto finito de casos de teste selecionados do domínio de execução. Esses casos de teste envolvem a execução do programa com entradas pré-definidas e a verificação das saídas correspondentes. Ferramentas como JUnit são comumente utilizadas para a realização desses testes, proporcionando uma avaliação objetiva da funcionalidade do software.

A realização eficaz dos testes depende de um planejamento cuidadoso, que inclui a definição de responsabilidades, recursos necessários e técnicas a serem empregadas. Os testes são divididos em quatro etapas principais: planejamento, projeto de casos de teste, execução dos testes e análise dos resultados.

Os casos de teste são fundamentais nesse processo, representando o elemento central que direciona a avaliação da qualidade do software. Um caso de teste é um conjunto de dados de entrada fornecido a um programa, juntamente com a saída esperada, que deve ser produzida de acordo com os requisitos estabelecidos. Esse processo visa identificar problemas no software, sendo crucial para o sucesso da atividade a escolha adequada dos casos de teste. Por exemplo, ao validar datas inseridas pelo usuário, diferentes combinações de datas são testadas para exercitar diversas partes do programa, aumentando a probabilidade de descobrir defeitos no código.

A qualidade dos casos de teste é essencial, pois um conjunto de baixa qualidade pode não exercitar adequadamente partes críticas do programa, prejudicando a confiabilidade do processo de teste. Por meio de exemplos, como o teste da funcionalidade de login em um sistema de reserva de passagens, ilustra-se a importância de detalhar os cenários de teste e especificar todas as condições para sua verificação (PRESSMAN, 2021). Cada conjunto de casos de teste deve estar associado a requisitos significativos, e uma abordagem eficaz envolve o planejamento, entendimento da aplicação e definição clara dos passos a serem executados.

Após a execução de um caso de teste, o resultado deve ser registrado, sendo comum classificá-lo como "passou", "falhou" ou "bloqueado". Essa etapa é crucial para avaliar o desempenho do software e identificar possíveis áreas de melhoria, contribuindo para o processo de desenvolvimento de software com alta qualidade e confiabilidade.

A estratégia de teste é essencial para estabelecer um plano detalhado que guie o processo de teste de um produto de software. Elementos como revisões de software, progressão do teste, depuração e seleção de técnicas são fundamentais nessa estratégia. As técnicas de teste são escolhidas com base nos objetivos específicos de teste e nas propriedades do sistema a serem avaliadas, como implementação correta de especificações funcionais, desempenho e usabilidade. Além disso, as diferentes abordagens de teste são posicionadas em três dimensões: quando testar, o que testar e como testar, para facilitar a compreensão e a aplicação das técnicas de teste.

Outro ponto importante para aplicar a competência desta unidade é compreender as abordagens de testes funcionais e estruturais. A abordagem de teste funcional se baseia nas especificações do software para derivar os requisitos de teste e se concentra nas funcionalidades do programa, visando verificar se as saídas geradas estão de acordo com as expectativas. Essa técnica não requer conhecimento detalhado da implementação do código, tornando-se uma opção viável mesmo sem acesso ao código-fonte. Por meio dessa abordagem, o testador avalia apenas os resultados obtidos para cada entrada fornecida, o que levou ao termo "teste de caixa preta".

O teste funcional é dividido em subtipos, como teste de unidade, teste de integração e teste de regressão, cada um focado em diferentes aspectos das funcionalidades do sistema. O planejamento desse tipo de teste envolve a identificação das funções a serem testadas e a criação de casos de teste para verificação. Além disso, existe o teste de componente, que se concentra em verificar a funcionalidade e/ou usabilidade de partes individuais do sistema, como o componente de login em um sistema de vendas online, por exemplo. Essa técnica permite examinar a operacionalidade de cada função específica, garantindo que o software atenda aos requisitos do usuário final (SOMMERVILLE, 2018).

O teste estrutural, também conhecido como caixa branca, é uma técnica que se baseia na arquitetura interna do programa, permitindo ao testador examinar o código-fonte e a estrutura do banco de dados para submeter o programa a uma ferramenta automatizada de teste. Essa abordagem é situada na mesma categoria do teste funcional, denominada "Como testar", e pode ser realizada de forma automatizada. Ao analisar o código do programa, uma ferramenta pode construir uma representação conhecida como grafo, onde os nós correspondem a blocos de código e as arestas representam o fluxo entre os nós (SOMMERVILLE, 2018).

Um exemplo prático desse teste é ilustrado com um programa que verifica a validade de um nome de identificador, onde cada trecho identificado no código é transformado em um nó no grafo gerado pela ferramenta de teste estrutural. O testador então seleciona casos de teste capazes de percorrer os nós, arcoss e caminhos representados no grafo do programa. Diferentes casos de teste exercitam sequências distintas do grafo, sendo essencial definir critérios de cobertura do código para garantir uma avaliação abrangente do programa e evitar escolhas incorretas que possam comprometer o teste.

Após entender as técnicas de testes, temos que compreender os testes que se baseiam nestas técnicas. O teste de caminho básico é uma técnica de teste de caixa-branca que permite ao projetista de casos de teste derivar um conjunto-base de caminhos de execução com base na complexidade lógica de um projeto procedural. Utilizando um grafo de fluxo, que representa a estrutura de controle do programa de forma simplificada, é possível mapear os caminhos independentes através do programa. Cada caminho independente deve introduzir pelo menos uma nova aresta no grafo, e um conjunto básico de caminhos garante a execução de cada comando do programa pelo menos uma vez.

O número de caminhos independentes a serem buscados é determinado pelo cálculo da complexidade ciclomática, uma métrica que oferece uma medida quantitativa da complexidade lógica de um programa. Existem diferentes formas de calcular a complexidade ciclomática, todas relacionadas ao número de regiões, nós e predicados no grafo de fluxo. Uma vez calculada a complexidade ciclomática, é estabelecido um limite superior para o número de caminhos independentes, indicando a quantidade máxima de testes necessários para garantir a cobertura do programa (PRESSMAN, 2021).

No contexto do método de teste de caminho básico, a complexidade ciclomática define o número máximo de caminhos independentes no conjunto-base de testes. Isso permite planejar um conjunto eficiente de testes que assegure a execução de todos os comandos do programa. Para um grafo de fluxo específico, o valor da complexidade ciclomática estabelece que seriam necessários no máximo quatro casos de teste para abranger cada caminho lógico independente.

O teste de caminho base é uma abordagem fundamental para testar a estrutura de controle de um programa, porém é necessário explorar outras variações para ampliar sua eficácia. O teste de condição foca nas condições lógicas do código, enquanto o teste de fluxo de dados considera a manipulação das variáveis. Além disso, o teste de ciclo se concentra na validação das estruturas de repetição, distinguindo entre ciclos simples e aninhados.

Para testar ciclos simples, são propostos conjuntos de testes que variam o número de iterações, enquanto para ciclos aninhados, a complexidade exponencial é enfrentada com uma estratégia que começa pelos ciclos internos, mantendo os externos com iterações mínimas e gradualmente aumentando a complexidade dos testes. Esse método permite uma cobertura eficiente dos ciclos aninhados, reduzindo o número total de testes necessários sem comprometer a qualidade da cobertura.

Uma especificação pode ser elaborada de diversas formas, incluindo descrições textuais em linguagem natural ou modelos formais, onde a ambiguidade pode levar a inconsistências na especificação. Modelagem permite capturar e reutilizar conhecimento sobre o sistema, facilitando a compreensão do que o sistema deve fazer. Durante os testes, a determinação dos objetivos pode ser um desafio, especialmente quando as especificações não são claras, podendo resultar em testes improdutivos e discrepâncias na implementação (PRESSMAN, 2021).

Para garantir a clareza das especificações e facilitar a automação dos testes, é essencial adotar abordagens formais de modelagem. Técnicas como Máquinas de Transição de Estados permitem descrever as possíveis sequências de ações durante o uso do sistema, simplificando

comportamentos complexos e garantindo uma representação precisa dos estados e transições. Diferentes métodos de Testes Baseados em Modelo, como TT, UIO, W e DS, oferecem maneiras específicas de gerar casos de teste eficientes a partir desses modelos, cada um com suas vantagens e limitações.

Esses métodos visam maximizar a cobertura dos estados e transições do modelo, garantindo que o sistema seja testado de forma abrangente e eficaz. A escolha do método mais adequado depende das características do sistema e dos objetivos do teste, mas todos compartilham o objetivo comum de utilizar modelos para orientar e otimizar o processo de teste de software.

O teste de software tradicionalmente ocorre após o desenvolvimento completo do produto, mas metodologias como o *Extreme Programming* (XP) trouxeram inovações, como o *Test Driven Development* (TDD). O TDD incentiva os desenvolvedores a escreverem testes automatizados continuamente ao longo do processo de desenvolvimento, antes mesmo da implementação do código, promovendo a produção de um código de maior qualidade (PRESSMAN, 2021).

NO TDD o desenvolvedor escreve um teste que falha, implementa o código para que o teste passe e então refatora o código (SOMMERVILLE, 2018). O TDD consiste em duas partes: implementação rápida e refatoração, com testes de unidade e de aceitação. A prática do TDD requer testes constantes e consideração de vários cenários para garantir a qualidade do código.

O uso de testes automatizados de software, como o TDD, torna-se crucial para verificar constantemente o código, especialmente em cenários onde testar manualmente é inviável. A abordagem do TDD melhora a qualidade do código, promovendo testes contínuos e cuidadosos ao longo do processo de desenvolvimento.

Em situações em que o teste de unidade é central para o TDD, questões relacionadas ao acesso a banco de dados e à complexidade das classes podem surgir, demandando um gerenciamento eficiente. Estratégias para garantir a rapidez e eficiência dos testes, especialmente quando envolvem acesso a dados, incluem a simulação de um banco de dados em memória volátil. Quando surge a dificuldade de testar uma classe específica, a criação de *mock objects* pode ser uma solução, simulando objetos reais mais complexos para fins de teste.

O desafio de garantir que todos os cenários de teste sejam cobertos envolve bom senso e experiência por parte dos desenvolvedores. A detecção de falhas durante os testes de aceitação pode indicar a necessidade de testes de unidade adicionais. Embora alguns considerem que o TDD aumenta a carga de trabalho da equipe de desenvolvimento, ele simplifica a implementação das classes e promove a colaboração intensa entre os membros da equipe, especialmente quando combinado com práticas como programação em pares.

A automação dos testes de unidade é fundamental para garantir a qualidade do produto de software, reduzindo custos e tempo. Teles (2004) destaca que cada classe do sistema deve ter uma classe correspondente dedicada a testá-la, sendo esse o primeiro passo para a automação. Aniche (2015) argumenta que escrever testes automatizados é semelhante a testes manuais, exemplificando com um caso de teste de um carrinho de compras em uma loja virtual.

No processo de automação, o desenvolvedor cria cenários de teste, realiza ações e verifica os resultados, seguindo uma sequência predefinida. A adoção do framework JUnit permite a validação automática dos resultados, garantindo independência da intervenção humana na verificação. Essa abordagem agiliza o teste, reduzindo a necessidade de revisão manual e permitindo a repetição dos testes conforme necessário (Aniche, 2015).

A automação dos testes, embora possa variar em complexidade dependendo do tamanho e da natureza das classes, oferece vantagens como agilidade, redução de custos e repetibilidade dos testes. Essa prática é essencial para a garantia da qualidade do software em desenvolvimento.

Em suma, os testes de software são essenciais para assegurar a qualidade dos produtos desenvolvidos, fornecendo uma abordagem sistemática para detectar e corrigir falhas antes da entrega aos clientes. Nesta unidade, exploramos uma variedade de técnicas, desde o teste de unidade automatizado até práticas mais avançadas, como o Desenvolvimento Orientado a Testes (TDD). A aplicação efetiva desses métodos não só melhora a confiabilidade e a estabilidade do software, mas também aprimora a experiência do usuário. Ao continuarmos a aprimorar nossas habilidades em testes de software, estaremos preparados para enfrentar os desafios em evolução do desenvolvimento de software e alcançar novos patamares de excelência na indústria de tecnologia.

## É Hora de Praticar!



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Este estudo de caso abordará os testes funcionais e estruturais. Portanto, antes de propor uma possível solução para o desafio descrito, pense em quais testes pertencem a essas estratégias.

#### **Estudo de Caso: Implementação de Testes Funcionais e Estruturais**

Recentemente uma empresa de desenvolvimento de software especializada em soluções para o setor financeiro iniciou o desenvolvimento de um novo sistema de gerenciamento financeiro (*Financial Management System - FMS*) destinado a instituições bancárias. Dada a criticidade e complexidade do sistema, a empresa reconheceu a importância de adotar uma abordagem robusta de testes de software que incluisse tanto testes funcionais quanto estruturais.

O FMS é um sistema complexo que integra várias funcionalidades, incluindo gestão de contas, transações financeiras e relatórios. Garantir a precisão, segurança e desempenho do sistema era essencial, considerando as rigorosas regulamentações do setor financeiro. A empresa enfrentou o desafio de implementar uma estratégia de testes que abordasse tanto a validação das funcionalidades do sistema (testes funcionais) quanto a avaliação da sua estrutura interna e

lógica de programação (testes estruturais). Portanto, apresente possíveis soluções para o desafio que a empresa enfrenta e descreva os possíveis benefícios que a empresa teria com a implementação da solução proposta.

- Como o teste de software pode influenciar não apenas a qualidade do produto final, mas também o processo de desenvolvimento como um todo, incluindo aspectos como colaboração entre equipes, tomada de decisões arquiteturais e abordagem iterativa?
- Como o teste funcional pode desafiar as suposições dos desenvolvedores sobre o comportamento esperado do software e contribuir para uma compreensão mais abrangente das necessidades e expectativas dos usuários finais?
- Como a prática do TDD pode influenciar a cultura de desenvolvimento de software de uma equipe, promovendo maior confiança no código, melhor compreensão dos requisitos do projeto e facilitando a manutenção do código ao longo do tempo?

O estudo de caso demonstra a eficácia de uma abordagem de testes que combina tanto testes funcionais quanto estruturais no desenvolvimento de software complexo e crítico. Implementando testes abrangentes e utilizando automação e feedback contínuo, a empresa conseguiu não apenas melhorar a qualidade do seu sistema de gerenciamento financeiro, mas também cumprir com as exigências regulatórias e atender às expectativas dos clientes no competitivo setor financeiro.

### Solução: Implementação de Testes Funcionais e Estruturais

#### 1. Testes Funcionais

A equipe de QA (Quality Assurance) da empresa começou com a implementação de testes funcionais para validar o comportamento do FMS em relação aos requisitos definidos. Isso incluiu:

- **Testes de Aceitação:** Verificação de que o sistema atendia às expectativas e necessidades dos usuários finais.
- **Testes de Sistema:** Avaliação do sistema como um todo para garantir a integração adequada de todas as funcionalidades.
- **Testes de Recessão:** Realizados após modificações no software para garantir que as mudanças não introduzissem novos defeitos em funcionalidades existentes.

#### 2. Testes Estruturais

Paralelamente, a equipe de QA implementou testes estruturais focados na análise interna do código e estrutura do FMS. Os testes estruturais incluíram:

- **Testes Unitários:** Focados em verificar a menor parte do software, como funções ou métodos, para garantir que cada unidade funcionasse corretamente.
- **Testes de Integração:** Para avaliar a interação entre diferentes módulos ou componentes do sistema, garantindo que eles trabalhassem juntos conforme esperado.

A equipe utilizou ferramentas automatizadas para facilitar a execução desses testes, permitindo a identificação e correção rápida de problemas.

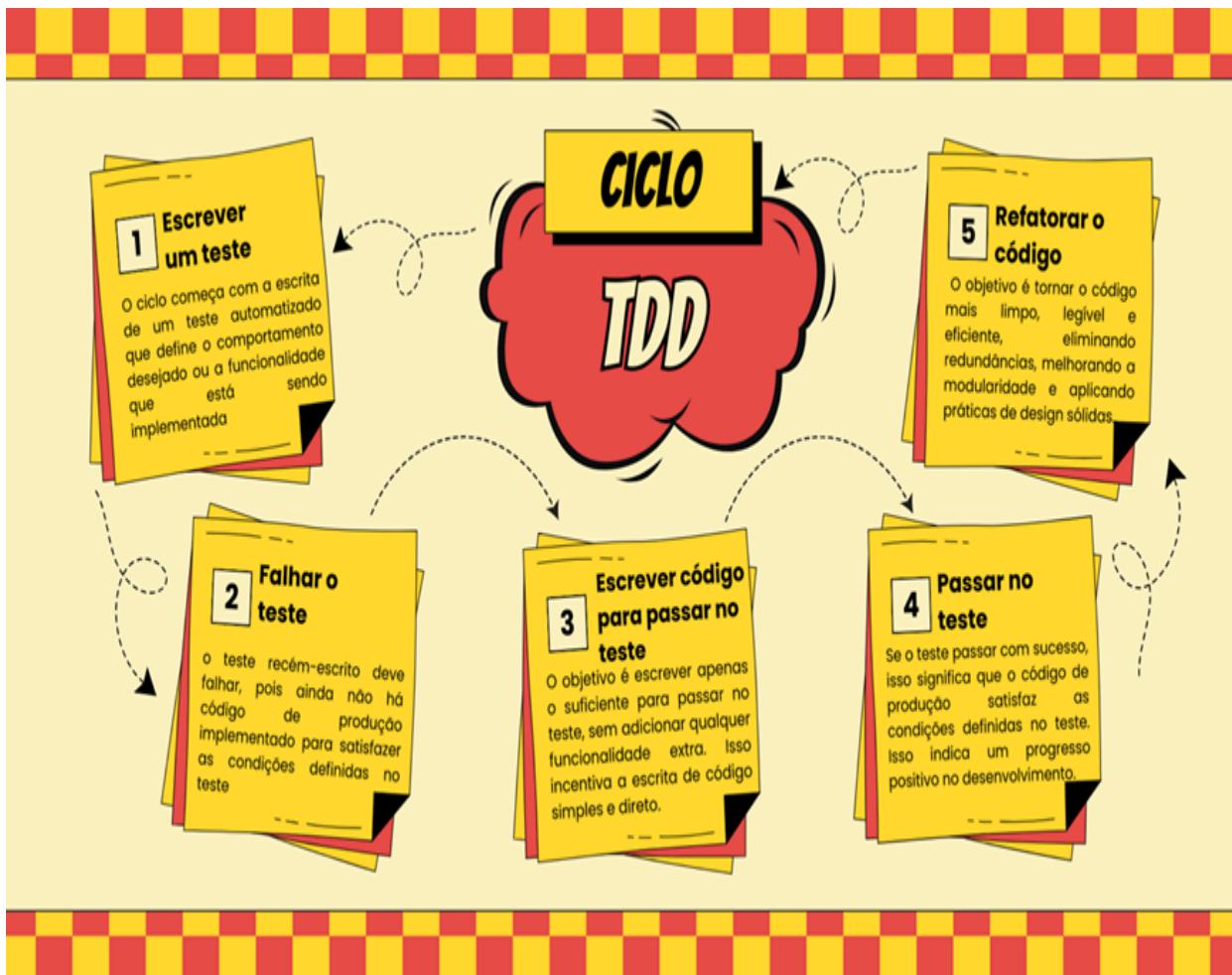
#### 3. Abordagem Iterativa e Feedback Contínuo

A empresa adotou uma abordagem iterativa para os testes, permitindo a identificação precoce de defeitos e a implementação de correções em ciclos de desenvolvimento curtos. Além disso, a equipe promoveu sessões de revisão de código e feedback contínuo entre os desenvolvedores e testadores para melhorar a qualidade do código e a eficácia dos testes.

### Resultados e Benefícios

- **Melhoria na Qualidade do Software:** A combinação de testes funcionais e estruturais resultou em uma melhoria significativa na qualidade do FMS, com uma redução notável no número de defeitos encontrados após o lançamento.
- **Conformidade Regulatória:** Os testes ajudaram a garantir que o sistema atendesse às rigorosas regulamentações do setor financeiro, evitando potenciais penalidades legais e reforçando a confiança dos clientes.
- **Eficiência no Desenvolvimento:** A abordagem iterativa e o uso de automação nos testes aceleraram o ciclo de desenvolvimento, permitindo à empresa responder rapidamente às mudanças nos requisitos do projeto e às necessidades do mercado.
- **Confiança dos Stakeholders:** A estratégia de testes implementada pela empresa fortaleceu a confiança dos stakeholders no sucesso e na confiabilidade do FMS, facilitando a adoção do sistema por instituições bancárias.

O infográfico sobre o ciclo do *Test-Driven Development* (TDD) oferece uma visão clara e concisa do processo iterativo fundamental para o desenvolvimento de software. O infográfico destaca os cinco estágios essenciais do ciclo TDD: escrever um teste, falhar o teste, escrever código mínimo para passar no teste, passar no teste e refatorar o código. Cada etapa é acompanhada de uma breve descrição que demonstram como o ciclo acontece.



ANICHE, M. **Testes automatizados de software: Um guia prático.** [S.I.]: Casa do Código, 2015.  
 PRESSMAN, Roger S. **Engenharia de software: uma abordagem profissional.** 9. ed. – Porto Alegre: AMGH, 2021.

SOMMERVILLE, Ian. **Engenharia de software.** 10. ed. São Paulo: Pearson, 2018.

TELES, V. M. **Extreme Programming:** aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. São Paulo: Novatec, 2004.

## Unidade 4

### Tópicos Avançados em Engenharia de Software

#### Aula 1

Manutenção e Evolução de Software

## Manutenção e evolução de software



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta aula mergulharemos na classificação e nos tipos de atividades de manutenção de software, explorando a engenharia reversa e reengenharia, bem como a evolução contínua do software. Esses temas são essenciais para a prática profissional na área de desenvolvimento de software, permitindo compreender e lidar com os desafios da manutenção e evolução dos sistemas ao longo do tempo. Prepare-se para adquirir insights valiosos e aprimorar suas habilidades nesta área em constante evolução da engenharia de software!

## Ponto de Partida

Nesta aula, exploramos um aspecto fundamental no ciclo de vida do software: manutenção de software. A manutenção de software é uma etapa essencial após o desenvolvimento inicial, envolvendo atividades como correção de bugs, adaptação a novos requisitos e melhorias de desempenho. Compreender a importância da manutenção é crucial, pois ela garante a funcionalidade contínua e a eficácia do software ao longo do tempo, atendendo às necessidades dos usuários e mantendo-o competitivo no mercado.

Em seguida, exploraremos a engenharia reversa e reengenharia como abordagens para lidar com sistemas legados ou desatualizados. A engenharia reversa consiste em analisar um sistema existente para compreender sua estrutura e funcionamento, enquanto a reengenharia visa reestruturar e melhorar o sistema com base nessa análise. Essas práticas são cruciais para modernizar sistemas legados, garantindo que permaneçam relevantes e funcionais em um ambiente tecnológico em constante evolução, além de facilitar a integração com novas tecnologias e requisitos de negócios.

Por fim, discutiremos o conceito de evolução de software, enfatizando a importância de adaptar continuamente o software às mudanças no ambiente operacional e nos requisitos do usuário. Isso inclui a adição de novas funcionalidades, correção de bugs e atualizações para garantir a competitividade e a eficácia contínua do software. Compreender e aplicar práticas eficazes de evolução de software é essencial para manter a relevância e o valor do software ao longo do

tempo, garantindo sua capacidade de atender às necessidades em constante mudança dos usuários e do mercado.

Para exercitar esses conceitos, vamos imaginar que você está analisando um sistema de gestão de bibliotecas desenvolvido há mais de uma década, que utiliza tecnologias que, ao longo dos anos, se tornaram obsoletas. Embora o sistema ainda funcione, enfrenta problemas de desempenho, dificuldades de integração com novos sistemas e uma interface de usuário antiquada. A equipe de TI da biblioteca reconheceu a necessidade de modernizar o sistema para melhorar sua eficiência, usabilidade e capacidade de adaptação a novas tecnologias.

Desafios identificados:

- **Tecnologia obsoleta:** o sistema foi construído usando uma pilha tecnológica que não é mais suportada, dificultando a manutenção e a atualização.
- **Código Monolítico:** a arquitetura monolítica do sistema dificulta a implementação de novas funcionalidades e a correção de bugs.
- **Interface de usuário antiquada:** a interface do usuário não é intuitiva e não segue as práticas modernas de UX/UI, resultando em uma experiência de usuário pobre.
- **Desempenho:** problemas de desempenho afetam a escalabilidade do sistema e a satisfação do usuário.
- **Integração:** dificuldades em integrar o sistema com novas plataformas e serviços on-line.

Com os conceitos de manutenção, evolução e reengenharia, proponha objetivos e soluções para minimizar esses desafios identificados.

Bons estudos!

## Vamos Começar!

## Classificação e tipos de atividades de manutenção de software

Após um período de uso prolongado, os sistemas operacionais eventualmente deixam de receber suporte das empresas, o que implica na interrupção das correções e atualizações disponibilizadas para eles. Por outro lado, os sistemas operacionais de código aberto não são descontinuados, pois a comunidade que os sustenta os aprimora continuamente, implementando correções e introduzindo novas funcionalidades de forma regular.

É notável como diferentes abordagens são adotadas para lidar com o envelhecimento do software. Os processos de evolução e manutenção acompanham o ciclo de vida do sistema e, em algumas situações, quando alcançam o ponto de maturidade, o software pode ser descontinuado. Nesta seção de aprendizado, exploraremos a necessidade de evolução e manutenção do software, discutiremos a classificação e os tipos de atividades de manutenção geralmente realizadas. Também examinaremos o papel das ferramentas nos processos de

manutenção, e, por fim, veremos como a engenharia reversa e a reengenharia podem contribuir para a manutenção e a evolução dos softwares.

Antes de nos aprofundarmos nessas discussões, é importante compreendermos a motivação para estudar esses tópicos. Vamos considerar, por exemplo, o desenvolvimento de um aplicativo para rastreamento de animais de estimação por meio de um chip de localização. Quando lançado, o aplicativo tinha um número limitado de usuários e oferecia apenas métodos de pagamento por cartão (crédito e débito). Com o tempo, o número de usuários cresceu significativamente, novos métodos de pagamento, como o PIX, foram introduzidos, e surgiram novas tecnologias de rastreamento.

Podemos observar como a evolução de outras tecnologias teve um impacto direto na aplicação. Para entendermos melhor os processos de evolução e manutenção, é fundamental compreendermos inicialmente como ocorre o envelhecimento de um software.

Os softwares são construídos como sequências lógicas de algoritmos com o objetivo de cumprir os requisitos estabelecidos, os quais podem ser sujeitos a mudanças de requisitos e ao ambiente operacional. O processo de envelhecimento é inevitável e requer uma análise das causas, a fim de permitir sua evolução e/ou manutenção, garantindo assim sua continuidade. Votorazzo (2018) identifica dois tipos de envelhecimento de software:

- **Falha de adequação:** ocorre quando a equipe encarregada da evolução do software comete erros ou falhas na adaptação ou implementação dos requisitos, resultando muitas vezes na perda da integridade e confiabilidade da aplicação. Exemplo prático: um software de conversão de formatos de vídeo que funcionava em uma versão específica do sistema operacional não é compatível com a versão mais recente desse sistema operacional. Se o software é usado por um usuário comum, é provável que ele busque outra solução. No entanto, se uma empresa depende desse software em suas operações e, em certo ponto, atualiza seus computadores para uma versão não compatível do sistema operacional, isso se torna um problema significativo.
- **Falha na mudança:** ocorre quando uma atualização, manutenção ou implementação afeta negativamente outras funcionalidades já em pleno funcionamento. Exemplo prático: Um e-commerce tem apenas um gerador de boletos funcionando, mas devido a demandas dos usuários, precisa adicionar outras formas de pagamento. Para isso, uma API de módulo de pagamentos é implementada incorretamente. O impacto desse erro é a falha no funcionamento tanto do sistema gerador de boletos quanto do módulo de pagamentos diversos, devido ao desconhecimento da estrutura do gerador de boletos.

Você percebeu a importância da evolução e manutenção dos softwares? São muitos aspectos a serem considerados: mudanças nos sistemas operacionais, padrões de consumo, novas modalidades e métodos de pagamento, questões de segurança, e uma variedade de outros elementos. No entanto, como podemos prever o envelhecimento do software? Isso só seria possível se tivéssemos acesso a informações de todas as empresas de tecnologia da informação, as quais, com uma simples alteração, poderiam influenciar o funcionamento do sistema de alguma forma – o que é praticamente impossível.

Entretanto, compreender a classificação e os tipos de atividades de manutenção de softwares oferece uma vantagem em termos de prontidão para lidar com ajustes, manutenções, atualizações e adaptações do sistema em situações de falhas.

Nesse sentido, Pressman (2021) argumenta que a evolução de um sistema pode ter diferentes propósitos, que vão além da simples correção de falhas, bugs, erros e inconsistências. Para entender melhor como são classificadas as atividades de manutenção, o Quadro 1 apresenta essas diferenças.

CLASSIFICAÇÃO	CONCEITO	APLICAÇÃO
Adaptativa	São modificações necessárias para estar de acordo com novos requisitos, os quais podem ser provenientes de leis, regras, ameaças, meios ou métodos novos.	Recentemente (no ano de 2020) o Banco Central autorizou as instituições financeiras a utilizarem o PIX como método de pagamento. Isso exigiu uma adaptação do sistema de internet banking para que essa nova funcionalidade estivesse disponível aos clientes.
Corretiva	Sua função é corrigir falhas ou qualquer outro aspecto que seja motivo de degradação dos serviços de software. Além disso, a manutenção corretiva pode ocorrer antes, nas fases de desenvolvimento, ou depois, com o software já em funcionamento.	Nas eleições municipais de 2020, para que as pessoas que não foram votar pudessem justificar o voto, o governo federal disponibilizou um aplicativo para mobile conhecido como e-título. No primeiro turno, ele apresentou problemas desde a sua instalação até o processo de realizar a justificativa. A única funcionalidade

		em conformidade era a consulta da situação do eleitor quanto à Justiça Eleitoral. No segundo turno, o aplicativo teve de passar por uma manutenção corretiva para que fossem efetuados os ajustes necessários.
Evolutiva	A manutenção evolutiva tem o objetivo de inserir novas funcionalidades no sistema.	Os chats eram um recurso bastante presente no e-commerce para atendimento aos clientes. Uma evolução desse tipo de atendimento é, em vez de ter um colaborador de plantão para atendimento no chat, utilizar atendimento virtual. Para isso, foram desenvolvidos algoritmos que utilizam inteligência artificial, os quais, com a evolução tecnológica, conseguem responder grande parte das dúvidas dos clientes.

Quadro 1 | Classificação e tipos de manutenção de software. Fonte: adaptado de Pressman (2021).

O conhecimento da classificação dos tipos de manutenção, em termos profissionais, permite que tanto um desenvolvedor quanto um gestor se posicione quanto às reais necessidades dentro da estrutura do sistema, facilitando o direcionamento de recursos dentro do ciclo de vida de desenvolvimento de software. Para tal, é necessário conhecer os processos e as ferramentas utilizados na manutenção de software, os quais são recursos de extrema importância para que, de fato, os processos sejam otimizados.

## Engenharia reversa e reengenharia

A reengenharia de software é um processo destinado a reorganizar e/ou modificar um sistema de forma a restaurar seu desempenho para um nível aceitável. Alguns fatores, como a falta de atualização do software ou um excesso de mudanças frequentes (especialmente quando realizadas por equipes distintas), podem levar à degradação dos serviços prestados pelo sistema (Pádua, 2019).

Nessas circunstâncias, tentar encontrar soluções dentro de um sistema comprometido pode não resultar no desempenho desejado e ainda consumir recursos significativos. Assim, optar por uma reconstrução que inclua a correção de erros e falhas, juntamente com as atualizações necessárias, emerge como uma abordagem eficaz na busca por soluções.

O objetivo da reengenharia de software é reimplementar sistemas legados, buscando (Pádua, 2019):

- **Aprimorar a manutenção:** ao reestruturar o sistema, as futuras atividades de manutenção se tornam mais acessíveis, uma vez que os erros antigos são corrigidos na nova versão.
- **Atualizar a documentação do software:** durante a reestruturação do software, a documentação é atualizada, possibilitando a inclusão de novas informações nos scripts.
- **Refazer a estrutura do sistema:** as estruturas que anteriormente dependiam de soluções paliativas e correções podem ser reformuladas, permitindo a criação de um sistema otimizado que atenda aos requisitos de forma mais eficaz.

Quando um software é desenvolvido, certas funcionalidades e módulos demandam consideráveis esforços de desenvolvimento. Apesar de passar por testes, ao longo do tempo, é comum que o sistema possa apresentar falhas. No entanto, no contexto da reengenharia de software, que pode ser compreendida como uma revisão dos algoritmos, é possível afirmar que a probabilidade de erro é reduzida. Pádua (2019) sustenta que os processos de reengenharia tendem a ter riscos mitigados, uma vez que alguns problemas já foram previamente tratados.

Esses processos são apoiados em metodologias de operacionalização, como apresentado na Figura 1.



Figura 1 | Modelo de reengenharia. Fonte: adaptada de Pressman (2021).

## Siga em Frente...

Observe que, ao longo dessa discussão, esses processos foram explorados e exemplificados. Porém, um novo termo muito importante quanto à manutenção e à evolução dos softwares foi citado no modelo representado na Figura 1: engenharia reversa.

A engenharia reversa é o processo de reconstrução do software que parte do princípio de recuperação do código para tornar compreensíveis suas funcionalidades. Assim, elas podem ser reescritas com vistas a serem otimizadas e corrigidas (Pressman, 2021).

É evidente que a aplicação da engenharia reversa em hardware torna a técnica visualmente mais acessível, pois podemos mentalmente desmontar os componentes, o que facilita a compreensão da ordem de montagem. Além disso, podemos analisar individualmente cada componente para compreender seu propósito e funcionamento.

No entanto, como a engenharia reversa é abordada no contexto do software? Antes de responder a essa pergunta, é instrutivo observar a Figura 3.

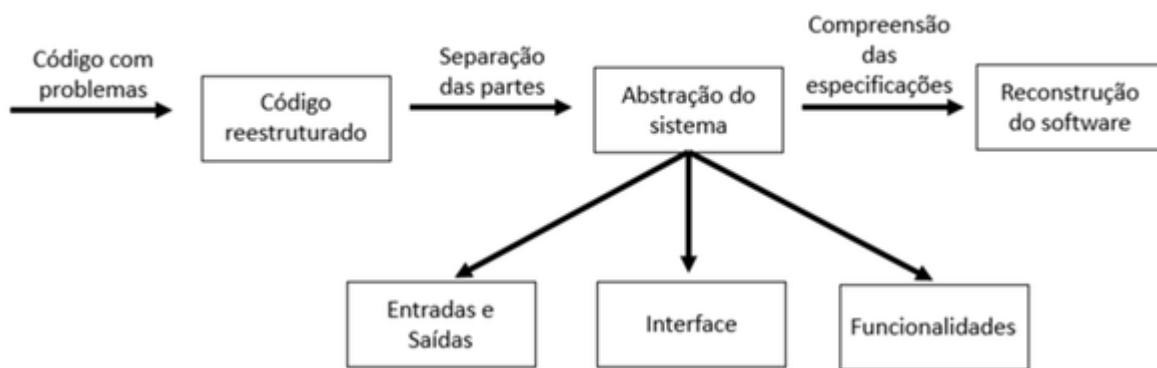


Figura 3 | Processo de engenharia reversa. Fonte: adaptada de Pressman (2021).

Observe que os métodos de engenharia reversa gradualmente se concentram na reestruturação do código, facilitando a compreensão das funcionalidades, incluindo entradas e saídas, interface e operações.

De acordo com Pádua (2019), ao empregar os processos e as técnicas de engenharia reversa, é possível examinar o software de diversas perspectivas, como:

- **Nível de implementação:** possibilita entender as características e as peculiaridades da linguagem de programação utilizada na implementação.
- **Nível estrutural:** permite compreender os diferentes módulos, funcionalidades e suas interdependências funcionais, analisando a estrutura da linguagem de programação empregada.
- **Nível funcional:** facilita a compreensão das partes que constituem o sistema, com foco na lógica aplicada durante o desenvolvimento.
- **Nível de domínio:** oferece insights sobre os ambientes em que o software é aplicado.

Dessa forma, em termos profissionais, tanto a reengenharia quanto a engenharia reversa demandam técnicas relativamente simples, porém, desafiadoras de operacionalizar. Essas técnicas são frequentemente utilizadas em práticas diárias para reduzir o tempo de desenvolvimento. Como resultado, espera-se melhorias substanciais, uma vez que muitas das funcionalidades já foram desenvolvidas anteriormente.

## Evolução de software

Assim como em todos os processos de software, não há um método padrão único para mudança ou evolução de software. O tipo de evolução mais adequado para um sistema de software depende da natureza do software em questão, dos processos de desenvolvimento de software adotados pela organização e das habilidades das pessoas envolvidas. Para certos tipos de sistemas, como aplicativos móveis, a evolução pode ser um processo informal, em que as alterações são geralmente sugeridas em conversas entre os usuários e os desenvolvedores do sistema. Por outro lado, para sistemas críticos embarcados, a evolução do software pode ser formalizada, com documentação estruturada produzida em cada fase do processo.

Propostas de mudança, tanto formais quanto informais, são impulsionadoras da evolução dos sistemas em todas as organizações. Uma proposta de mudança envolve um indivíduo ou grupo sugerindo modificações e atualizações em um sistema de software existente. Essas propostas podem se originar de requisitos existentes que não foram implementados no sistema original, solicitações de novos requisitos, relatórios de defeitos emitidos por partes interessadas no sistema, ou novas ideias da equipe de desenvolvimento para melhorar o software (Sommerville, 2018). Os processos de identificação de mudanças e evolução do sistema são cíclicos e continuam ao longo de toda a vida útil de um sistema, apresentado na Figura 4.

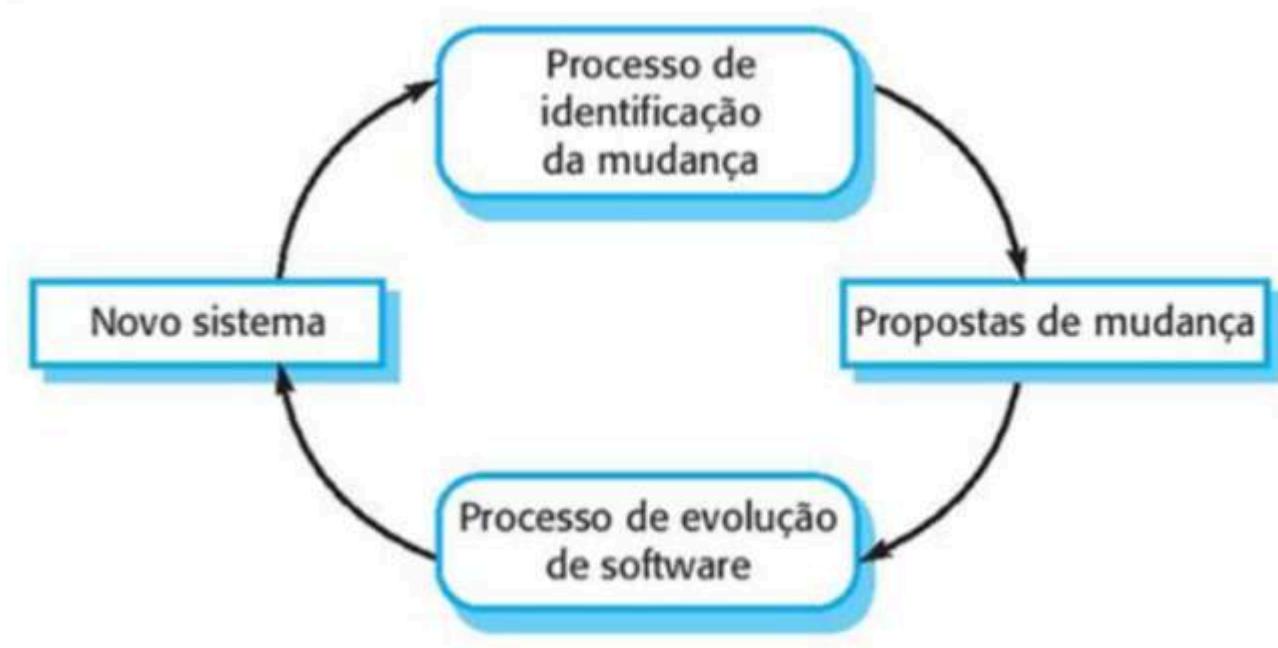


Figura 4 | Processo de identificação da mudança e evolução de software. Fonte: Sommerville (2018).

Antes que uma proposta de mudança seja aceita, é necessário realizar uma análise detalhada do software para determinar quais componentes precisam ser alterados. Essa análise permite a avaliação do custo e do impacto da mudança, sendo parte integrante do processo global de gerenciamento de mudanças. Esse processo também visa garantir que as versões corretas dos componentes sejam incluídas em cada lançamento do sistema.

A Figura 5 ilustra algumas das atividades envolvidas na evolução do software. O processo engloba atividades fundamentais, como análise da mudança, planejamento de lançamento, implementação do sistema e lançamento do sistema para os clientes. Durante esse processo, o custo e o impacto das mudanças são cuidadosamente avaliados para determinar em que medida o sistema será afetado pela mudança e qual seria o custo de sua implementação.

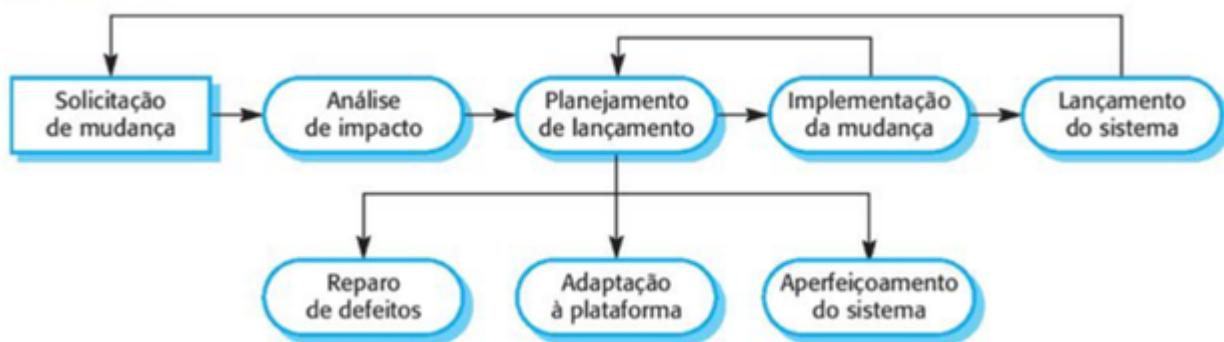


Figura 5 | Modelo geral de processo de evolução de software. Fonte: Sommerville (2018).

Se as mudanças propostas forem aceitas, é iniciado o planejamento de um novo lançamento do sistema, no qual todas as mudanças propostas, sejam elas para reparar defeitos, adaptar ou adicionar novas funcionalidades, são cuidadosamente consideradas. Em seguida, uma decisão é tomada sobre quais mudanças serão implementadas na próxima versão do sistema. Após a implementação e a validação das mudanças, uma nova versão do sistema é lançada. Esse processo é então iterado com um novo conjunto de mudanças propostas para o próximo lançamento (Sommerville, 2018).

Em situações em que o desenvolvimento e a evolução são integrados, a implementação da mudança é simplesmente uma iteração do processo de desenvolvimento. As revisões no sistema são projetadas, implementadas e testadas. A única diferença entre o desenvolvimento inicial e a evolução é que o feedback do cliente após a entrega precisa ser considerado no planejamento dos novos lançamentos de uma aplicação.

## Vamos Exercitar?

Este estudo de caso demonstra a importância da reengenharia e manutenção de software para adaptar sistemas antigos às exigências atuais e futuras. Essa são possíveis sugestões para enfrentar o desafio de um sistema legado de uma biblioteca.

### Objetivos da Reengenharia

- **Atualizar a pilha tecnológica:** migrar para tecnologias modernas e suportadas para facilitar a manutenção e a expansão futura.
- **Refatorar o código:** transformar o código monolítico em uma arquitetura baseada em microserviços para melhorar a modularidade e facilitar a introdução de novas

funcionalidades.

- **Melhorar a interface de usuário:** redesenhar a interface do usuário para melhorar a usabilidade e a acessibilidade, seguindo as práticas atuais de UX/UI.
- **Otimizar o desempenho:** identificar e corrigir gargalos de desempenho para melhorar a eficiência e escalabilidade.
- **Facilitar a integração:** implementar APIs RESTful para facilitar a integração com outros sistemas e serviços.

## Processo de Reengenharia

- **Análise e planejamento:** realização de uma auditoria completa do sistema existente para identificar componentes críticos, funcionalidades e dados a serem preservados ou melhorados.
- **Seleção de tecnologias:** escolha das tecnologias modernas adequadas para a reengenharia do sistema, incluindo frameworks, bancos de dados e ferramentas de desenvolvimento.
- **Desenvolvimento incremental:** aplicação de uma abordagem ágil para o desenvolvimento, permitindo a entrega iterativa de funcionalidades e facilitando o feedback contínuo dos usuários finais.
- **Testes rigorosos:** implementação de testes automatizados para garantir a qualidade do código, a funcionalidade e a segurança do sistema remodelado.
- **Migração de dados:** planejamento e execução de uma estratégia de migração de dados para transferir informações existentes para o novo sistema sem perda de dados.

## Saiba mais

Existem métricas de Manutenção. Para ler mais sobre esse assunto, leia o capítulo 23.5 do livro [Engenharia de Software - Pressman](#).

Quer saber mais sobre Gerenciamento de mudanças? Leia o Capítulo 25.3 de [Engenharia de Software - Sommerville](#).

Para saber mais sobre a Engenharia reversa, leia o Capítulo 27.2.3 do livro [Engenharia de Software - Pressman](#).

## Referências

PÁDUA, W. **Engenharia de Software**. 1<sup>a</sup> Ed. Rio de Janeiro: LTC, 2019.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

VETORAZZO, A. de S. **Engenharia de software**. Porto Alegre: SAGAH, 2018.

## Aula 2

Gestão de Risco

### Gestão de risco

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Na aula de hoje abordaremos os riscos de software, técnicas de identificação e previsão de riscos, além de estratégias de mitigação, monitoramento e gestão de riscos (RMMM). Esses temas são essenciais para a prática profissional na área de desenvolvimento de software, permitindo uma abordagem proativa na identificação e tratamento de potenciais problemas que podem surgir durante o ciclo de vida do projeto. Prepare-se para aprender a lidar com os desafios do gerenciamento de riscos e a garantir o sucesso dos seus projetos de software.

### Ponto de Partida

Nesta aula será abordado os fundamentos dos riscos de software, começando com a sua definição e importância na gestão de projetos de desenvolvimento de software. Em seguida, são discutidas as etapas de identificação e previsão do risco, destacando a necessidade de antecipar e compreender os potenciais problemas que podem surgir ao longo do projeto. A importância dessa fase reside na capacidade de tomar medidas proativas para evitar ou minimizar os impactos negativos dos riscos identificados.

Outro ponto abordado na aula é a mitigação, o monitoramento e a gestão de riscos (RMMM). Essa etapa envolve o desenvolvimento de estratégias para lidar com os riscos, tanto antes quanto durante a execução do projeto. A mitigação visa reduzir a probabilidade de ocorrência de riscos, enquanto o monitoramento é essencial para acompanhar a evolução dos riscos ao longo do tempo e garantir a implementação eficaz das medidas de mitigação. Por fim, a gestão de

riscos envolve a elaboração de planos de contingência para lidar com os riscos que se materializam, garantindo assim a continuidade e o sucesso do projeto.

Esses conceitos são fundamentais para os profissionais de desenvolvimento de software, uma vez que ajudam a garantir a entrega de projetos dentro do prazo, do orçamento e com a qualidade esperada. Ao compreender e aplicar técnicas de identificação, previsão, mitigação, monitoramento e gestão de riscos, os profissionais podem minimizar os impactos negativos dos imprevistos e aumentar as chances de sucesso em seus projetos de software.

Para exercitar esses conceitos, vamos imaginar a seguinte situação: um grande sistema de informações hospitalares estava enfrentando desafios significativos devido a tecnologias obsoletas, problemas de segurança de dados e ineficiências operacionais. A direção do hospital decidiu empreender uma migração completa do sistema para uma plataforma moderna e mais segura. Dada a criticidade dos dados de saúde e a necessidade de manter operações ininterruptas, a gestão de riscos e a aplicação do Modelo de Mitigação de Riscos, Monitoramento e Manutenção (RMMM) tornaram-se fundamentais para o sucesso do projeto.

Desafios identificados:

- **Interrupção das operações hospitalares:** riscos associados à transição de sistemas que poderiam impactar as operações diárias do hospital.
- **Segurança de dados:** potencial de violação de dados durante a migração, afetando a privacidade e a segurança do paciente.
- **Compatibilidade de dados:** desafios na migração de dados existentes para o novo sistema sem perda ou corrupção.
- **Aceitação dos usuários:** resistência do pessoal do hospital à adoção do novo sistema.
- **Sobrecarga de custos e prazos:** riscos de exceder o orçamento e os prazos devido a complicações imprevistas.

Com os conceitos de gestão de riscos e RMMM, proponha objetivos e soluções para minimizar esses desafios identificados.

Bons estudos!

## Vamos Começar!

## Riscos de software

Ao abordar riscos na engenharia de software é importante ter uma preocupação com o futuro, examinando quais riscos podem afetar negativamente o projeto de software. A mudança é outra área de foco, considerando como alterações nos requisitos do cliente, tecnologias de desenvolvimento e outros fatores impactam o andamento e o sucesso do projeto. Além disso, as escolhas são cruciais, exigindo considerações sobre métodos, ferramentas e alocação de recursos para garantir o progresso e a qualidade adequada.

O conceito de débito técnico surge como uma preocupação relacionada aos custos associados ao adiamento de atividades essenciais, como a documentação de software e a refatoração. Ignorar o débito técnico pode resultar em produtos de software com funcionalidades deficientes, comportamentos inconsistentes e documentação inadequada. A analogia com juros financeiros destaca como o débito técnico aumenta ao longo do tempo, ressaltando a importância de resolver problemas técnicos durante fases anteriores do desenvolvimento para evitar complicações futuras.

Embora o desenvolvimento ágil de software seja amplamente adotado, não elimina a necessidade de uma gestão de riscos eficaz. Estudos demonstram que equipes ágeis podem acumular débito técnico devido a demandas crescentes por código e falta de dedicação para reduzi-lo (Sommerville, 2018). Além disso, equipes inexperientes podem produzir mais defeitos e utilizar ferramentas não padronizadas, resultando em processos menos documentados e propensos a repetir erros. No entanto, a conscientização e o planejamento cuidadoso durante sprints definidos podem ajudar a mitigar esses riscos e garantir o sucesso dos projetos.

Embora haja controvérsias sobre a definição precisa de risco de software, existe um consenso geral de que o risco sempre possui duas características essenciais: incerteza e perda. A incerteza implica que um risco pode ou não se concretizar, não existindo riscos com uma probabilidade de 100%. Já a perda refere-se às consequências indesejadas que ocorrerão se o risco se materializar. Ao analisar os riscos, é crucial quantificar tanto o nível de incerteza quanto o grau de perda associado a cada um, considerando diferentes categorias de risco.

Os riscos de projeto representam ameaças ao plano do projeto, podendo resultar em atrasos no cronograma e aumento nos custos caso se materializem. Eles abrangem problemas potenciais relacionados ao orçamento, ao cronograma, à equipe, aos recursos e aos requisitos, bem como à complexidade, ao tamanho e à incerteza estrutural do projeto. Por outro lado, os riscos técnicos ameaçam a qualidade e a entrega do software, dificultando ou impossibilitando sua implementação. Eles englobam problemas relacionados ao projeto, à implementação, à interface, à verificação, à manutenção, às especificações ambíguas, à incerteza técnica e à tecnologia obsoleta (Pressman, 2021).

Os riscos de negócio colocam em risco a viabilidade do software a ser desenvolvido e frequentemente afetam o projeto ou o produto como um todo. Eles incluem criar um produto que não atende às demandas do mercado, não se alinha à estratégia de negócios da empresa, não é compreendido pela equipe de vendas, perde o apoio da alta gerência devido a mudanças organizacionais ou enfrenta restrições orçamentárias. Identificar e gerenciar esses riscos é crucial para o sucesso de um projeto de software.

É crucial ressaltar que uma simples classificação de riscos nem sempre é eficaz, pois alguns riscos são simplesmente impossíveis de serem previstos. Os riscos conhecidos são aqueles que podem ser identificados por meio de uma avaliação cuidadosa do plano do projeto, do ambiente técnico e comercial envolvido, e de outras fontes confiáveis de informação. Por exemplo, datas de entrega irrealistas, falta de documentação de requisitos ou escopo do software e ambientes de desenvolvimento inadequados são considerados riscos conhecidos. Os riscos previsíveis, por sua vez, são derivados da experiência em projetos anteriores, como a rotatividade da equipe,

comunicação deficiente com o cliente e diluição do esforço da equipe à medida que novas solicitações de manutenção surgem. Já os riscos imprevisíveis são uma incógnita. Embora possam ocorrer, são extremamente difíceis de serem identificados com antecedência (Pressman, 2021).

## Identificação e previsão do risco

A identificação de riscos é um processo sistemático para especificar ameaças ao plano do projeto, incluindo estimativas, cronogramas e recursos. Ao reconhecer os riscos conhecidos e previsíveis, o gerente de projeto dá o primeiro passo para evitar ou controlar essas ameaças, quando possível. Há dois tipos de riscos: os genéricos, que representam ameaças potenciais a qualquer projeto de software, e os específicos do produto, identificados apenas por aqueles familiarizados com a tecnologia, as pessoas e o ambiente específico do software. Embora os riscos genéricos sejam importantes, são os riscos específicos do produto que frequentemente causam os maiores problemas. Portanto, é crucial dedicar tempo à identificação minuciosa desses riscos.

Uma abordagem para identificar riscos é criar uma lista de verificação de itens de risco, concentrando-se em subconjuntos dos riscos conhecidos e previsíveis. Essa lista pode incluir subcategorias genéricas como o tamanho do produto, o impacto nos negócios, as características dos envolvidos, a definição do processo, o ambiente de desenvolvimento, a tecnologia envolvida e a experiência da equipe. Ao examinar o plano do projeto e a definição do escopo, é possível desenvolver respostas para perguntas como "Quais características especiais deste produto podem ameaçar nosso plano de projeto?" Esta abordagem estruturada ajuda a garantir que os riscos sejam identificados de maneira abrangente e que as estratégias apropriadas possam ser implementadas para mitigá-los.

Mas como podemos determinar se o projeto de software em que estamos envolvidos está enfrentando sérios riscos? As seguintes questões foram elaboradas com base em dados de risco coletados por meio de entrevistas com experientes gerentes de projetos de software em diversas partes do mundo. Essas questões foram organizadas por sua importância relativa para o sucesso do projeto (Pressman, 2021).

- A alta gerência e o cliente estão formalmente comprometidos em apoiar o projeto?
- Os usuários estão genuinamente comprometidos com o projeto e o sistema/produto a ser desenvolvido?
- Os requisitos são claramente compreendidos pela equipe de engenharia de software e pelos clientes?
- Os clientes foram totalmente envolvidos na definição dos requisitos?
- As expectativas dos usuários são realistas?
- O escopo do projeto permanece estável?
- A equipe de engenharia de software possui as habilidades necessárias?
- Os requisitos de projeto estão bem definidos e estáveis?
- A equipe de projeto possui experiência com a tecnologia a ser utilizada?
- O tamanho da equipe de projeto é adequado para a tarefa?

- Todos os clientes e usuários concordam com a importância do projeto e os requisitos do sistema/produto a ser desenvolvido?

Se alguma dessas questões receber uma resposta negativa, medidas de mitigação, monitoramento e gerenciamento devem ser imediatamente implementadas. O nível de risco do projeto está diretamente relacionado ao número de respostas negativas a essas perguntas.

A Força Aérea Americana publicou um livreto contendo diretrizes valiosas para a identificação e gestão de riscos de software. Segundo a abordagem da Força Aérea, o gerente de projeto deve identificar os fatores de risco que afetam os componentes cruciais do software: desempenho, custo, suporte e cronograma. Esses componentes de risco são definidos da seguinte maneira (Pressman, 2021):

- **Risco de desempenho:** refere-se à incerteza sobre se o produto atenderá aos requisitos e será adequado para o uso pretendido.
- **Risco de custo:** indica a incerteza sobre a manutenção do orçamento do projeto.
- **Risco de suporte:** relaciona-se à incerteza sobre a facilidade de correção, adaptação e melhoria do software resultante.
- **Risco de cronograma:** reflete a incerteza sobre a manutenção do cronograma do projeto e a entrega oportuna do produto.

O impacto de cada fator de risco sobre esses componentes é categorizado como negligenciável, marginal, crítico ou catastrófico. Um impacto negligenciável resulta apenas em inconvenientes, com um custo adicional de mitigação muito baixo. Um impacto marginal pode afetar requisitos secundários ou objetivos de missão, mas não compromete o sucesso geral da missão, com um custo um pouco mais elevado, porém, factível (Pádua, 2019). Um impacto crítico afeta diretamente o desempenho do sistema e parte ou todos os requisitos, colocando em risco o sucesso da missão, com um custo significativamente mais alto para mitigação. Por fim, um impacto catastrófico resultaria no fracasso da missão, com um custo de mitigação inaceitável.

A previsão de risco, também conhecida como estimativa de risco, busca classificar cada risco de duas maneiras distintas: (1) a probabilidade de o risco se materializar e (2) as consequências associadas caso o risco se concretize. Em colaboração com outros gerentes e profissionais técnicos, são executadas quatro etapas para a projeção de risco (Pressman, 2021):

1. Estabelecimento de uma escala que represente a probabilidade percebida de um risco.
2. Descrição das possíveis consequências do risco.
3. Avaliação do impacto do risco sobre o projeto e o produto.
4. Análise da precisão geral da projeção de risco para evitar mal-entendidos.

O propósito dessas etapas é abordar os riscos de maneira a definir prioridades, uma vez que nenhuma equipe de software dispõe de recursos para lidar com todos os riscos de forma igualmente rigorosa. Ao priorizar os riscos, é possível direcionar os recursos para áreas em que terão o maior impacto.

Elaborar uma tabela de riscos é uma técnica simples para a projeção de riscos. Um exemplo é apresentado na Figura 1.

Risco	Categoria	Probabilidade	Impacto	RMMM
A estimativa de tamanho pode ser significativamente baixa	PS	60%	2	
Número de usuários maior do que o planejado	PS	30%	3	
Reutilização menor do que a planejada	PS	70%	2	
Os usuários resistem ao sistema	BU	40%	3	
O prazo de entrega será apertado	BU	50%	2	
Financiamento será perdido	CU	40%	1	
O cliente mudará os requisitos	PS	80%	2	
A tecnologia não atingirá as expectativas	TR	30%	1	
Falta de treinamento no uso das ferramentas	DE	80%	3	
Pessoal sem experiência	ST	30%	2	
A rotatividade do pessoal será alta	ST	60%	2	

Valores de impacto:  
 1 – catastrófico  
 2 – crítico  
 3 – marginal  
 4 – negligenciável

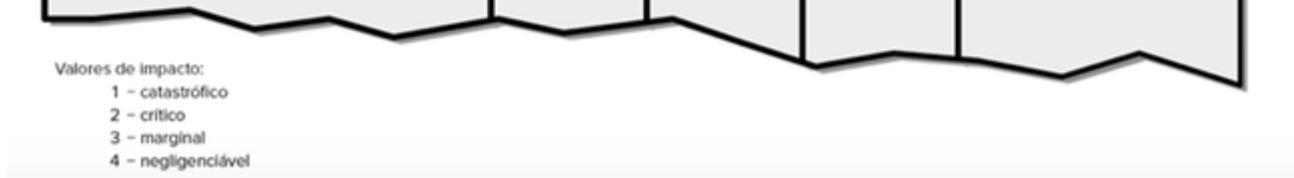


Figura 1 | Exemplo de tabela de riscos antes da ordenação. Fonte: Pressman (2021).

O processo de priorização de riscos inicia-se com a listagem de todos os riscos na primeira coluna de uma tabela, independentemente de sua probabilidade de ocorrência. Em seguida, cada risco é caracterizado na segunda coluna, indicando sua natureza, como risco de tamanho do projeto ou risco de negócio (Pádua, 2019). A probabilidade de cada risco é estimada pelos membros da equipe e registrada na terceira coluna da tabela. Posteriormente, o impacto de cada risco é avaliado com base em componentes como desempenho, custo e cronograma, e é determinada uma categoria de impacto para cada um.

Após o preenchimento das primeiras quatro colunas, a tabela é ordenada por probabilidade e impacto, priorizando os riscos de alta probabilidade e alto impacto. O gerente de projetos estabelece uma linha de corte na tabela, acima da qual todos os riscos devem ser gerenciados. As estratégias de mitigação, monitoramento e gestão de risco são então definidas no plano RMMM ou em formulários de informações específicos para cada risco, discutidos em seções subsequentes (Pressman, 2021).

Embora a probabilidade dos riscos possa ser determinada inicialmente por estimativas individuais e, posteriormente, por consenso, abordagens mais avançadas, como o uso de lógica difusa, têm sido desenvolvidas para avaliar a probabilidade de riscos mais complexos e inter-relacionados em projetos de software. Essas técnicas buscam compreender melhor o impacto combinado dos riscos, especialmente em situações em que a incerteza é elevada e múltiplos fatores estão envolvidos.

**Siga em Frente...**

## Mitigação, monitoramento e gestão de riscos (RMMM)

Todas as etapas de análise de risco até agora têm um objetivo único: auxiliar a equipe de projeto na formulação de uma estratégia para lidar com os riscos. Uma estratégia eficiente precisa abordar três aspectos essenciais: como evitar, monitorar e gerenciar os riscos, além de planejar a contingência (Sommerville, 2018).

Quando a equipe adota uma postura proativa em relação aos riscos, a prevenção se torna a estratégia primordial. Isso envolve a criação de um plano de mitigação de riscos. Por exemplo, se a alta rotatividade de pessoal é identificada como um risco significativo, medidas para reduzi-lo podem incluir investigar suas causas, abordá-las antes do início do projeto e desenvolver técnicas para garantir a continuidade mesmo em caso de saída de membros da equipe.

Durante o projeto, é fundamental antecipar a possibilidade de rotatividade e implementar práticas para minimizar seu impacto. Isso pode incluir a disseminação ampla das informações entre as equipes, o estabelecimento de padrões para os produtos do projeto e a designação de substitutos para membros críticos da equipe, garantindo assim a continuidade das operações (Pressman, 2021).

À medida que o projeto avança, iniciam-se as atividades de monitoramento de risco, nas quais o gerente de projeto acompanha fatores que indicam a evolução da probabilidade de ocorrência dos riscos. No caso da alta rotatividade de pessoal, são observados aspectos como a dinâmica da equipe, as relações interpessoais, questões salariais e a disponibilidade de oportunidades externas.

Além disso, é fundamental monitorar a eficácia das medidas de mitigação adotadas. Por exemplo, se foram estabelecidos padrões para os produtos, é necessário garantir que cada um seja autossuficiente e contenha as informações essenciais para integrar novos membros à equipe, se necessário.

Em situações em que as medidas de mitigação falham e o risco se materializa, entra em vigor a gestão de riscos e o plano de contingência. Por exemplo, se um grupo de pessoas anuncia sua saída durante o projeto, é crucial contar com substitutos disponíveis, documentação adequada e transferência de conhecimento para garantir a continuidade das operações e minimizar o impacto da transição (Pressman, 2021).

As etapas de mitigação, monitoramento e gestão de risco, embora essenciais, implicam custos adicionais para o projeto. Por exemplo, criar backups para tecnologias críticas consome recursos financeiros e de tempo. Portanto, é crucial realizar uma análise de custo-benefício para determinar se os benefícios das medidas de gerenciamento de risco superam os custos associados à sua implementação. Em alguns casos, pode ser mais sensato continuar monitorando o risco em vez de mitigá-lo, especialmente se os custos de mitigação forem consideráveis em relação à exposição ao risco.

Ao identificar e avaliar os riscos de um grande projeto, é importante aplicar a regra de Pareto, conhecida como regra 80-20. Esta sugere que a maior parte do risco geral do projeto está associada a uma minoria dos riscos identificados. Portanto, é fundamental concentrar os esforços de gerenciamento nos riscos mais críticos, que representam 20% dos riscos identificados. Nem todos os riscos identificados precisam ser incluídos no plano de mitigação, pois muitos podem não contribuir significativamente para a exposição ao risco.

Para incentivar os desenvolvedores a seguir procedimentos de conformidade com processos, como qualidade e gestão de riscos, a gamificação tem sido proposta como uma abordagem eficaz. Isso envolve conceder recompensas não monetárias, como pontos ou medalhas, aos membros da equipe por seguir as melhores práticas. No entanto, é crucial garantir que essa abordagem não encoraje comportamentos prejudiciais, como introduzir problemas no processo para resolvê-los posteriormente e ganhar recompensas. Além disso, os riscos associados ao software não se limitam ao desenvolvimento; é necessário considerar também os riscos após a entrega do software ao cliente, como problemas de segurança e imprevistos que podem afetar negativamente o sistema.

## O plano RMMM

A estratégia de gestão de risco pode ser integrada ao plano geral do projeto de software ou ser delineada em um plano separado de mitigação, monitoramento e gestão (RMMM). O plano RMMM registra todas as atividades realizadas durante a análise de risco e serve como parte integrante do plano global do projeto.

Algumas equipes de software optam por não elaborar um documento formal RMMM. Em vez disso, cada risco é registrado individualmente por meio de um formulário de informações de risco (RIS). Esses formulários são frequentemente mantidos em um sistema de banco de dados na nuvem para facilitar a inserção de dados, a priorização, as análises e o compartilhamento entre as equipes de software. Essa abordagem pode favorecer a implementação da gamificação no processo de gestão de riscos e facilitar a colaboração entre as equipes de software da organização. O formato do RIS é exemplificado no Quadro 1:

Formulário de informações de risco			
ID do risco: P02-4.32	Data: 09/05/19	Prob: 80%	Impacto: alto
<b>Descrição:</b> Somente 70% dos componentes de software programados para reutilização serão, de fato, integrados na aplicação. A funcionalidade restante terá de ser desenvolvida de maneira personalizada.			
<b>Refinamento/contexto:</b> <b>Subcondição 1:</b> Certos componentes reutilizáveis foram desenvolvidos por uma equipe terceirizada que não conhecia os padrões internos de projeto.			

**Subcondição 2:** O padrão de projeto para as interfaces do componente não foi completamente estabelecido e pode não estar em conformidade com certos componentes reutilizáveis existentes.

**Subcondição 3:** Certos componentes reutilizáveis foram implementados em uma linguagem não suportada no ambiente em que serão usados.

**Mitigação/monitoramento:**

1. Contate a equipe terceirizada para determinar a conformidade com os padrões de projeto.
2. Pressione para que haja padronização da interface; considere a estrutura de componente ao decidir sobre o protocolo de interface.
3. Determine o número de componentes que estão na categoria da subcondição; determine se pode ser adquirido o suporte de linguagem.

**Gerenciamento/plano de contingência/disparo:**

Foi calculada a exposição ao risco: US\$ 20.200. Reserve esse valor no custo de contingência do projeto. Desenvolva um cronograma revisado assumindo que 18 componentes adicionais terão de ser criados de forma personalizada; defina e adote um padrão de interface consistente. Estarão disponíveis as provisões para mitigação imprevistas em 01/07/19.

**Estado atual:**

12/05/19: Providências para mitigação iniciadas.

Autor. D. Gagne

Designado: B. Laster

Quadro 1 | Formulário de informações de risco. Fonte: adaptado de Pressman (2021).

Após a documentação do RMMM e o início do projeto, entram em cena as etapas de mitigação e monitoramento de risco. Como já explorado, a mitigação de risco visa evitar problemas, enquanto o monitoramento de risco é uma atividade contínua durante o projeto, com três objetivos principais: (1) avaliar se os riscos antecipados realmente se materializam; (2) garantir a implementação adequada das medidas de mitigação definidas para cada risco; e (3) reunir informações úteis para análises de riscos futuras. Em muitas situações, os problemas que surgem ao longo do projeto podem estar associados a diversos riscos. Portanto, outra função crucial do monitoramento de risco é tentar identificar a causa raiz, isto é, quais riscos contribuíram para a ocorrência de determinados problemas durante o projeto (Pressman, 2021).

## Vamos Exercitar?

Este estudo de caso destaca a importância do RMMM na gestão de projetos complexos e de alta stakes, como a migração de sistemas de informações hospitalares. Ao identificar, analisar, mitigar e monitorar continuamente os riscos, a equipe de projeto foi capaz de navegar com sucesso pelos desafios, garantindo a segurança dos dados dos pacientes e a continuidade das

operações hospitalares. A abordagem RMMM provou ser uma ferramenta valiosa na garantia do sucesso do projeto e na manutenção da confiança das partes interessadas.

## Objetivos do RMMM

- **Identificação e análise de riscos:** mapear todos os riscos potenciais e avaliar sua probabilidade e impacto.
- **Desenvolvimento de estratégias de mitigação:** criar planos de ação específicos para reduzir, evitar ou transferir riscos.
- **Implementação de monitoramento contínuo:** estabelecer processos para monitorar o status dos riscos e a eficácia das estratégias de mitigação ao longo do projeto.
- **Manutenção e ajustes:** adaptar e refinar estratégias de mitigação com base no feedback do monitoramento contínuo.

## Processo de Implementação do RMMM

### 1. Fase de Planejamento:

- **Identificação de riscos:** realização de sessões de brainstorming com as partes interessadas para identificar riscos.
- **Análise de riscos:** utilização de técnicas como análise qualitativa e quantitativa para priorizar riscos.

### 2. Desenvolvimento de estratégias de mitigação:

- Para **interrupções operacionais**, foi planejada uma implementação faseada, com migração durante horas de menor atividade.
- Em relação à **segurança de dados**, adotaram-se protocolos rigorosos de criptografia e transferência segura.
- Para enfrentar a **compatibilidade de dados**, realizou-se uma validação cruzada intensiva dos dados antes da migração final.
- Para a **aceitação dos usuários**, foram organizadas sessões de treinamento e suporte contínuo.
- Sobre **sobrecarga de custos e prazos**, estabeleceu-se um fundo de contingência e um cronograma flexível.

### 3. Monitoramento contínuo:

- Implementação de um sistema de relatórios periódicos e checkpoints de projeto para avaliar o progresso e identificar novos riscos.

### 4. Manutenção e ajustes:

- Ajustes nas estratégias de mitigação com base no monitoramento contínuo e no feedback das partes interessadas.

## Resultados

A aplicação do RMMM permite à equipe de projeto gerenciar efetivamente os riscos durante a migração do sistema de informações hospitalares. A implementação faseada minimiza as interrupções operacionais, e as estratégias de segurança de dados garantiram uma transição segura dos registros dos pacientes.

## Saiba mais

Para ler mais sobre a Gestão de riscos, leia o Capítulo 26 do livro [Engenharia de Software - Pressman](#).

Quer saber mais sobre o gerenciamento de riscos? Leia o Capítulo 22.1 de [Engenharia de Software - Sommerville](#).

Veja no artigo [Gerência de riscos em desenvolvimento de software](#), os aspectos e impactos da utilização de métodos relacionados a uma área específica da engenharia de software, a Gerência de riscos.

## Referências

PÁDUA, W. **Engenharia de software**. 1. ed. Rio de Janeiro: LTC, 2019.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Aula 3

Métricas e Análise de Software

### Métricas e análise de software

**Este conteúdo é um vídeo!**

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo



para assistir mesmo sem conexão à internet.

#### Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Na aula de hoje vamos explorar a medição de software, incluindo métricas de produtos, processos e projetos. Esses conceitos são fundamentais para a prática profissional na área de desenvolvimento de software, permitindo avaliar a qualidade, o desempenho e a eficiência dos produtos desenvolvidos, bem como dos processos e os projetos envolvidos em sua criação. Prepare-se para adquirir conhecimentos valiosos sobre como medir e melhorar continuamente a sua prática profissional na engenharia de software.

## Ponto de Partida

Na aula de hoje iremos explorar os conceitos fundamentais relacionados à medição de software e ao uso de métricas para avaliar diferentes aspectos dos projetos de desenvolvimento. A medição de software é essencial para quantificar características como tamanho, complexidade e desempenho dos sistemas em desenvolvimento. As métricas de processos e projetos são cruciais para monitorar o progresso do projeto, identificar áreas de melhoria e garantir que os objetivos sejam alcançados dentro do prazo e do orçamento estabelecidos. Por fim, estabelecer um programa de métricas auxilia a ter um comprometimento com essa prática.

Compreender a medição de software e o uso de métricas é fundamental para o sucesso de qualquer projeto de desenvolvimento de software. Ao quantificar diferentes aspectos do software e do processo de desenvolvimento, as equipes podem identificar áreas de risco, tomar decisões informadas e implementar melhorias contínuas ao longo do ciclo de vida do projeto. As métricas fornecem uma base objetiva para avaliar o progresso, tomar decisões estratégicas e garantir que os padrões de qualidade sejam atendidos, resultando em produtos finais mais eficientes e confiáveis.

Ao integrar a medição de software e o uso de métricas em seus processos de desenvolvimento, as organizações podem melhorar a eficiência, reduzir custos e minimizar riscos. As métricas fornecem dados tangíveis que ajudam as equipes a identificar tendências, implementar melhores práticas e tomar medidas corretivas quando necessário. Em última análise, a medição de software e o uso de métricas capacitam as equipes a realizar entregas de alta qualidade de forma consistente, atendendo às necessidades dos clientes e alcançando os objetivos organizacionais.

Para sintetizar o conteúdo, vamos imaginar que você precise alinhar as operações da engenharia de software com os objetivos estratégicos de uma empresa de tecnologia. A primeira etapa envolve entender profundamente as aspirações comerciais da organização. O que a empresa está tentando alcançar? Com essa compreensão, você será capaz de determinar os

conhecimentos que são necessários para apoiar essas aspirações. Apresente o motivo que uma implementação de um programa de métricas de software é importante para esta empresa.

Bons estudos!

## Vamos Começar!

### Medição de software

A medição é um componente essencial em qualquer processo de engenharia. No contexto do desenvolvimento de software, as medições desempenham um papel fundamental na melhoria contínua do processo. Elas são aplicadas ao longo do ciclo de vida do projeto para diversas finalidades, como aprimoramento das estimativas, garantia da qualidade, aumento da produtividade e controle do projeto. As medidas fornecem insights valiosos sobre os atributos dos modelos desenvolvidos e permitem a avaliação da qualidade dos produtos ou sistemas construídos.

Os engenheiros de software utilizam medições para avaliar a qualidade dos artefatos e apoiar decisões táticas durante o progresso do projeto. No entanto, ao contrário de outras disciplinas de engenharia, a engenharia de software não se baseia em leis quantitativas da física, o que torna as medidas e métricas indiretas e sujeitas a debate.

No contexto do desenvolvimento de software e dos projetos associados, a equipe se concentra principalmente em métricas de produtividade e qualidade. Isso inclui medir a saída do desenvolvimento em relação ao esforço e ao tempo aplicados, bem como a adequação dos artefatos produzidos para uso (Pádua, 2019). Para planejamento e estimativa, o foco está no histórico de produtividade e qualidade dos projetos anteriores, buscando extrapolar esses dados para o presente.

A medição serve tanto como ferramenta técnica quanto de gerenciamento, proporcionando maior entendimento ao gerente de projeto e ajudando na tomada de decisões para o sucesso do projeto. Este capítulo apresenta medidas para avaliar a qualidade do produto durante o processo de desenvolvimento e para gerenciar projetos de software, oferecendo uma visão em tempo real da eficácia dos processos e da qualidade geral do software em criação.

Embora os termos medida, medição e métricas sejam frequentemente utilizados de forma intercambiável, é essencial observar as sutis diferenças entre eles. Quando um único dado é coletado, como o número de erros descobertos em um componente de software, uma medida é estabelecida. A medição ocorre quando são coletados um ou mais pontos de dados, como ao investigar um conjunto de revisões de componente e testes de unidade para coletar medidas do número de erros em cada um. Por sua vez, uma métrica de software relaciona as medidas individuais de alguma maneira, como o número médio de erros encontrados por revisão ou por teste de unidade.

Os engenheiros de software coletam medidas e desenvolvem métricas para obter indicadores. Um indicador é uma métrica ou uma combinação delas que fornecem informações sobre o processo de software, em um projeto de software ou no produto em si (Pádua, 2019).

Já foram propostas centenas de métricas para programas de computadores, porém, nem todas são viáveis para os engenheiros de software. Algumas exigem medições complexas, outras são tão abstratas que poucos profissionais do mundo real conseguem compreendê-las, e outras ainda contradizem as noções básicas do que constitui um software de alta qualidade (Pressman, 2021). A experiência mostra que uma métrica só será amplamente adotada se for clara e fácil de calcular. Se demandar dezenas de contagens e cálculos complexos, é improvável que seja amplamente aceita.

Um conjunto de atributos que devem ser contemplados por métricas de software eficazes. Elas devem ser relativamente fáceis de aprender a derivar, sem exigir esforço ou tempo excessivo. Além disso, devem estar alinhadas às noções intuitivas dos engenheiros sobre o atributo do produto considerado. Por exemplo, uma métrica que avalia a coesão de módulos deve aumentar à medida que a coesão aumenta. As métricas também devem produzir resultados inequívocos e evitar combinações de unidades bizarras durante o cálculo matemático. Devem ser baseadas no modelo de requisitos, no modelo de projeto ou na estrutura do programa, sem depender das nuances sintáticas ou semânticas das linguagens de programação. Por fim, as métricas devem fornecer informações que possam contribuir para um produto de melhor qualidade.

## Métricas de processos e de projetos

As métricas de processo são coletadas ao longo de todos os projetos e durante períodos prolongados, visando fornecer um conjunto de indicadores que contribuem para aprimorar o processo de software em longo prazo. Por outro lado, as métricas de projeto permitem que o gerente de projeto de software avalie o progresso de um projeto em andamento, identifique potenciais riscos, detecte áreas problemáticas antes que se tornem críticas, ajuste o fluxo de trabalho ou tarefas e avalie a capacidade da equipe de controlar a qualidade dos artefatos de software.

As medidas coletadas pela equipe de projeto e transformadas em métricas para uso durante o projeto também podem ser compartilhadas com os responsáveis pelo aprimoramento do processo de software. Portanto, muitas dessas métricas são empregadas tanto no âmbito do processo quanto no do projeto. Ao contrário das métricas de processo de software, que têm finalidade estratégica, as métricas de projeto de software têm caráter tático. Em outras palavras, as métricas de projeto e os indicadores derivados delas são utilizados pelo gerente de projeto e pela equipe de software para ajustar o fluxo de trabalho do projeto e as atividades técnicas.

Aprimorar qualquer processo requer uma abordagem lógica que envolve a medição de atributos específicos, o desenvolvimento de métricas significativas com base nesses atributos e o uso dessas métricas para orientar estratégias de aprimoramento. No entanto, antes de explorarmos as métricas de software e seu impacto na melhoria do processo de software, é crucial reconhecer que o processo é apenas um dos vários elementos que influenciam a qualidade do

software e o desempenho organizacional. Conforme ilustrado na Figura 1, o processo está no cerne do triângulo que interconecta três fatores-chave que exercem profunda influência sobre a qualidade do software e o desempenho organizacional. Estudos anteriores demonstraram que a habilidade e a motivação das pessoas representam os fatores mais influentes na qualidade e no desempenho. Além disso, a complexidade do produto pode ter um impacto significativo sobre a qualidade e o desempenho da equipe, enquanto a tecnologia – ou seja, os métodos e ferramentas de engenharia de software – que sustenta o processo também desempenha um papel crucial.

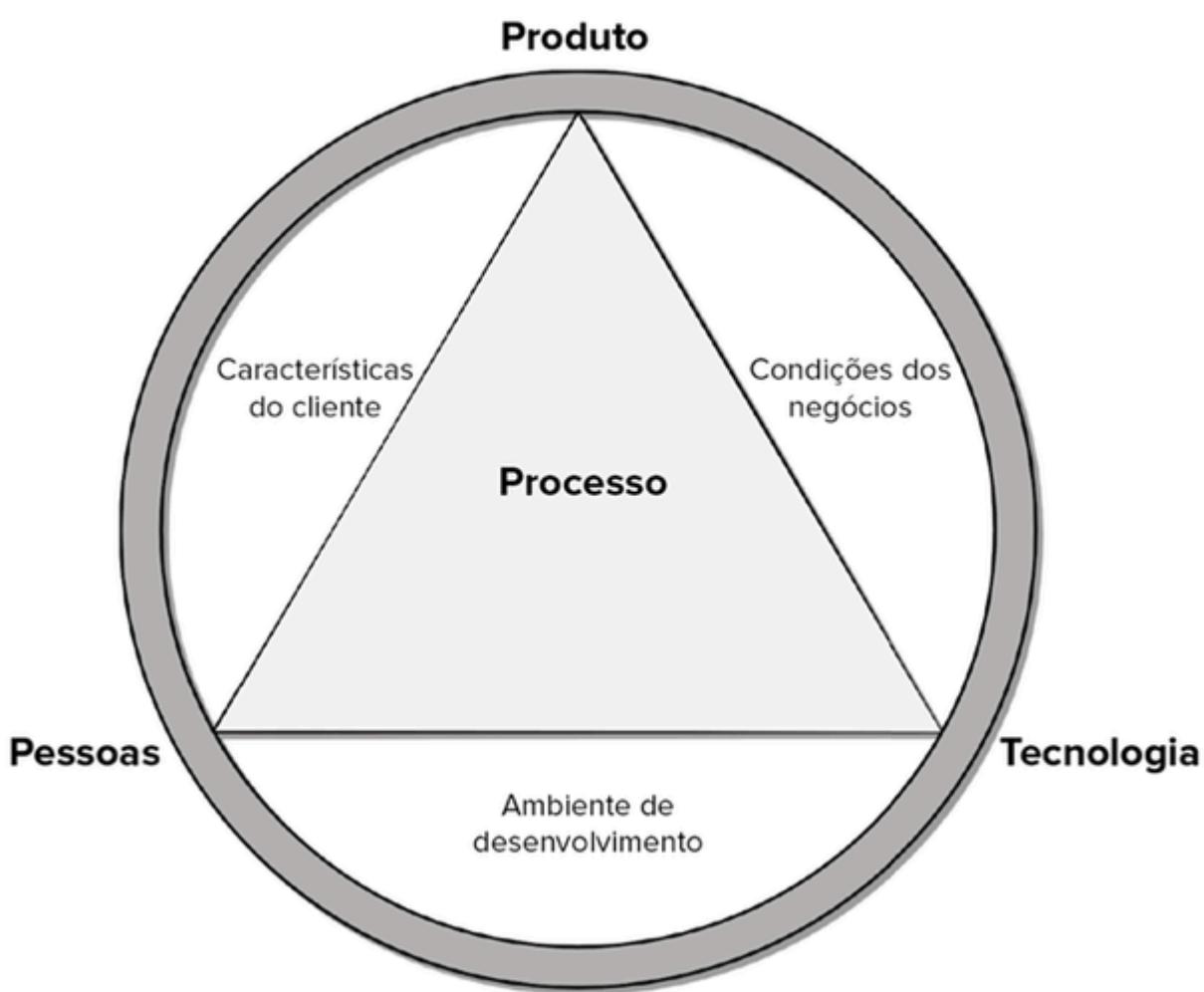


Figura 1 | Determinantes para a qualidade do software e a eficácia organizacional. Fonte: Pressman (2021).

Além disso, o triângulo do processo é situado dentro de um contexto ambiental mais amplo, que engloba o ambiente de desenvolvimento (por exemplo, ferramentas de software integradas), as condições de negócios (como prazos de entrega e regras do negócio) e as características do cliente (incluindo facilidade de comunicação e colaboração) (Pressman, 2021).

A avaliação da eficácia de um processo de software ocorre de forma indireta. Isso significa que um conjunto de métricas é desenvolvido com base nos resultados alcançados pelo processo.

Esses resultados incluem medidas como erros identificados antes da entrega do software, defeitos relatados pelos usuários finais, produtividade (medida pelos artefatos entregues), esforço humano empregado, tempo utilizado, aderência ao cronograma e outras variáveis relevantes. Também é possível obter métricas de processo ao medir características específicas das tarefas de engenharia de software.

Normalmente, as métricas de projeto são aplicadas pela primeira vez durante o processo de estimativa em projetos de software. Métricas provenientes de projetos anteriores servem como base para estimativas de esforço e tempo para o trabalho de software atual. À medida que o projeto avança, as métricas de esforço e tempo são comparadas com as estimativas iniciais e com o cronograma do projeto. O gerente de projeto utiliza esses dados para monitorar e controlar o progresso do projeto.

À medida que o trabalho técnico é iniciado, outras métricas de projeto tornam-se relevantes. Isso inclui a medição das taxas de produção, representadas em modelos criados, revisões, pontos de função e linhas de código-fonte entregues. Além disso, são rastreados os erros identificados durante cada etapa da engenharia de software. Conforme o software evolui dos requisitos para o projeto, métricas técnicas são coletadas para avaliar a qualidade do projeto e fornecer indicadores que orientam a estratégia adotada para a geração de código e os testes (Pressman, 2021).

O propósito das métricas de projeto é duplo. Primeiramente, são empregadas para minimizar o cronograma de desenvolvimento, realizando os ajustes necessários para evitar atrasos e mitigar possíveis problemas e riscos. Em segundo lugar, as métricas de projeto são utilizadas para avaliar continuamente a qualidade do produto e, se necessário, adaptar a abordagem técnica para aprimorar sua qualidade.

Conforme a qualidade do produto se aprimora, a incidência de defeitos é reduzida, o que, por sua vez, diminui a necessidade de retrabalho durante o projeto. Isso resulta em uma diminuição no custo global do projeto.

As métricas de processo de software podem gerar benefícios substanciais quando uma organização busca melhorar seu nível geral de maturidade de processo. No entanto, assim como ocorre com todas as métricas, seu uso inadequado pode gerar mais problemas do que soluções (Pressman, 2021).

Uma sugestão de práticas adequadas para gerentes e profissionais ao implementar um programa de métricas de processo é apresentada como uma "etiqueta de métricas de software", descrito a seguir (Pressman, 2021):

- Use discernimento e sensibilidade organizacional ao analisar os dados das métricas.
- Forneça feedback regularmente às pessoas e equipes responsáveis pela coleta de medidas e métricas.
- Evite utilizar métricas para avaliar o desempenho individual.
- Colabore com profissionais e equipes para estabelecer metas claras e as métricas a serem utilizadas para alcançá-las.

- Evite ameaçar indivíduos ou equipes com base em métricas.
- Não interprete dados de métricas que apontam áreas problemáticas como "negativos".  
Esses dados são apenas indicativos de áreas que requerem melhoria no processo.
- Evite fixar-se excessivamente em uma única métrica, negligenciando outras métricas importantes.

À medida que uma organização se torna mais familiarizada com a coleta e aplicação de métricas de processo, a transição para uma abordagem mais rigorosa pode ocorrer, conhecida como Melhoria Estatística de Processo de Software (SSPI, do inglês *Statistical Software Process Improvement*). Essencialmente, a SSPI emprega a análise de falhas de software para reunir informações sobre todos os erros e defeitos encontrados durante o desenvolvimento e uso de uma aplicação, sistema ou produto (Pressman, 2021).

## Siga em Frente...

## Estabelecimento de um programa de métricas de software

O *Software Engineering Institute* (SEI) elaborou um guia abrangente [Par96b] para estabelecer um programa de métricas de software com foco em metas. Este livro sugere as seguintes etapas (Pressman, 2021):

1. Identificar as metas de negócio.
2. Identificar o que se deseja saber ou aprender.
3. Identificar as submetas.
4. Identificar as entidades e atributos relacionados às submetas.
5. Formalizar as metas de medição.
6. Identificar questões quantificáveis e os indicadores associados que serão utilizados para alcançar as metas de medição.
7. Identificar os elementos de dados a serem coletados para construir os indicadores.
8. Identificar as medidas a serem utilizadas e definir suas operacionalizações.
9. Identificar as ações a serem tomadas para implementar as medidas.
10. Elaborar um plano para a implementação das medidas.

Uma discussão detalhada dessas etapas está disponível no guia do SEI. No entanto, o exemplo a seguir oferece uma visão geral sucinta dos pontos principais.

Dado que o software desempenha funções críticas para o negócio, seja suportando operações, diferenciando sistemas ou produtos baseados em computadores, ou atuando como um produto em si mesmo, os objetivos estabelecidos para os negócios podem ser diretamente relacionados a metas específicas no nível da engenharia de software. Em colaboração entre a equipe de engenharia de software e os gestores de negócios, uma lista de metas comerciais com prioridades é desenvolvida (Pressman, 2021):

- Melhorar a satisfação de nossos clientes com nossos produtos.

- Tornar os seus produtos mais fáceis de usar.
- Reduzir o tempo necessário para ter um novo produto no mercado.
- Tornar o suporte aos nossos produtos mais fáceis.
- Melhorar nossa lucratividade geral.

A empresa de software examina cada meta de negócio e busca compreender: "Quais são as atividades que gerenciamos, executamos ou suportamos e o que queremos melhorar nessas atividades?". Para responder a essas indagações, o SEI recomenda a elaboração de uma "lista entidade-questão", na qual são enumeradas todas as áreas (entidades) dentro do processo de software que são gerenciadas ou influenciadas pela organização de software. Exemplos de entidades incluem recursos de desenvolvimento, artefatos, código-fonte, casos de teste, solicitações de alteração, tarefas de engenharia de software e cronogramas. Para cada entidade listada, os profissionais de software desenvolvem uma série de questões que exploram características quantitativas da entidade (por exemplo, tamanho, custo, tempo de desenvolvimento). As questões resultantes da criação da lista entidade-questão levam à definição de uma série de submetas diretamente relacionadas às entidades identificadas e às atividades realizadas como parte do processo de software.

Considere a quarta meta: "Tornar o suporte aos nossos produtos mais fácil". Para essa meta, pode ser criada a seguinte lista de questões (Pressman, 2021):

- As solicitações de alterações do cliente contêm as informações de que precisamos para avaliar adequadamente a alteração e então implementá-la dentro do prazo normal?
- Qual é o acúmulo de solicitações de alteração?
- Nosso tempo de resposta para corrigir os erros é aceitável com base nas necessidades do cliente?
- Nosso processo de controle de alterações é seguido?
- As alterações de alta prioridade são implementadas dentro do prazo normal?

A partir dessas questões, a organização de software pode derivar a seguinte submeta: Aperfeiçoar o desempenho do processo de gerenciamento de alterações. As entidades e os atributos pertinentes ao processo de software relacionados à submeta são identificados, e são delineadas as metas de medição associadas a eles.

O SEI oferece diretrizes detalhadas para as etapas 6 a 10 de sua estratégia de medição orientada por metas, que envolvem o refinamento das metas de medição em questões, entidades, atributos e finalmente métricas.

Para empresas de desenvolvimento de software com menos de 20 profissionais, desenvolver programas extensos de métricas pode não ser prático. No entanto, é recomendável que organizações de todos os tamanhos implementem medições para melhorar os seus processos e a qualidade de seus produtos. Empresas menores podem começar focalizando nos resultados e estabelecendo objetivos claros para melhorias, como reduzir o tempo para avaliar e implementar solicitações de alterações.

A maioria dos desenvolvedores de software ainda não realiza medições e, infelizmente, muitos não estão dispostos a começar. Tentar introduzir medidas em um ambiente em que elas não foram utilizadas antes muitas vezes encontra resistência. "Por que precisamos fazer isso?" questiona o gerente de projeto apressado. "Não vejo necessidade disso", concorda um programador sobrecarregado. Por que é tão crucial medir o processo de engenharia de software e os artefatos que ele produz? A resposta é bastante clara. Sem medição, não há maneira real de determinar se houve melhorias. E se não houver melhorias, é fácil se perder (Pressman, 2021).

## Vamos Exercitar?

A seguir está uma descrição esperada para a atividade, destacando os conceitos-chave de uma implementação de um programa de métricas de software.

Ao se considerar a implementação de um programa de métricas de software em uma organização, é essencial começar com uma compreensão clara dos objetivos de negócios da empresa. Com esses objetivos em mente, é possível determinar o que se deseja saber ou aprender para apoiar esses objetivos. Isso envolve uma análise detalhada para identificar submetas mais específicas e mensuráveis. Cada submeta está vinculada a entidades e atributos específicos que são críticos para a sua realização, e é vital reconhecer e entender essas conexões.

A próxima fase consiste em formalizar as metas de medição, estabelecendo definições claras de sucesso. Isto requer a identificação de questões específicas que possam ser respondidas por meio de dados e a definição de indicadores correspondentes. Estes indicadores, por sua vez, são baseados em elementos de dados que devem sermeticulosamente coletados e analisados. Com esses dados em mãos, as medidas específicas podem ser identificadas e suas operacionalizações definidas, garantindo que cada medida seja implementada de forma eficaz.

Por último, é crucial planejar as ações necessárias para a implementação das medidas e desenvolver um plano abrangente para guiar o processo de implantação. Este plano não apenas detalha os passos a serem tomados, mas também considera a forma como as medidas serão integradas às operações diárias da empresa.

Concluindo, estabelecer um programa de métricas de software é um processo estratégico que envolve não apenas a identificação de metas e medidas, mas também um planejamento cuidadoso e uma execução considerada. Este processo é essencial para garantir que as funções críticas do software estejam alinhadas com os objetivos mais amplos da empresa, desde melhorar a satisfação do cliente e a usabilidade do produto até acelerar o tempo de lançamento no mercado, facilitar o suporte ao produto e, finalmente, aumentar a lucratividade geral.

## Saiba mais

Para ler mais sobre Métricas de software, leia o Capítulo 23 do livro [Engenharia de Software - Pressman](#).

Quer saber mais sobre a Medição de software? Leia o Capítulo 24.5 de [Engenharia de Software - Sommerville](#)

Leia o artigo [Como medir a qualidade do desenvolvimento de software?](#) E conheça mais sobre esse tema.

## Referências

PÁDUA, W. **Engenharia de software**. 1. ed. Rio de Janeiro: LTC, 2019.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Aula 4

Reuso de Software

### Reúso de software

#### Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Na aula de hoje, vamos explorar o reúso de software, abordando o panorama, os benefícios e os desafios associados a essa prática. Além disso, discutiremos os frameworks de aplicação, ferramentas essenciais para facilitar e promover o reúso de componentes de software em diversos projetos. Dominar esses conceitos é fundamental para aumentar a eficiência e a

produtividade no desenvolvimento de software. Prepare-se para aprender como aproveitar ao máximo o potencial do reúso na sua prática profissional em engenharia de software.

## Ponto de Partida

Nesta aula vamos explorar o conceito fundamental de aproveitar componentes de software existentes em novos projetos, em vez de desenvolvê-los do zero. Discutimos o panorama do reúso, que abrange várias abordagens, como reúso de sistemas, aplicações, componentes e objetos. Essa visão geral permite aos alunos entenderem as diferentes formas de implementar o reúso e identificar as estratégias mais adequadas para seus projetos específicos.

Uma compreensão aprofundada do reúso de software é essencial, pois oferece inúmeras vantagens para o desenvolvimento de software. O reúso permite economizar tempo, esforço e recursos, além de promover a consistência e a qualidade do software produzido. Ao dominar as técnicas de reúso, os profissionais podem acelerar o ciclo de desenvolvimento, reduzir custos e manter um alto padrão de excelência em seus projetos.

Além disso, exploramos os frameworks de aplicação, que são estruturas fundamentais para facilitar o desenvolvimento de software. Esses frameworks oferecem um conjunto de funcionalidades pré-implementadas e padronizadas, permitindo aos desenvolvedores criar aplicações de forma mais eficiente e consistente. Compreender o funcionamento e a aplicação dos frameworks de aplicação é crucial para maximizar a eficácia do reúso de software e promover a produtividade no desenvolvimento de sistemas complexos.

Para aplicarmos esses conceitos, vamos pensar na seguinte situação: você é um engenheiro de software em uma empresa de desenvolvimento de sistemas e foi encarregado de projetar uma plataforma de e-commerce escalável. A empresa deseja reduzir o tempo de desenvolvimento e os custos, mantendo a qualidade e a flexibilidade do software. O reúso de software foi identificado como um componente-chave para alcançar esses objetivos. A arquitetura *Model-View-Controller* (MVC) é proposta para facilitar o reúso e a manutenção do sistema. Com isso, você deve elaborar um relatório discutindo a importância do reúso de software na engenharia de software moderna, com foco na aplicação da arquitetura MVC como um meio de promover o reúso eficiente de software. O seu relatório deve conter os seguintes pontos:

- Explorar a Importância do Reúso de Software.
- Descrever a Arquitetura MVC.
- Relacionar Reúso de Software com MVC.
- Reflexão sobre Reúso Estratégico.

Bons estudos!

## Vamos Começar!

## Reúso de Software

Engenharia de software baseada em reúso é uma estratégia na qual o processo de desenvolvimento é direcionado para aproveitar o software existente. Até aproximadamente o ano 2000, o reúso sistemático de software era pouco comum, porém, hoje é amplamente utilizado no desenvolvimento de novos sistemas de negócios (Hirama, 2011). Essa mudança para o desenvolvimento baseado em reúso foi uma resposta às demandas por redução de custos de produção e manutenção de software, entrega mais rápida de sistemas e maior qualidade de software. As empresas agora veem o software como um ativo valioso e estão promovendo o reúso de sistemas existentes para aumentar o retorno sobre os investimentos em software.

Atualmente, existe uma ampla gama de software reusável disponível em diversos formatos. O movimento de código aberto gerou uma extensa base de código pronta para ser reaproveitada, seja na forma de bibliotecas de programas ou de aplicações completas. Muitos sistemas de aplicação específicos para domínios, como os sistemas ERP, estão prontamente disponíveis e podem ser adaptados para atender aos requisitos do cliente. Algumas empresas de grande porte oferecem uma variedade de componentes reutilizáveis para seus clientes. Além disso, os padrões, como os padrões de web services, têm facilitado o desenvolvimento e o reúso de serviços de software em uma variedade de aplicações.

A abordagem de engenharia de software baseada em reúsobusca maximizar a utilização de software existente. As unidades de software reutilizadas podem variar significativamente em tamanho. Por exemplo (Sommerville, 2018):

- **Reúso de sistemas:** sistemas completos, compostos por uma série de programas de aplicação, podem ser reutilizados como parte de um sistema de sistemas.
- **Reúso de aplicações:** uma aplicação pode ser incorporada sem alterações em outros sistemas ou configurada para atender a diferentes clientes. Alternativamente, famílias de aplicações ou linhas de produtos de software com uma arquitetura comum, adaptadas para requisitos individuais, podem ser usadas para desenvolver um novo sistema.
- **Reúso de componentes:** componentes de uma aplicação, desde subsistemas até objetos únicos, podem ser reutilizados. Por exemplo, um sistema de reconhecimento de padrões desenvolvido como parte de um sistema de processamento de texto pode ser reutilizado em um sistema de gerenciamento de banco de dados. Esses componentes podem ser hospedados na nuvem ou em servidores privados e acessados como serviços por meio de uma API.
- **Reúso de objetos e funções:** componentes de software que implementam uma única função, como uma função matemática, ou uma classe de objeto, podem ser reutilizados. Esse tipo de reúso, centrado em torno de bibliotecas padrões, tem sido comum nas últimas quatro décadas. Muitas bibliotecas de funções e classes estão disponíveis gratuitamente. As classes e as funções nessas bibliotecas são reutilizadas quando vinculadas ao código da aplicação recém-desenvolvida. Em áreas como algoritmos matemáticos e gráficos, em que o conhecimento especializado é necessário para desenvolver objetos e funções eficientes, o reúso é particularmente econômico.

Todos os sistemas de software e componentes que contêm funcionalidades genéricas têm o potencial de serem reutilizáveis. No entanto, em algumas situações, esses sistemas ou componentes podem ser tão específicos que modificá-los para se adequarem a uma nova situação é bastante dispendioso. Nesses casos, em vez de reutilizar o código, é viável reutilizar as ideias subjacentes que deram origem ao software. Esse tipo de reuso é denominado reuso de conceito.

No reuso de conceito, um componente de software em si não é reutilizado; em vez disso, são aproveitadas ideias, métodos de trabalho ou algoritmos. O conceito reutilizado é representado de forma abstrata, como um modelo de sistema, que não contém detalhes de implementação (Hirama, 2011). Assim, ele pode ser configurado e adaptado para diversas situações. O reuso de conceito está integrado a abordagens como a de padrões de design, produtos de sistema configuráveis e geradores de programas. Quando os conceitos são reutilizados, o processo de reuso deve incluir uma etapa na qual esses conceitos abstratos sejam transformados em componentes executáveis.

Uma vantagem evidente do reuso de software é a redução dos custos gerais de desenvolvimento (Pádua, 2019). Há menos componentes de software para especificar, projetar, implementar e validar. No entanto, a redução de custos é apenas um dos benefícios do reuso de software. Outras vantagens são detalhadas no Quadro 1.

Benefício	Explicação
Desenvolvimento acelerado	Lançar um sistema no mercado o mais brevemente possível costuma ser mais importante do que os custos totais do desenvolvimento. O reuso de software pode acelerar a produção do sistema, pois tanto o tempo de desenvolvimento quanto de validação podem ser reduzidos.
Uso eficaz de especialistas	Em vez de refazer o mesmo trabalho repetidamente, os especialistas de aplicação podem desenvolver um software reusável que encapsule o seu conhecimento.
Maior dependabilidade	O software reusado, que foi testado e aprovado em sistemas em funcionamento, deve ser mais confiável do que o novo software. Seus defeitos de projeto e

	implementação devem ter sido descobertos e corrigidos.
Custos de desenvolvimento mais baixos	Os custos de desenvolvimento são proporcionais ao tamanho do software que está sendo desenvolvido. Reusar o software significa que menos linhas de código têm de ser escritas.
Menos risco para o processo	O custo do software existente já é conhecido, enquanto os custos de desenvolvimento são sempre uma incógnita. Esse fator é importante para o gerenciamento do projeto porque reduz a margem de erro na estimativa de custo do projeto. Isso vale especialmente quando grandes componentes de software, como os subsistemas, são reusados.
Conformidade com os padrões	Alguns padrões, como os de interface com o usuário, podem ser implementados como um conjunto de componentes reusáveis. Por exemplo, se os menus em uma interface com o usuário forem implementados usando componentes reusáveis, todas as aplicações apresentam os mesmos formatos de menu para os usuários. O uso de uma interface com o usuário melhora a dependabilidade, porque os usuários cometem menos erros quando é apresentada a eles uma interface conhecida.

Quadro 1 | Benefícios do Reúso de Software. Fonte: adaptado de Sommerville (2018).

No entanto, há custos e desafios associados ao reúso, conforme apresentado no Quadro 2. Existe um custo significativo associado à avaliação da adequação de um componente para reúso em uma determinada situação e à realização de testes para garantir sua confiabilidade. Esses custos adicionais podem resultar em uma economia nos custos de desenvolvimento menor do que o inicialmente previsto. No entanto, os demais benefícios do reúso ainda são válidos.

Problema	Explicação
Criar, manter e usar uma biblioteca de componentes	Povoar uma biblioteca de componentes reusáveis e garantir que os desenvolvedores de software possam usar essa biblioteca pode custar caro. Os processos de desenvolvimento têm de ser adaptados para garantir que a biblioteca seja utilizada.
Encontrar, entender e adaptar componentes reusáveis	Os componentes de software têm de ser descobertos em uma biblioteca, compreendidos e, às vezes, adaptados para trabalhar em um novo ambiente. Os engenheiros devem estar razoavelmente confiantes de que encontrarão um componente na biblioteca antes de incluírem uma busca por componentes como parte de seu processo de desenvolvimento normal.
Maiores custos de manutenção	Se o código-fonte de um sistema de software ou componente reusado não estiver disponível, então os custos de manutenção podem ser mais altos porque os elementos reusados do sistema podem se tornar incompatíveis com as mudanças feitas no sistema.
Falta de suporte da ferramenta	Algumas ferramentas de software não fornecem suporte ao desenvolvimento com reúso. Pode ser difícil ou impossível integrar essas ferramentas com um sistema de biblioteca de componentes. O processo de software empregado por essas ferramentas pode não levar em conta o reúso. É mais provável que isso aconteça com as ferramentas que apoiam a engenharia de sistemas embarcados do que com

	as ferramentas de desenvolvimento orientado a objetos.
Síndrome do 'não inventado aqui'	Alguns engenheiros de software preferem reescrever os componentes porque acreditam que podem aperfeiçoá-los. Isso tem a ver em parte com a confiança e em parte com o fato de que escrever o software original é encarado como algo mais desafiador do que reusar o software de outras pessoas.

Quadro 2 | Problemas do Reúso de Software. Fonte: adaptado de Sommerville (2018).

É necessário adaptar os processos de desenvolvimento de software para incorporar o reúso. Especificamente, é essencial incluir uma fase de refinamento dos requisitos do sistema, na qual esses requisitos sejam ajustados para incorporar o software reutilizável disponível. Além disso, as etapas de projeto e a implementação do sistema podem incorporar atividades explícitas para identificar e avaliar possíveis componentes para reúso (Sommerville, 2018).

## Siga em Frente...

## Panorama do Reúso

Nos últimos 20 anos, foram desenvolvidas diversas técnicas para facilitar o reúso de software. Essas técnicas se baseiam na observação de que sistemas dentro de um mesmo domínio de aplicação compartilham semelhanças e têm potencial para reúso; na compreensão de que o reúso pode ocorrer em diferentes níveis, desde funções simples até aplicações completas; e na utilização de padrões para componentes reutilizáveis, o que simplifica o processo de reúso. A Figura 1 ilustra o panorama do reúso, demonstrando diferentes maneiras de implementar essa prática no desenvolvimento de software.



Figura 1 | O Panorama de Reúso. Fonte: Sommerville (2018).

Algumas das abordagens citadas para o reúso está brevemente descrita no Quadro 3.

Abordagem	Descrição
Frameworks de aplicação	Coleções de classes abstratas e concretas são adaptadas e estendidas para criar sistemas de aplicação.
Integração de sistemas de aplicação	Dois ou mais sistemas de aplicação são integrados para proporcionar mais funcionalidade.
Padrões de arquitetura	Padrões de arquiteturas de software que apoiam tipos comuns de sistemas de aplicação são utilizados como base de aplicações.
Desenvolvimento de software orientado a aspectos	Componentes compartilhados são ‘entrelaçados’ em uma aplicação em diferentes lugares quando o programa é compilado.
Engenharia de software baseada em componentes	Sistemas são desenvolvidos integrando componentes (coleções de objetos) que se adaptam aos padrões de modelo de componente.

Sistemas de aplicação configuráveis	Sistemas específicos de domínio são projetados para que possam ser configurados para as necessidades de clientes de sistemas específicos.
Padrões de projeto	Abstrações genéricas que ocorrem entre aplicações são representadas como padrões de projeto que exibem objetos e interações abstratos e concretos.

Quadro 3 | Abordagens que apoiam o reúso de software. Fonte: adaptado Sommerville (2018).

Considerando o conjunto de técnicas disponíveis para o reúso, a pergunta central é: qual técnica é mais adequada para uma situação específica? Naturalmente, a resposta depende dos requisitos do sistema em desenvolvimento, da tecnologia disponível, dos ativos reutilizáveis disponíveis e da experiência da equipe de desenvolvimento. Os principais fatores a serem considerados ao planejar o reúso são os seguintes (Sommerville, 2018):

- **Cronograma de desenvolvimento do software:** se há uma necessidade de desenvolver o software rapidamente, a preferência pode ser pelo reúso de sistemas completos em vez de componentes individuais. Embora a adequação aos requisitos possa não ser perfeita, essa abordagem minimiza a quantidade de desenvolvimento necessário.
- **Tempo de vida previsto para o software:** ao desenvolver um sistema destinado a uma longa vida útil, é crucial considerar a capacidade de manutenção do sistema. Não se deve apenas considerar os benefícios imediatos do reúso, mas também suas implicações a longo prazo. Durante a vida útil de um sistema, é provável que ele precise ser adaptado a novos requisitos, o que implica mudanças em partes dele. Se não houver acesso ao código-fonte dos componentes reutilizáveis, pode ser preferível evitar componentes de prateleira e sistemas de fornecedores externos, pois esses fornecedores podem não oferecer suporte contínuo ao software reutilizado. Nesse caso, pode ser mais seguro reusar sistemas e componentes de código aberto, pois isso permite acessar e manter cópias do código-fonte.
- **Formação, habilidades e experiência da equipe de desenvolvimento:** todas as tecnologias de reúso são bastante complexas e exigem tempo para serem compreendidas e utilizadas com eficácia. Portanto, é crucial concentrar os esforços de reúso nas áreas em que a equipe de desenvolvimento tenha experiência.
- **Criticidade do software e requisitos não funcionais:** para um sistema crítico, que precisa ser certificado por um regulador externo, pode ser necessário criar um caso de segurança ou de segurança da informação para esse sistema. Isso pode ser difícil se não for possível ter acesso ao código-fonte do software.
- **O domínio de aplicação:** é outro ponto a considerar. Em diversos domínios, como sistemas de informação industriais e sistemas de informação médica, existem produtos genéricos disponíveis que podem ser reutilizados, bastando serem configurados para se adequarem a uma situação específica. Esta é uma das abordagens mais eficazes para o reúso e muitas vezes é mais econômico comprar do que desenvolver um sistema novo.

- **A plataforma na qual o sistema será executado:** também desempenha um papel importante. Alguns modelos de componentes, como o .NET, são específicos para plataformas da Microsoft. Da mesma forma, sistemas de aplicação genéricos podem ser projetados para uma plataforma específica e só serão reutilizáveis se o novo sistema for compatível com essa plataforma.

A variedade de técnicas de reúso disponíveis é tão ampla que, na maioria das situações, existe a possibilidade de algum nível de reúso de software. No entanto, a decisão sobre se o reúso será realizado ou não é mais uma questão de gestão do que técnica. Isso ocorre porque os gestores podem hesitar em comprometer seus requisitos para permitir a utilização de componentes reutilizáveis (Sommerville, 2018). Eles podem não compreender os riscos associados ao reúso tão bem quanto compreendem os riscos do desenvolvimento original. Embora os riscos envolvidos no desenvolvimento de um novo software possam ser maiores, alguns gestores podem preferir lidar com os riscos conhecidos do desenvolvimento em vez dos riscos desconhecidos do reúso. Para promover o reúso em toda a empresa, pode ser necessário implementar um programa de reúso que se concentre na criação de ativos e processos reutilizáveis para facilitar a prática.

## Frameworks de Aplicação

Os pioneiros do desenvolvimento orientado a objetos sugeriram que um dos principais benefícios dessa abordagem era a possibilidade de reutilizar objetos em sistemas diferentes. No entanto, a experiência demonstrou que os objetos muitas vezes são granulares e especializados demais para uma aplicação específica. Frequentemente, é necessário mais tempo para entender e adaptar o objeto do que para implementá-lo. Hoje em dia, é evidente que a reutilização orientada a objetos é mais adequadamente suportada por um processo de desenvolvimento orientado a objetos, por meio de abstrações mais genéricas chamadas de frameworks.

Um framework é uma estrutura genérica estendida para criar um subsistema ou uma aplicação mais específicos. Um framework pode ser definido como um conjunto integrado de artefatos de software, como classes, objetos e componentes, que colaboram para fornecer uma arquitetura reutilizável para uma família de aplicações relacionadas (Sommerville, 2018).

Os frameworks fornecem suporte a características genéricas que tendem a ser utilizadas em todas as aplicações de um tipo similar. Por exemplo, um framework de interface com o usuário oferece suporte para o tratamento de eventos de interface e inclui um conjunto de componentes de interface que podem ser usados para construir telas. Em seguida, cabe ao desenvolvedor especializar esse framework adicionando funcionalidades específicas para uma determinada aplicação. Por exemplo, em um framework de interface com o usuário, o desenvolvedor define os layouts de tela adequados para a aplicação que está sendo implementada.

Os frameworks promovem a reutilização de projeto ao fornecerem um esqueleto de arquitetura para a aplicação, além da reutilização de classes específicas no sistema. A arquitetura é implementada pelas classes e suas interações. As classes são reutilizadas diretamente e podem ser estendidas com o uso de características como herança e polimorfismo.

Os frameworks são coleções de classes concretas e abstratas implementadas em linguagens de programação orientadas a objetos, como Java, C#, C++, Ruby e Python. Eles são específicos para cada linguagem e são utilizados para facilitar o desenvolvimento de aplicações. Os frameworks podem ser incorporados uns aos outros e são projetados para dar suporte ao desenvolvimento de diferentes partes de uma aplicação, podendo ser utilizados para criar uma aplicação completa ou implementar partes específicas, como a interface gráfica do usuário.

Os frameworks mais comuns são os frameworks de aplicação web (WAFs), que auxiliam na construção de sites dinâmicos. Eles são geralmente baseados no padrão composto Modelo-Visão-Controlador (MVC), que separa o estado da apresentação em uma aplicação, permitindo a atualização do estado a partir de diferentes apresentações. O padrão MVC foi inicialmente proposto nos anos 1980 para o projeto de interfaces gráficas de usuário (GUI), permitindo múltiplas representações de um objeto e diferentes estilos de interação com cada uma dessas representações. A Figura 2, apresenta o padrão MVC.

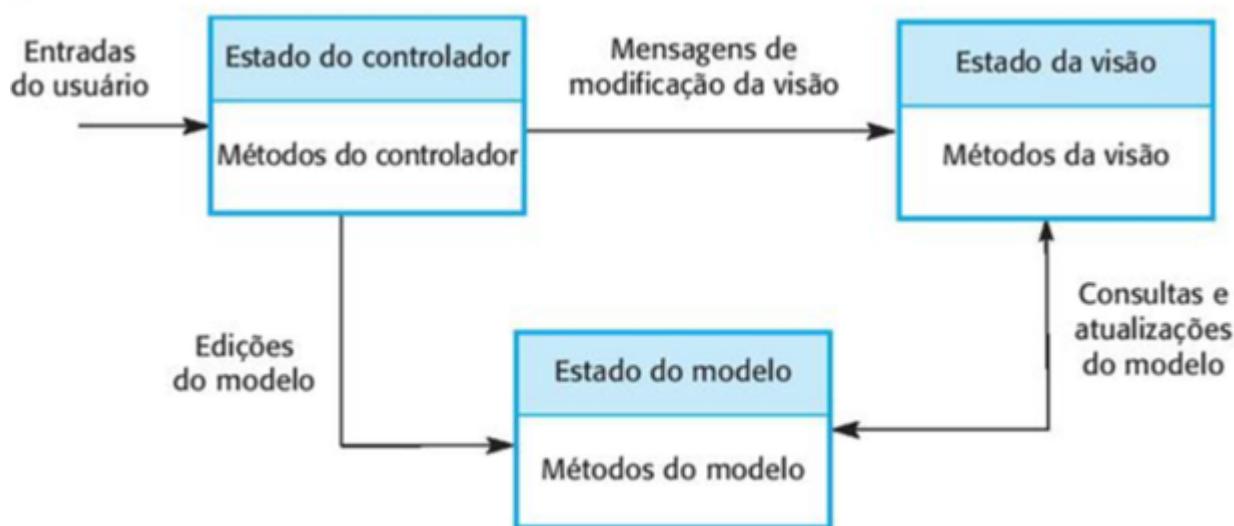


Figura 2 | Padrão MVC. Fonte: Sommerville (2018).

Um framework MVC ajuda na apresentação dos dados de diferentes maneiras e permite a interação com cada uma dessas apresentações. Quando os dados são modificados em uma das apresentações, o modelo do sistema é modificado, e os controladores associados a cada visão atualizam a sua apresentação.

- Outras classes de frameworks, estão descritas a seguir (Sommerville, 2018):
- Os frameworks de infraestrutura de sistema auxiliam no desenvolvimento de infraestruturas de sistema, incluindo comunicações, interfaces com o usuário e compiladores.
- Os frameworks de integração e middleware consistem em conjuntos de padrões e classes associadas que facilitam a comunicação entre componentes e a troca de informações. Exemplos incluem o .NET da Microsoft e o Enterprise Java Beans (EJB), que suportam modelos de componentes padronizados.

- Os frameworks de aplicação corporativa são específicos para domínios de aplicação como telecomunicações ou sistemas financeiros. Eles incorporam conhecimento do domínio da aplicação e ajudam no desenvolvimento de aplicações para usuários finais. Embora não sejam amplamente utilizados, foram praticamente substituídos por linhas de produtos de software.

Aplicações desenvolvidas com frameworks podem servir como base para uma futura reutilização, seguindo o conceito de linhas de produtos de software ou famílias de aplicações. Como essas aplicações são construídas com base em um framework, modificar os membros da família para criar novas instâncias de sistema geralmente é um processo direto, envolvendo a modificação de classes e métodos específicos que foram adicionados ao framework (Sommerville, 2018).

Embora os frameworks sejam uma abordagem eficaz para o reúso, introduzi-los nos processos de desenvolvimento de software é dispendioso, pois são naturalmente complexos e pode levar meses para dominar seu uso. Avaliar os frameworks disponíveis e selecionar o mais adequado também pode ser uma tarefa difícil e custosa. Além disso, a depuração de aplicações baseadas em frameworks é mais desafiadora do que depurar o código original, pois nem sempre é claro como os métodos do framework interagem. Ferramentas de depuração podem fornecer informações sobre os componentes do framework reutilizados que podem ser difíceis de compreender para o desenvolvedor.

## Vamos Exercitar?

Uma possível abordagem dos temas pedidos no relatório está apresentada a seguir:

### Importância do Reúso de Software

- **Conceito:** reúso de software refere-se à prática de utilizar componentes de software existentes em novas aplicações, o que pode aumentar a eficiência e qualidade do desenvolvimento, ao mesmo tempo que reduz custos.
- **Benefícios:** incluem a aceleração do desenvolvimento, a padronização de componentes, e a redução de erros, já que os componentes reutilizados tendem a ser mais testados e confiáveis.
- **Desafios:** abrangem a integração de componentes com diferentes sistemas e a necessidade de manter a compatibilidade entre as várias partes do software.

### Arquitetura MVC

- **Descrição:** a arquitetura MVC separa a lógica de negócios (Modelo), a interface do usuário (Visão) e a lógica de entrada (Controlador), promovendo uma organização clara do código e facilitando a manutenção e o reúso.
- **Facilitação do Reúso:** a separação de responsabilidades permite que cada componente seja desenvolvido, testado e reutilizado independentemente.

## Reúso de Software com MVC

- **Promoção do Reúso:** MVC apoia o reúso através de modelos que podem ser aplicados a diferentes funcionalidades e visões que podem ser adaptadas para múltiplos contextos, sem a necessidade de duplicação de código.
- **Exemplos no e-commerce:** modelos para gerenciamento de usuários e produtos podem ser reutilizados em diferentes módulos; componentes de visualização, como listas de produtos, podem ser padronizados e reutilizados.

## Reflexão sobre Reúso Estratégico

- **Estratégias para Promover Reúso:** incluem o desenvolvimento de bibliotecas de componentes reutilizáveis, o emprego de frameworks MVC e a implementação de práticas de design modular.
- **Impacto a Longo Prazo:** o reúso estratégico pode significar facilidade de atualização e expansão da plataforma, adaptando-se eficientemente a mudanças de requisitos ou tecnologia.

## Reflexão sobre Reúso Estratégico

- **Estratégias para Promover Reúso:** incluem o desenvolvimento de bibliotecas de componentes reutilizáveis, o emprego de frameworks MVC e a implementação de práticas de design modular.
- **Impacto a Longo Prazo:** o reúso estratégico pode significar facilidade de atualização e expansão da plataforma, adaptando-se eficientemente a mudanças de requisitos ou tecnologia.

## Saiba mais

Quer saber mais sobre o Reúso de Software? Leia o Capítulo 15 de [Engenharia de Software - Sommerville](#)

Leia o artigo [A reutilização de software e suas aplicações](#) para aprofundar seus conhecimentos nesse assunto.

Reúso de software é um processo recorrente no ciclo de desenvolvimento de software, o artigo [Visão Geral Reuso de Software](#) apresenta os principais conceitos e técnicas associados a este processo vital para competitividade das empresas de desenvolvimento de software.

## Referências

HIRAMA, K. **Engenharia de software:** qualidade e produtividade com tecnologia. Rio de Janeiro: Elsevier, 2011.

PÁDUA, W. **Engenharia de software**. 1. ed. Rio de Janeiro: LTC, 2019.

SOMMERVILLE, I. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

## Aula 5

Encerramento da Unidade

### Videoaula de Encerramento



#### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula você relembrará os pilares da manutenção de software, a gestão de riscos, as métricas de software e o reúso de componentes. Esses são aspectos fundamentais da engenharia de software, permitindo aprimorar a qualidade, eficiência e confiabilidade dos sistemas desenvolvidos. Ao compreender e aplicar esses conceitos, os profissionais estarão mais bem preparados para enfrentar os desafios do desenvolvimento e a manutenção de software, maximizando o valor entregue aos clientes e aos usuários. Prepare-se para uma jornada de aprendizado enriquecedora e prática!

### Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é avaliar e saber utilizar as métricas e a análise de software, você deverá primeiramente conhecer os conceitos fundamentais da manutenção de software. Essa é uma atividade crucial no ciclo de vida do desenvolvimento de software, que envolve modificar, corrigir e aprimorar um sistema de software existente para garantir sua eficácia contínua e adaptação às necessidades em evolução. Essa prática abrange diversas atividades, como correção de bugs, implementação de melhorias funcionais, otimização de desempenho e atualizações de segurança. A manutenção de software não apenas prolonga a vida útil de um sistema, mas também contribui para sua confiabilidade,

segurança e usabilidade, garantindo que ele permaneça relevante e competitivo no ambiente tecnológico em constante mudança.

Os softwares são desenvolvidos com algoritmos para atender requisitos específicos, porém, podem sofrer mudanças ao longo do tempo, sujeitos a falhas de adequação e falhas na mudança. A falha de adequação ocorre quando os requisitos não são implementados corretamente, levando à perda de integridade e confiabilidade do software (Pádua, 2019). Por exemplo, uma atualização de sistema operacional pode tornar um software incompatível. Já a falha na mudança acontece quando uma atualização afeta negativamente outras funcionalidades, como implementar uma nova forma de pagamento que interfere no funcionamento de outros sistemas. É essencial identificar e corrigir essas falhas para garantir a continuidade e a eficácia do software.

A manutenção de software abrange três principais tipos: adaptativa, corretiva e evolutiva. A manutenção adaptativa é realizada para ajustar o software a mudanças no ambiente operacional, como atualizações de sistemas ou hardware. Já a manutenção corretiva envolve a correção de defeitos ou bugs identificados no software, garantindo sua funcionalidade adequada. Por fim, a manutenção evolutiva tem como objetivo aprimorar o software, adicionando novos recursos ou melhorias de desempenho para atender às necessidades em constante evolução dos usuários. Cada tipo de manutenção desempenha um papel importante na garantia da qualidade e eficácia contínua do software ao longo do seu ciclo de vida.

Outro ponto importante da manutenção de software é a reengenharia de software, que visa reorganizar ou modificar um sistema para restaurar seu desempenho aceitável, sendo desencadeada por fatores como falta de atualização ou excesso de mudanças. Optar pela reconstrução pode ser mais eficaz do que tentar resolver problemas em um sistema comprometido, permitindo correção de erros, atualizações necessárias e melhorias estruturais. Seus objetivos incluem aprimorar a manutenção, atualizar a documentação e refazer a estrutura do sistema, facilitando futuras atividades.

A probabilidade de erro em sistemas revisados durante a reengenharia é reduzida, uma vez que os processos tendem a mitigar riscos já previamente tratados. A aplicação de técnicas como a engenharia reversa desempenha um papel crucial nesse contexto, permitindo a reconstrução do código para compreensão e otimização das funcionalidades. Por meio de métodos específicos, como análise em níveis de implementação, estrutura, funcionalidade e domínio, é possível examinar o software de diversas perspectivas e melhorar sua qualidade de maneira significativa (Pádua, 2019).

Profissionalmente, a reengenharia e a engenharia reversa exigem técnicas desafiadoras, mas valiosas, para reduzir o tempo de desenvolvimento e alcançar melhorias substanciais. Ao utilizar essas práticas, espera-se um aumento na eficiência e na confiabilidade do software, visto que muitas funcionalidades fundamentais já foram desenvolvidas anteriormente, proporcionando um ambiente mais propício à evolução e manutenção dos sistemas de software (Pressman, 2021).

A evolução de software faz parte da manutenção, e cabe destacar que não existe um método padrão único para a evolução do software, variando de acordo com a natureza do sistema, os

processos de desenvolvimento adotados e as habilidades das pessoas envolvidas. Em sistemas como aplicativos móveis, a evolução pode ser informal, com alterações sugeridas em conversas entre usuários e desenvolvedores, enquanto em sistemas críticos embarcados, a evolução pode ser formalizada, com documentação estruturada em cada fase do processo. Propostas de mudança, formais ou informais, impulsionam a evolução dos sistemas, originando-se de requisitos não implementados, solicitações de novos requisitos, relatórios de defeitos ou ideias da equipe de desenvolvimento para melhorar o software (Sommerville, 2018).

Antes da aceitação de uma proposta de mudança, é realizada uma análise detalhada do software para determinar quais componentes precisam ser alterados, avaliando custo e impacto. Essa análise faz parte do gerenciamento global de mudanças e visa garantir a inclusão das versões corretas dos componentes em cada lançamento do sistema (Pressman, 2021). O processo de evolução de software envolve atividades como análise da mudança, planejamento de lançamento, implementação do sistema e lançamento para os clientes, com uma cuidadosa avaliação do custo e impacto das mudanças antes da decisão sobre a implementação na próxima versão.

Ao abordar os riscos na engenharia de software, é essencial considerar o impacto futuro e as mudanças nos requisitos do projeto, tecnologias e escolhas de desenvolvimento. O conceito de débito técnico destaca os custos associados ao adiamento de atividades essenciais, como documentação e refatoração, enfatizando a importância de abordar problemas técnicos precocemente para evitar complicações futuras. Apesar da adoção generalizada do desenvolvimento ágil, equipes podem acumular débito técnico devido a demandas crescentes por código e falta de dedicação para reduzi-lo, destacando a necessidade de conscientização e planejamento cuidadoso durante os sprints definidos.

A análise de riscos na engenharia de software considera categorias como riscos de projeto, técnicos e de negócio, que ameaçam o cronograma, a qualidade e a viabilidade do produto. Identificar e gerenciar esses riscos são cruciais para o sucesso do projeto. Enquanto alguns riscos podem ser previstos por meio de uma avaliação cuidadosa do ambiente técnico e comercial, outros são imprevisíveis e representam desafios significativos durante o desenvolvimento do software.

A identificação de riscos é um processo importante para antecipar e controlar ameaças ao plano do projeto, abordando riscos genéricos e específicos do produto (Pressman, 2021). Uma abordagem comum para identificar riscos é criar uma lista de verificação de itens de risco, considerando subcategorias como tamanho do produto, impacto nos negócios e nas habilidades da equipe. Questões-chave, como comprometimento da alta gerência e compreensão dos requisitos, são fundamentais para avaliar o nível de risco do projeto.

A previsão de riscos busca classificar cada risco com base na probabilidade de ocorrência e nas consequências associadas, priorizando-os para direcionar os recursos de forma eficaz. Elaborar uma tabela de riscos é uma técnica comum, ordenando os riscos com base na probabilidade e no impacto para estabelecer estratégias de mitigação, monitoramento e gestão de risco. Abordagens avançadas, como o uso de lógica difusa, podem ser empregadas para avaliar riscos complexos e inter-relacionados, especialmente em cenários de alta incerteza.

O processo de análise de riscos na engenharia de software visa auxiliar a equipe do projeto na elaboração de estratégias para lidar com potenciais ameaças. Essas estratégias devem abordar como evitar, monitorar e gerenciar os riscos, além de planejar contingências (Sommerville, 2018). Uma postura proativa é crucial, com a prevenção como prioridade, incluindo a criação de planos de mitigação para riscos identificados, como a alta rotatividade de pessoal.

Durante o projeto, é essencial antecipar e implementar práticas para minimizar os riscos, como a disseminação de informações entre equipes e a designação de substitutos para membros críticos. O monitoramento contínuo dos riscos ao longo do projeto é fundamental, permitindo ajustes e avaliando a eficácia das medidas de mitigação adotadas.

A gestão de riscos e o plano de contingência entram em ação quando as medidas de mitigação falham e os riscos se materializam. A análise de custo-benefício é essencial para determinar se os benefícios das medidas de gerenciamento de risco superam os custos associados à sua implementação. A regra de Pareto sugere concentrar esforços nos riscos mais críticos, enquanto a gamificação pode incentivar a conformidade com processos, como qualidade e gestão de riscos. O plano RMMM é uma ferramenta valiosa que documenta todas as atividades relacionadas à análise de riscos e serve como parte integrante do plano global do projeto de software (Pressman, 2021).

A medição desempenha um papel importante no desenvolvimento de software, permitindo melhorias contínuas ao longo do ciclo de vida do projeto. Essas medidas são empregadas para aprimorar estimativas, garantir qualidade, aumentar produtividade e controlar o progresso do projeto. Elas oferecem insights sobre os atributos dos modelos desenvolvidos e possibilitam a avaliação da qualidade dos produtos ou sistemas construídos.

Engenheiros de software usam medições para avaliar a qualidade dos artefatos e auxiliar em decisões táticas durante o projeto. Métricas de produtividade e qualidade são focos principais, abrangendo a saída de desenvolvimento em relação ao esforço e tempo aplicados, além da adequação dos artefatos produzidos. Essas medições não apenas fornecem um entendimento técnico, mas também apoiam o gerenciamento do projeto, auxiliando na tomada de decisões para o sucesso do empreendimento (Pressman, 2021).

Embora os termos medida, medição e métricas sejam frequentemente intercambiáveis, há diferenças sutis entre eles. Uma medida é um único dado coletado, enquanto a medição envolve a coleta de um ou mais pontos de dados. Já uma métrica de software relaciona as medidas individuais de alguma maneira, oferecendo indicadores sobre o processo de software ou o produto em si. Métricas eficazes devem ser claras, fáceis de calcular, alinhadas às intuições dos engenheiros, produzir resultados inequívocos e contribuir para um produto de melhor qualidade.

As métricas de produto são usadas para quantificar atributos internos de sistemas de software, como tamanho e complexidade. Elas podem ser classificadas em métricas dinâmicas, obtidas durante a execução do programa, e métricas estáticas, medidas em representações do sistema. As métricas dinâmicas, como o número de defeitos relatados ou o tempo de execução de uma computação, ajudam a avaliar eficiência e confiabilidade. Já as métricas estáticas, como aquelas relacionadas à complexidade do código, têm uma relação mais indireta com atributos de

qualidade, mas o tamanho do programa e a complexidade do controle emergem como indicadores confiáveis de compreensibilidade e manutenibilidade (Sommerville, 2018).

As métricas de processo são coletadas ao longo de vários projetos, proporcionando indicadores para aprimorar o processo de software a longo prazo. Por outro lado, as métricas de projeto permitem avaliar o progresso de um projeto em andamento, identificar riscos potenciais e ajustar o fluxo de trabalho conforme necessário (Pressman, 2021). Essas medidas, compartilhadas entre equipes de projeto e de aprimoramento de processos, desempenham um papel tanto estratégico quanto tático, fornecendo insights valiosos para a melhoria contínua.

A implementação de métricas de processo requer uma abordagem lógica, envolvendo a medição de atributos específicos, o desenvolvimento de métricas significativas e seu uso para orientar estratégias de melhoria. Além disso, é crucial reconhecer que o processo é apenas um dos vários elementos que influenciam a qualidade do software e o desempenho organizacional. Esses fatores incluem habilidade e motivação das pessoas, complexidade do produto, tecnologia utilizada e contexto ambiental mais amplo, como ferramentas de software, condições de negócios e características do cliente. O uso adequado das métricas, juntamente com práticas recomendadas de etiqueta de métricas de software, pode levar a melhorias substanciais no processo e na qualidade do produto ao longo do tempo.

A engenharia de software baseada em reúso tem se tornado uma estratégia essencial, impulsionada pela necessidade de reduzir custos, acelerar o desenvolvimento e melhorar a qualidade do software (Pressman, 2021). Atualmente, uma variedade de software reutilizável está disponível, incluindo bibliotecas de código aberto, sistemas de aplicação específicos para domínios e componentes oferecidos por empresas. Essa diversidade de recursos permite o reúso em diferentes níveis, desde sistemas completos até objetos e funções específicas, facilitando a adaptação e a configuração para atender a novas necessidades.

O reúso de software oferece diversos benefícios, incluindo redução de custos, menor tempo de desenvolvimento e maior qualidade. No entanto, existem desafios associados, como os custos de avaliação e teste de componentes para garantir sua adequação e confiabilidade (Sommerville, 2018). Para incorporar o reúso de forma eficaz, os processos de desenvolvimento de software devem ser adaptados para incluir fases específicas de refinamento de requisitos e atividades de identificação e avaliação de componentes reutilizáveis durante o projeto e implementação do sistema.

Nos últimos 20 anos, várias técnicas surgiram para facilitar o reúso de software, impulsionadas pela percepção de que sistemas dentro do mesmo domínio de aplicação compartilham semelhanças. Essas técnicas variam desde o reúso de funções simples até a reutilização de aplicações completas, com o suporte de padrões para componentes reutilizáveis (Sommerville, 2018).

Entretanto, a escolha da técnica de reúso mais adequada para uma situação específica depende de diversos fatores, como o cronograma de desenvolvimento, o tempo de vida previsto para o software, as habilidades da equipe de desenvolvimento, a criticidade do software e o domínio de aplicação. A gestão desempenha um papel crucial na decisão de adotar o reúso, pois os

gestores podem relutar em comprometer requisitos para permitir o uso de componentes reutilizáveis, devido aos riscos associados e à falta de compreensão em comparação com o desenvolvimento original. Assim, promover o reúso em toda a empresa pode exigir a implementação de um programa focado na criação de ativos e processos reutilizáveis.

Os primeiros proponentes do desenvolvimento orientado a objetos sugeriram que a reutilização de objetos poderia ser um dos principais benefícios dessa abordagem. No entanto, a granularidade e a especialização dos objetos muitas vezes dificultavam sua aplicação direta em contextos específicos. Hoje, a reutilização orientada a objetos é mais eficazmente suportada por frameworks, estruturas genéricas que fornecem uma arquitetura reutilizável para diferentes aplicações. Os frameworks abstraem características comuns e genéricas que são necessárias em várias aplicações, permitindo que os desenvolvedores as personalizem conforme necessário.

Os frameworks são compostos por coleções de classes e objetos que colaboram para fornecer funcionalidades comuns a uma família de aplicações relacionadas. Eles são implementados em linguagens de programação orientadas a objetos e podem abranger diferentes domínios, desde aplicações web até sistemas corporativos específicos. Os frameworks mais comuns incluem os frameworks de aplicação web, baseados no padrão *Model-View-Controller* (MVC), que separam o estado da apresentação em uma aplicação, facilitando sua atualização a partir de diferentes visões. Além disso, existem frameworks para infraestrutura de sistema, integração e middleware, e aplicação corporativa, cada um com sua finalidade específica (Sommerville, 2018).

Embora os frameworks sejam uma estratégia eficaz para o reúso de software, sua introdução nos processos de desenvolvimento pode ser dispendiosa devido à sua complexidade. Selecionar e dominar um framework adequado pode levar tempo e recursos significativos, e a depuração de aplicações baseadas em frameworks pode ser mais desafiadora devido à complexidade das interações entre os componentes do framework. No entanto, uma vez dominados, os frameworks podem facilitar o desenvolvimento e a manutenção de aplicações, promovendo a reutilização de código e a eficiência no desenvolvimento de software (Sommerville, 2018).

## É Hora de Praticar!



### Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Esta atividade consiste na elaboração de relatório no qual o aluno deve explorar a relação entre o reúso de software e a manutenção de software. Este relatório deve abordar os seguintes pontos:

1. **Definição de reúso de software:** breve explicação sobre o que é reúso de software e suas principais formas (e.g., bibliotecas, frameworks, componentes, padrões de projeto).
2. **Definição de manutenção de software:** explicação sobre o que compreende a manutenção de software, incluindo suas categorias (corretiva, adaptativa, perfectiva e preventiva).
3. **Impactos do reúso na manutenção:** discussão sobre como o reúso de software pode influenciar cada tipo de manutenção de software, destacando tanto os aspectos positivos quanto os negativos.
4. **Estratégias para maximizar os benefícios:** propostas de estratégias ou práticas que podem ser adotadas para maximizar os benefícios do reúso de software na manutenção, minimizando os desafios e os riscos associados.

**Conclusão:** reflexões finais sobre a importância de considerar o reúso de software como uma estratégia para melhorar a eficiência e a eficácia da manutenção de sistemas.

- Como a gestão eficaz da manutenção de software pode impactar não apenas a estabilidade e o desempenho dos sistemas, mas também a produtividade da equipe de desenvolvimento e a satisfação do cliente, e quais estratégias podem ser adotadas para maximizar os benefícios dessa prática enquanto se minimizam os custos e os riscos associados?
- Até que ponto as métricas de software são realmente eficazes para avaliar a qualidade e o progresso de um projeto de desenvolvimento de software?
- Até que ponto o reúso de software promove a eficiência e a inovação no desenvolvimento de novos sistemas, e quais são os desafios éticos e técnicos associados à prática do reúso em termos de segurança, propriedade intelectual e manutenção a longo prazo?

Para essa atividade, uma possível elaboração para o relatório está apresentada a seguir:

1. **Definição de reúso de software:** reúso de software é a prática de usar código existente para novos desenvolvimentos, reduzindo o tempo e o custo de desenvolvimento e melhorando a qualidade do software ao aproveitar soluções já testadas.
2. **Definição de manutenção de software:** a manutenção de software é o processo de modificar um software após a entrega, para corrigir falhas, melhorar a performance ou outros atributos, ou adaptar o software a um ambiente alterado.
3. **Impactos do reúso na manutenção:**
4. **Positivos:** pode reduzir o esforço de manutenção ao usar componentes bem testados e documentados; facilita a manutenção corretiva e perfectiva ao promover a consistência no código; e pode acelerar a adaptação a novas plataformas ao reutilizar software compatível.
5. **Negativos:** pode introduzir dependências externas que complicam a manutenção adaptativa; a necessidade de compatibilidade com componentes reutilizados pode limitar as opções de atualização e melhoria.
6. **Estratégias para maximizar os benefícios:**
  - Adotar políticas de documentação e versionamento rigorosas para componentes reutilizáveis.
  - Fomentar uma cultura de qualidade e testes abrangentes para o software reutilizável.
  - Utilizar ferramentas e plataformas que facilitam a gestão de dependências e a integração contínua.

**Conclusão:** o reúso de software, quando bem planejado e executado, pode ser uma poderosa ferramenta para melhorar a manutenção de software, mas requer atenção cuidadosa aos detalhes de implementação e gestão de dependências para maximizar seus benefícios.

O infográfico sobre métricas de produto, projeto e processo oferece uma visão sobre essas métricas e a importância de cada uma delas.

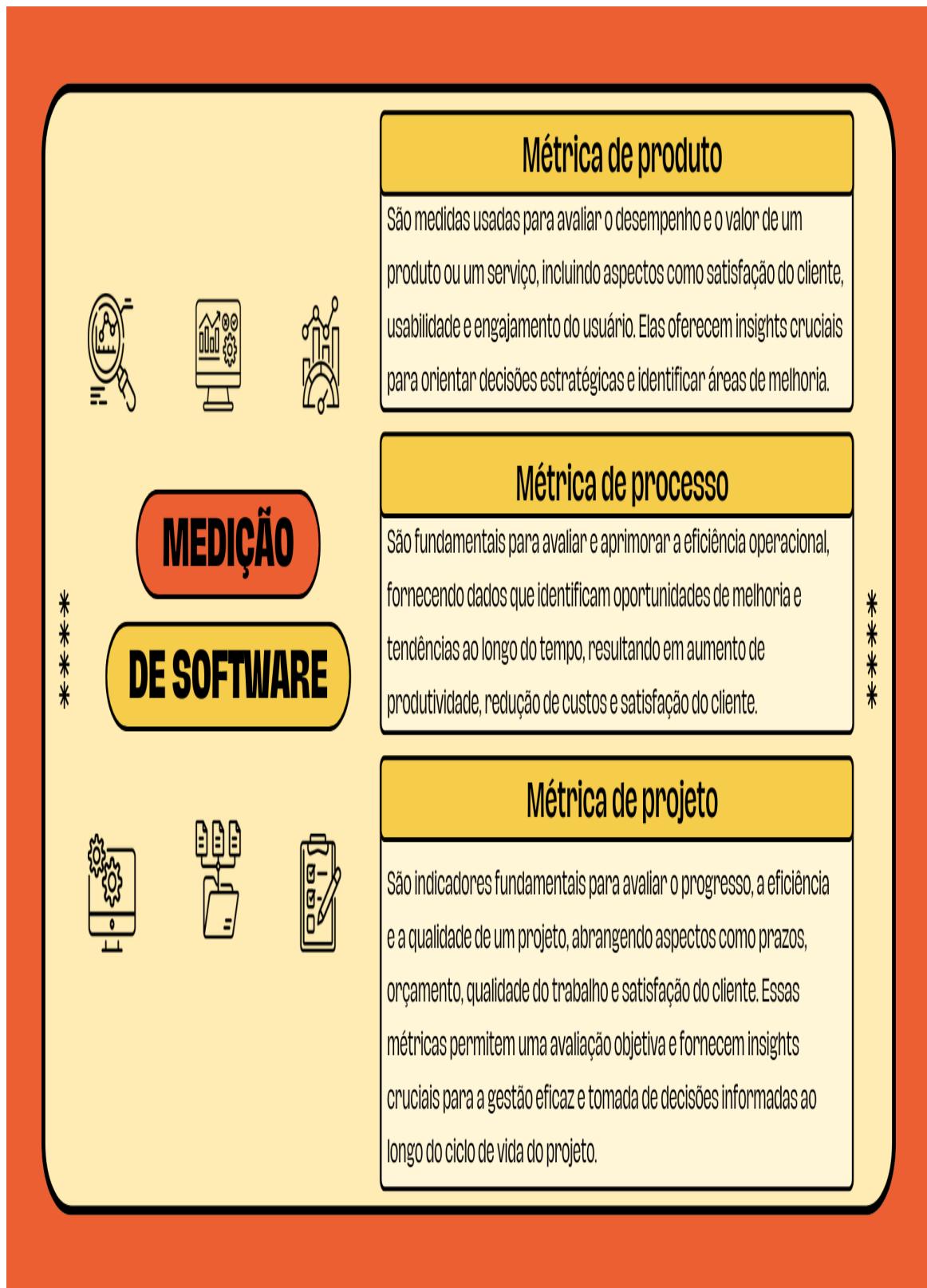


Figura | Medição de software.

PÁDUA, W. **Engenharia de software**. 1. ed. Rio de Janeiro: LTC, 2019.

# ENGENHARIA DE SOFTWARE

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional.** 9. ed. Porto Alegre: AMGH, 2021.

SOMMERVILLE, I. **Engenharia de software.** 10. ed. São Paulo: Pearson, 2018.