

Engenharia de Software

Definição:

Tem por objetivo a aplicação de teorias, modelos, formalismos, técnicas e ferramentas da lógica para o desenvolvimento de sistemas de softwares.

Processos de Desenvolvimento do Software:

Especificação = as funções do software são definidas.

Desenvolvimento = o software é produzido atendendo à especificação.

Validação = o software é validado, para garantir que atenda ao que o cliente deseja.

Evolução = o software deve evoluir para atender às mudanças nas necessidades dos clientes.

Modelos de Processos de Desenvolvimento:

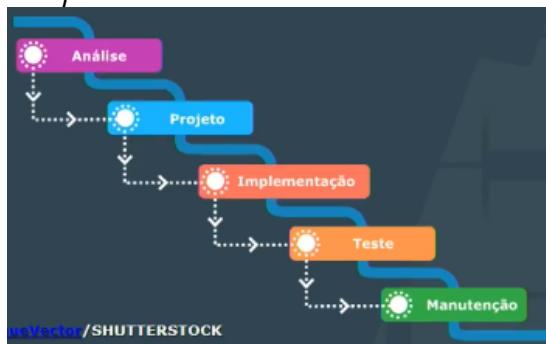
Processos Prescritivos = tradicionais, surgiram para organizar e direcionar as atividades do desenvolvimento de software.

Prescrevem um conjunto de elementos e atividades, como atividades metodológicas, ações de engenharia de software, tarefas, produtos de trabalho, garantia de qualidade e mecanismos de controle de mudanças para cada projeto.

Processos Ágeis = abordagem contrária aos processos prescritivos. Possuem iterações curtas, em que o resultado é medido por meio do produto pronto, ao contrário dos modelos prescritivos, em que o desenvolvimento é dividido em etapas bem definidas.

Modelo Cascata:

Surgiu em 70 a partir de modelos militares. Apresenta o desenvolvimento em fases, uma só inicia após conclusão da anterior.



Análise: identificação dos problemas e levantamento de requisitos do software.

Projeto: reparte os requisitos entre hardware e software, estabelece uma arquitetura global do sistema.

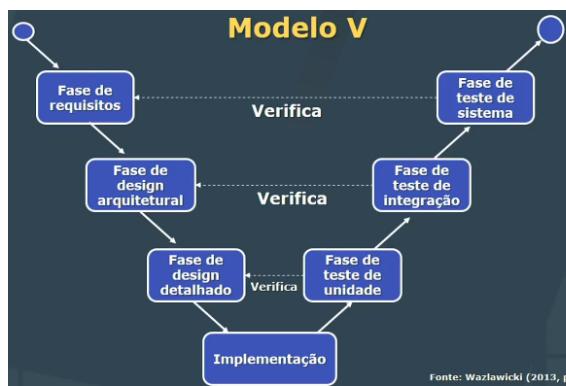
Implementação: desenvolvimento propriamente dito, de acordo com a linguagem de programação, arquitetura, equipe.

Teste: verificação de tudo o que foi feito, se tem algum problema ou não.

Manutenção: corrigir os erros encontrados e se possível melhorar o sistema de modo geral.

Vantagens	Desvantagens
Fases bem definidas ajudam a detectar erros mais cedo.	Não produz resultados tangíveis até a fase de codificação.
Procura promover a estabilidade dos requisitos.	É difícil estabelecer requisitos completos antes de começar a codificar.
Funciona bem quando os requisitos são conhecidos e estáveis.	Desenvolvedores sempre reclamam que os usuários não sabem expressar aquilo de que precisam.
É adequado para equipes tecnicamente fracas ou inexperientes.	Não há flexibilidade com requisitos.

Modelo V:



Fase de Requisitos = equipe e cliente elaboram documento de requisitos, na forma de um contrato de desenvolvimento.

Fase de Design Arquitetural = equipe organiza os requisitos em unidades, definindo como as diferentes partes arquiteturais do sistema vão se interconectar e colaborar.

Fase de Design Detalhado = equipe aprofunda a descrição das partes do sistema e decide como serão implementadas.

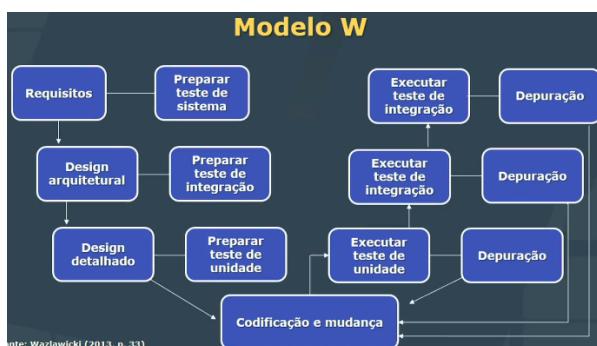
Fase de Implementação = o software é implementado de acordo com a especificação.

Fase de Teste de Unidade = verifica se todas as unidades se comportam como especificado. O projeto só segue se todas as unidades passam em todos os testes.

Fase de Teste de Integração = verifica se o sistema se comporta conforme a especificação do design arquitetural.

Fase de Teste de Sistema = verifica se o sistema satisfaz os requisitos especificados.

Modelo W:



Fase de Requisitos = apenas requisitos que possam ser testados são aceitáveis.

Fase de Design Arquitetural = arquiteturas devem ser simples e fáceis de testar, caso contrário, necessita ser refatorada.

Fase de Projeto Detalhado = unidades devem ser coesas são e fáceis de testar.

Modelo Espiral:

Criado em 1986, é feito de ciclos iterativos e subprojetos. O projeto é dividido em subprojetos, cada qual abordando um ou mais elementos de alto risco, até que todos os riscos identificados tenham sido tratados.



Primeira Iteração = concepção das operações.

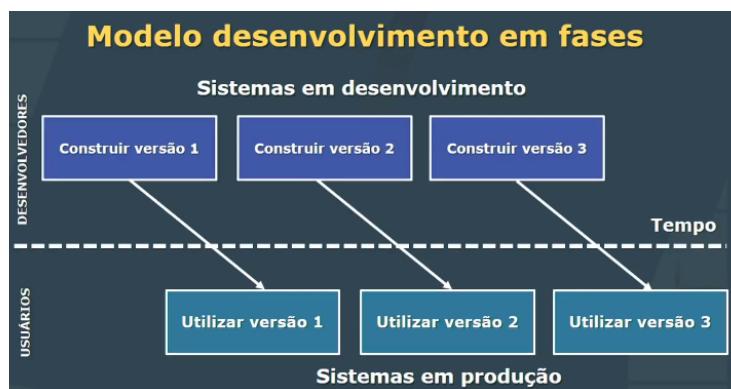
Segunda Iteração = requisitos do produto de software.

Terceira Iteração = desenvolvimento do sistema.

Quarta Iteração = testes.

Modelo de Desenvolvimento em Fases:

O sistema é projetado de modo que possa ser entregue em partes, possibilitando aos usuários dispor de alguns recursos enquanto o restante do sistema está sendo desenvolvido.



Sistema em Produção = é o sistema que está sendo atualmente utilizado pelo cliente.

Sistema em Desenvolvimento = é a versão seguinte que está sendo desenvolvida para substituir o sistema em produção.

Há dois principais modos de fazer isso:

Desenvolvimento Incremental = o sistema é dividido por funcionalidades e cada versão entregue corresponde a uma ou um conjunto dessas funcionalidades.

Desenvolvimento Iterativo = a primeira versão entregue corresponde ao sistema completo com funcionalidades limitadas. Cada versão posterior corresponde a mudanças realizadas nas funcionalidades limitadas.

Prototipação Evolucionária:

São usados protótipos para que o sistema ou parte dele seja construído rapidamente. Tem o mesmo objetivo de um protótipo de

engenharia, quando os requisitos ou o projeto necessitam de investigações repetidas para garantir que o desenvolvedor, usuário e cliente cheguem a um consenso sobre o que é necessário e o que é proposto.



Throw Away = ou descartável, consiste na construção de protótipos que usados unicamente para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos.

Cornerstone = ou fundamental, consiste na construção de protótipos para serem parte do sistema final, de modo que o protótipo vai evoluindo até se tornar um sistema que possa ser entregue.

Processo Unificado da Rational (RUP):

Concepção = é feito um plano de negócios para o sistema, para identificar as entidades externas e os requisitos do sistema a fim de avaliar a contribuição do sistema para o negócio.

Elaboração = são desenvolvidos os requisitos e a arquitetura do sistema.

Construção = implementação e testes do sistema.

Transição = implantação do sistema em ambiente real.



Disciplinas de projeto, acontecem em cada uma das quatro etapas do RUP:

Modelagem de negócios = estuda a empresa e seus processos para descrever as regras de negócio.

Requisitos = elicitação dos requisitos e design da interface do sistema.

Análise e design = detalhamento dos requisitos para elaboração da modelagem do sistema.

Implementação = desenvolvimento do sistema e testes de unidade.

Teste = exclui testes de unidade realizados na implementação e foca nos testes de integração e interação entre os componentes.

Implantação = produção de versões do sistema para serem entregues aos clientes.

Disciplinas de apoio:

Ambiente = aborda o ambiente de desenvolvimento do sistema, focando na configuração do processo usado para desenvolver o projeto. Trata das ferramentas de apoio para que a equipe tenha sucesso.

Gerenciamento de Mudança e Configuração = mantém a integridade do conjunto de artefatos produzidos ao longo do projeto.

Gerência de Projeto = planeja o projeto como um todo, cada iteração individualmente, gerencia os riscos do e monitora o progresso.

Desenvolvimento Ágil:

Surgiu para melhorar os modelos prescritivos, torná-los rápidos, em 90. Se perdia mais tempo em decisões que no desenvolvimento.

Manifesto para o Desenvolvimento Ágil de Software = criado pela Agile Alliance em 2001. Considera:

Indivíduos e interações > processos e ferramentas;

Software em funcionamento > documentação abrangente;

Colaboração com o cliente > negociação de contratos;

Responder a mudanças > seguir um plano.

Incentiva estruturação e atitudes em equipe que facilitem a comunicação; enfatiza a entrega rápida do software operacional e diminui a importância dos artefatos intermediários; põe o cliente como parte da equipe de desenvolvimento, elimina o "nós e eles"; reconhece que o plano de projeto deve ser flexível.

12 Princípios de Agilidade:

1. A prioridade é satisfazer o cliente por meio da entrega adiantada e contínua de software;
2. Acolha bem pedidos de alterações, mesmo atrasados no desenvolvimento;
3. Entregue softwares em funcionamento frequentemente, dando preferência a intervalos curtos;
4. O comercial e desenvolvedores devem trabalhar em conjunto diariamente em todo o projeto;
5. Construa projetos com indivíduos motivados, dê o ambiente e apoio necessários e confie neles para ter o trabalho feito;
6. O método mais eficiente de transmitir informações é uma conversa aberta, de forma presencial;
7. Software em funcionamento é a principal medida de progresso;
8. Os processos ágeis promovem desenvolvimento sustentável, desenvolvedores devem manter um ritmo constante;
9. Atenção contínua para excelência técnica e bons projetos aumenta a agilidade;
10. Simplicidade é essencial;
11. As melhores arquiteturas, requisitos e projetos emergem de equipes que se auto organizam;
12. A intervalos regulares, a equipe se avalia para tornar-se mais eficiente e ajustar seu comportamento de acordo.

Método Extreme Programming (XP):

Programação Extrema.

Surgiu nos EUA em 1990. Traz mudanças incrementais e feedback rápido, e considera a mudança positiva, parte do processo. Qualidade > rapidez: ganhos a curto prazo pelo sacrifício da qualidade não são compensados pelas perdas a médio e longo prazo. Jogo de Planejamento = semanalmente, se reunir com o cliente para ver o que será desenvolvido. O cliente diz as necessidades e a equipe estima quais podem ser feitas naquela semana. Ao final da semana, são entregues ao cliente.

Metáfora = conhecer a linguagem do cliente e saber se comunicar sem deixá-lo de fora.

Equipe Coesa = o cliente faz parte da equipe de desenvolvimento, a equipe deve eliminar barreiras de comunicação.

Reuniões em Pé = reuniões rápidas, objetivas e efetivas.

Design Simples = não emperiquitar demais. Fazer o que o cliente precisa, não o que o desenvolvedor gostaria que ele precisasse.

Versões Pequenas = a liberação de pequenas versões do sistema ajuda o cliente a testar as funcionalidades de forma contínua.

Ritmo Sustentável = trabalhar não mais que oito horas por dia. Horas extras só com produtividade, mas não pode ser rotina.

Posses Coletiva = o código não tem dono e não é necessário pedir permissão a ninguém para modificá-lo.

Programação em Pares = a programação é feita por duas pessoas em cada computador, em geral um programador experiente e um aprendiz. O aprendiz deve usar a máquina, enquanto o experiente ajuda a evoluir suas capacidades. O código sempre será verificado por pelo menos duas pessoas, reduzindo drasticamente a possibilidade de erros.

Padrões de Codificação = estabelecer padrões de codificação, de forma que o código pareça desenvolvido pela mesma pessoa.

Testes de Aceitação = testes planejados e conduzidos pela equipe com o cliente, para verificar se os requisitos foram atendidos.

Desenvolvimento Orientado a Teste = antes de programar, definir e implementar os testes pelos quais se deverá passar.

Refatoração = não fugir da refatoração quando necessária. Ela permite manter a complexidade do código em um nível gerenciável, além de ser um investimento que traz benefícios em médio e longo prazo.

Integração Contínua = nunca esperar até o final do ciclo para integrar uma nova funcionalidade. Assim que viável, deverá ser integrada ao sistema para evitar surpresas.



Método Feature Driven-Development (FDD):

Desenvolvimento Dirigido por Funcionalidade.

Criado em 1997, com ênfase em orientação a objetos, desenvolvimento incremental e iterativo. Faz entregas rápidas, assim que a função tá pronta já pode entregar, e faz testes de software.

Dividido em concepção e planejamento (planejamento, identificação do problema, pensar na melhor solução, 1 a 2 semanas) e construção (desenvolvimento, construção do projeto, 1 a 2 semanas).

Concepção e Planejamento:

DMA (Desenvolver Modelo Abrangente) = é feita a modelagem dos dados, com orientação a objetos.

CLF (Construir Lista de Funcionalidades) = identificar funcionalidades que o sistema deve ter, o que deve ser feito.

PPF (Planejar por Funcionalidade) = definir o planejamento dos ciclos, o que será feito e quando será feito, as prioridades.

Construção:

DPF (Detalhar por Funcionalidade) = realizar o design orientado a objetos do sistema.

CPF (Construir por Funcionalidade) = construir e testar o software utilizando linguagem e técnica de teste orientadas a objetos.

Método Dynamic Systems Development Method (DSDM):

Desenvolvimento de Sistemas Dinâmicos.

Ênfase em desenvolvimento incremental e iterativo, entregas rápidas, testes de software, autonomia dos desenvolvedores, ritmo sustentável e participação do usuário com feedbacks. Basicamente o mesmo que o FDD, mas sem a necessidade da orientação a objetos. Usa o Princípio de Pareto. Inicia pelo estudo e implementação de 20% dos requisitos mais determinantes para o sistema, mais complexos ou de maior risco. É dividido em pré-projeto, ciclo de vida, e pós-projeto.

Pré-Projeto = o projeto é identificado e negociado, seu orçamento é definido e um contrato é assinado.

Ciclo de Vida = fase de análise de viabilidade e de negócio. Depois entra em ciclos iterativos de desenvolvimento.

Pós-Projeto = período de operação ou manutenção. A evolução do software é vista como uma continuação do processo de desenvolvimento, podendo, inclusive, retomar fases anteriores, se necessário.

Método Crystal Clear:

Criado por Alistair Cockburn em 1997. Abordagem ágil para equipes de até oito pessoas, com um designer líder e dois a sete programadores. Propõe radiadores de informação (como quadros e murais), acesso a especialistas de domínio, eliminação de distrações, cronograma de desenvolvimento e ajuste quando necessário. É dividido em iteração, entrega e projeto.

Iteração = estimativa, desenvolvimento e celebração, costuma durar poucas semanas.

Entrega = várias iterações, no espaço máximo de dois meses vai entregar funcionalidades úteis ao cliente.

Projeto = conjunto de todas as entregas.

Possui 7 pilares = entregas frequentes, melhoria reflexiva, comunicação osmótica (equipe em uma sala, todos podem se auxiliar mutuamente), segurança pessoal (falar sem medo de repreensões), foco, acesso fácil a especialistas e ambiente tecnológico.

Método Adaptive Software Development (ASD):

Desenvolvimento de Software Adaptativo.

Desenvolvimento = sistema complexo. Possui agentes (desenvolvedores e clientes), ambientes (organizacional, tecnológico e de processo) e saídas emergentes (produtos sendo desenvolvidos), e é dividido em três fases: especular, colaborar e aprender.

Especulação = planejamento de ciclos adaptáveis. Determinar o tempo do projeto, quantidade, duração e objetivos dos ciclos, componentes a serem desenvolvidos, tecnologias necessárias e listas de tarefas.

Colaboração = realizar as atividades mais previsíveis e as naturalmente incertas, simultaneamente.

Aprender = revisão de qualidade. Repetidas revisões de qualidade e testes de aceitação com o cliente e especialistas do domínio.

Método Scrum:

Usado para gestão de projetos de software, pode ser integrado a outros métodos ágeis, no desenvolvimento de softwares e no ambiente de trabalho. Um conceito importante é o sprint, um ciclo de desenvolvimento que vai de duas semanas a um mês. O scrum se divide em três fases: planejamento, ciclos e encerramento.

Primeira Fase = planejamento geral, se estabelecem os objetivos do projeto e da arquitetura do software.

Segunda Fase = ocorre uma série de ciclos de sprint, cada ciclo desenvolve um incremento do sistema.

Terceira Fase = encerra-se o projeto, completa-se a documentação exigida e avaliam-se as lições aprendidas.

Três papéis importantes no scrum:

Scrum Master (pacificador) = responsável por manter o time em um ambiente propício para concluir o projeto, é um facilitador, solucionador de conflitos.

Product Owner (dono do produto) = responsável pela voz do cliente, por garantir a qualidade. Indica os requisitos mais importantes em cada sprint.

Scrum Team (time scrum) = responsável pelo desenvolvimento e entender os requisitos do product owner. É a equipe de desenvolvimento. Em geral são recomendadas equipes de seis a dez pessoas.

Product Backlog = lista com as funcionalidades a serem implementadas em cada projeto. Não precisa ser completa no início, apenas as funcionalidades mais evidentes. Em cada sprint a equipe prioriza os elementos a serem implementados e transfere esses elementos para o Sprint Backlog, ou seja, a lista de funcionalidades a serem implementadas no ciclo que se inicia.

Product Backlog					
Código	Nome	Importância	Estimativa de esforço	Como demonstrar	Notas
1	Depósito	30	5	Logar, abrir página de depósito, depositar R\$ 10,00, ir para a página de saldo e verificar que ele aumentou em R\$ 10,00.	Precisa de um Diagrama de Sequência UML. Não há necessidade de se preocupar com criptografia por enquanto.
2	Ver extrato	10	8	Logar, clicar em <i>Transações</i> . Fazer um depósito. Voltar para <i>Transações</i> , ver que o depósito apareceu.	Usar paginação para evitar consultas grandes ao BD. Design similar para visualizar da página de usuário.

Stand Up Meeting = reunião diária que atualiza o time sobre o andamento do sprint. Dura no máximo 15 minutos, e todo o time participa, respondendo três questões: o que eu fiz ontem? O que farei hoje? Quais obstáculos estou enfrentando?

Reunião de Revisão = reunião ao final de cada sprint para avaliar o produto e processos do trabalho. Dura no máximo 4 horas, e o time apresenta as entregas para o product owner, que avalia se atendem os requisitos.

Reuniões de Retrospectiva = reunião para analisar os sprints concluídos e identificar melhorias. Dura no máximo 3 horas, e o time conversa, avalia o que deu certo e errado, o que pode ser feito para reduzir erros. Identifica melhorias que podem resultar em simples acordos entre os membros para fazer as coisas de forma diferente e também na definição de regras formais.

Projetos de Arquitetura de Software:

Para construir uma casa, precisa projetar a planta. Para construir um software, precisa projetar a arquitetura de software.

A arquitetura não é o software operacional, mas uma representação para analisar o atendimento dos requisitos, a estrutura do sistema, os componentes de software, as propriedades visíveis e as relações entre eles.

É dividida em: projeto de dados; derivação da estrutura da arquitetura do sistema; análise de estilos ou padrões de arquitetura

alternativos; implementação da arquitetura utilizando-se um modelo de processos.

Projeto de Dados = são definidas as entidades externas com que o software irá interagir, podendo ser adquiridas durante o levantamento de requisitos.

Derivação Da Estrutura Da Arquitetura Do Sistema = são identificados arquétipos arquiteturais, que representam abstrações de elementos do sistema.

Análise De Estilos Ou Padrões De Arquitetura Alternativos = é definido o padrão de arquitetura de software a ser implementado.

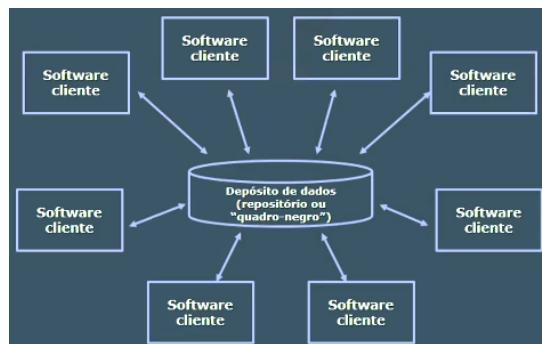
Implementação Da Arquitetura Utilizando-Se Um Modelo De Processos = desenvolvimento do software.

O projeto é um processo iterativo composto por projeto conceitual e projeto técnico:

Projeto Conceitual = apresenta ao cliente o que o sistema fará. Havendo aprovação, é traduzido num modelo mais detalhado, dando origem ao projeto técnico.

Projeto Técnico = diz aos desenvolvedores quais o hardware e software necessários para resolver o problema do cliente. A função desse projeto é descrever a forma que o sistema terá.

Arquitetura Centralizada a Dados:



Arquitetura de Fluxo de Dados:



Arquitetura em Camadas:



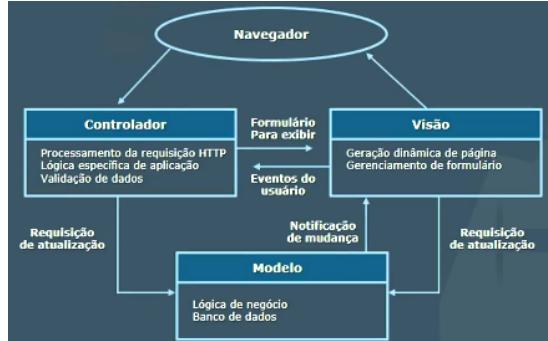
Arquitetura Cliente-Servidor:



Arquitetura Orientada a Objetos:



Arquitetura Modelo, Visão e Controlador (MVC):



Requisitos:

Requisitos = descrições do que o sistema deve fazer, serviços que oferece e restrições. Reflete as necessidades do cliente.

São classificados como requisitos funcionais e requisitos não funcionais.

Requisitos Funcionais = o que o cliente precisa que o software faça. Declarações do que o sistema deve fornecer, modo como deve agir a determinadas entradas e como deve se comportar em determinadas situações. Também podem declarar explicitamente o que o sistema não deve fazer.

Requisitos Não Funcionais = o que o desenvolvedor precisa que o software faça. Restrições sobre os serviços ou funções oferecidas pelo sistema. Incluem restrições de tempo, restrições sobre o processo de desenvolvimento e restrições impostas por padrões. Se aplicam frequentemente ao sistema como um todo, em vez de às características individuais ou aos serviços.

Elicitação dos Requisitos:

O objetivo é compreender o trabalho que as partes envolvidas no processo realizam e entender como usariam o novo sistema para

apoiar seus trabalhos. Os engenheiros de software trabalham junto com os usuários finais para saber sobre o domínio da aplicação, os serviços que o sistema deve oferecer, o desempenho dos sistemas, restrições de hardware, entre outros.

Reunir informações sobre o sistema requerido e os sistemas existentes e separar os requisitos de usuário e de sistema. As fontes de informação incluem documentações, usuários do sistema, especificações de sistemas similares, entre outros.

Entrevistas = A equipe questiona sobre o sistema usado e sobre o sistema que será desenvolvido. Os requisitos surgem a partir das respostas.

Cenários = podem ser escritos como texto, suplementados por diagramas, telas, entre outros. Cada cenário geralmente cobre um pequeno número de iterações possíveis. Diferentes cenários são desenvolvidos e oferecem diversas informações em variados níveis de detalhamento sobre o sistema.

Casos de Uso = considerada uma abordagem mais estruturada de cenários. Identifica os envolvidos em uma iteração e dá nome ao tipo de iteração.

Etnografia = técnica de observação usada para compreender os processos operacionais e extraír requisitos de apoio para esses processos. Um analista faz uma imersão no ambiente de trabalho em que o sistema será usado. O trabalho do dia a dia é observado e são feitas anotações sobre as tarefas reais em que os participantes estão envolvidos.

Especificação De Requisitos:

Processo de escrever os requisitos de usuário e de sistema em um documento de requisitos. Há quatro principais métodos de especificação de requisitos.

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 suplementados 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 possam reduzir a ambiguidade em um documento de requisitos, a maioria dos clientes não comprehende uma representação formal.

Gestão de Configuração de Mudança:

Conjunto de atividades para gerenciar alterações, identificando artefatos que precisam ser alterados, estabelecendo relações entre eles, definindo mecanismos para gerenciar diferentes versões desses artefatos, controlando alterações impostas e auditando e relatando alterações feitas. Usa-se uma ferramenta automatizada para gerenciar isso. Ela garante que as versões anteriores não se percam, possam ser recuperadas. O gerenciamento de configuração divide-se em:

Controle de Versão = manter o controle das várias versões dos componentes do sistema e garantir que as mudanças feitas em componentes por diferentes desenvolvedores não interfiram com as outras.

Construção de Sistema = reunir componentes, dados e bibliotecas do programa, os compilando e ligando num sistema executável.

Gerenciamento de Mudanças = manter o controle das solicitações de mudança de clientes e desenvolvedores no software já entregue, elaborar os custos e o impacto de fazer essas mudanças e decidir se e quando as alterações devem ser implementadas.

Gerenciamento de Lançamentos (releases) = preparação de software para lançamento externo e acompanhamento das versões de

sistema lançadas para uso do cliente.

Git = ferramenta mais utilizada para gestão de configuração de mudança. Permite que vários desenvolvedores trabalhem juntos em um mesmo projeto, e guarda todas as versões que já existiram daquele projeto, não passa a borracha em nada.

Manutenção e Evolução de Software:

O software será desvalorizado com o tempo, visto que falhas são descobertas; requisitos mudam; produtos menos complexos, mais eficientes e mais avançados são disponibilizados. É necessário, então, que falhas sejam corrigidas; novos requisitos sejam acomodados; sejam buscadas simplicidade, eficiência e atualização tecnológica.

Alterações englobam = erros corrigidos; adaptação a um novo ambiente; cliente solicitando novas características ou funções; aplicação passa por um processo de reengenharia para proporcionar benefício em um contexto moderno.

Existem três tipos de manutenção de software = correção de defeitos, adaptação ambiental e adição de funcionalidade.

Análise de Pontos de Função:

Técnica para medir o tamanho do software, tudo o que ele faz, e então estimar o esforço para implementação.

Pontos de Função = não medem o esforço, produtividade, custo ou outras informações específicas. Mede o tamanho funcional do software, do aplicativo.

Tipo De Contagem = há três tipos: projeto de desenvolvimento (criação de um novo projeto); projeto de melhoria (adicionar, modificar ou eliminar funções de um projeto existente); aplicação (projeto finalizado).

Escopo Da Contagem E Fronteira Da Aplicação = o escopo de contagem diz se a contagem é em um ou mais sistemas ou em partes de um ou mais sistemas, e quais funções serão incluídas na contagem, podendo ser todas as funcionalidades, apenas as utilizadas ou algumas específicas. A fronteira da aplicação divide os componentes do aplicativo e os componentes de outros aplicativos. É a linha que separa uma aplicação da outra.

Contar Funções = identifica cinco componentes básicos: arquivos lógicos internos (ALI), arquivos de interface externa (AIE), entrada externa (EE), saída externa (SE) e consulta externa (CE), enquadrados nas categorias funções do tipo dados e funções do tipo transação.

Tipo De Dados = funcionalidade para armazenamento de dados da aplicação, caracterizadas como arquivos lógicos mantidos dentro (ALI) ou fora da aplicação (AIE).

	1 a 19 DER	20 a 50 DER	Acima de 50 DER
1 RLR	Simples	Simples	Média
2 a 5 RLR	Simples	Média	Complexa
Acima de 5 RLR	Média	Complexa	Complexa

Função	Simples	Média	Complexa
ALI	7 pontos	10 pontos	15 pontos
AIE	5 pontos	7 pontos	10 pontos

DER são os atributos das tabelas (nome, cpf, código,...)

Tipo De Transação = funcionalidades de processamento de dados do sistema, sendo classificadas em entradas externas (EE), saídas externas (SE) e consultas externas (CE). EE manipula dados originados fora da aplicação e inclui, altera ou exclui dados de um ou mais ALI. SE envia dados para fora da fronteira da aplicação, apresenta informação ao usuário com fórmulas matemáticas ou cálculos. CE envia dados para fora da fronteira da aplicação, apresenta informação ao usuário sem fórmulas matemáticas ou cálculos.

	1 a 4 DER	5 a 15 DER	Acima de 15 DER
1 ALR	Simples	Simples	Média
2 ALR	Simples	Média	Complexa
Acima de 2 ALR	Média	Complexa	Complexa

Função	Simples	Média	Complexa
EE	3 pontos	4 pontos	6 pontos
SE	4 pontos	5 pontos	7 pontos
CE	3 pontos	4 pontos	6 pontos

Contagem De Pontos De Função Não Ajustados = somar o total de pontos nos dados e nas transações.

Valor Do Fator De Ajuste = avaliar de 14 características, os itens de influência. Cada um recebe uma nota de 0 a 5, podendo variar o tamanho do software em até 35%, para mais ou menos. Esse fator, aplicado aos pontos não ajustados, obtém os pontos ajustados da aplicação.

Comunicação de dados	Funções distribuídas
Performance	Utilização do equipamento
Volume de transações	Entrada de dados on-line
Interface com o usuário	Atualizações on-line
Processamento complexo	Reusabilidade
Facilidade de implantação	Facilidade operacional
Múltiplos locais	Facilidade de mudanças

Elaborado com base em Silva e Oliveira, 2005, p. 7

$$\text{Fator de ajuste} = (\text{pontos de influência} \times 0,01) + 0,65$$

Número Dos Pontos De Função Ajustados (AFP) = multiplicar o fator de ajuste pelos pontos de função. O resultado é o tamanho funcional do software.

Duração E Custo De Um Projeto = cada linguagem demanda um esforço diferente, e suas características influenciam o esforço para produzir cada ponto de função. Para definir o tempo e custo de desenvolvimento de um projeto, em horas, por ponto de função, precisa se possuir histórico de projetos.

Pontos de Casos de Uso:

Surgiu em 1993, simplificando a análise de pontos de função. Analisa a quantidade e complexidade dos atores e casos de uso, o

que gera pontos de caso de uso não ajustados. Depois, a aplicação dos fatores técnicos e ambientais levam aos pontos de caso de uso ajustados.

Atores = definir a complexidade dos atores, contados uma única vez, mesmo que relacionados a vários casos de uso, e somar todos eles para ter o peso não ajustado dos atores.

Alta complexidade (3 pontos): humanos que interagem com o sistema por interface gráfica;

Média complexidade (2 pontos): humanos que interagem com o sistema por linha de comando e sistemas que interagem por TCP/IP;

Baixa complexidade (1 ponto): sistemas acessados por interfaces de programação (API).

Casos de Uso = definir a complexidade dos casos, contados uma única vez e somente se completos. A complexidade é medida pelo número de transações, classes e risco.

Transações:

Alta complexidade (15 pontos): mais de 7 transações;

Média complexidade (10 pontos): de 4 a 7 transações;

Baixa complexidade (5 pontos): até 3 transações.

Classes:

Alta complexidade (15 pontos): mais de 10 classes;

Média complexidade (10 pontos): de 6 a 10 classes;

Baixa complexidade (5 pontos): até 5 classes.

Risco:

Alta complexidade (15 pontos): casos não padronizados com um número desconhecido de transações, pois há regras de negócio desconhecidas, e deve-se descobrir o fluxo principal e as sequências alternativas;

Média complexidade (10 pontos): casos padronizados com um número conhecido e limitado de transações, pois a lógica de funcionamento é conhecida, mas regras de negócio obscuras podem existir;

Baixa complexidade (5 pontos): casos que não alteram dados, como relatórios que têm apenas 1 ou 2 transações.

UUCP = total dos pontos de casos de uso não ajustados, obtido somando o total do peso não ajustado dos atores com o peso não ajustado dos casos de uso.

Fator de Complexidade Técnica (TFC) = avaliar 13 fatores técnicos. Cada um recebe uma nota de 0 a 5.

FATOR TÉCNICO	PESO	FATOR TÉCNICO	PESO
Sistema distribuído	2	Performance	2
Eficiência de usuário final	1	Complexidade de processamento	1
Projeto visando código reusável	1	Facilidade de instalação	0,5
Facilidade de uso	0,5	Portabilidade	2
Facilidade de mudança	1	Concorrência	1
Segurança	1	Acesso fornecido a terceiros	1
Necessidades de treinamento	1		

Fonte: Elaborado com base em Wazlawick, 2013, p. 172

$$TCF = 0,6 + (0,01 * \text{Total da soma dos pontos de fatores técnicos})$$

Fator Ambiental Total (EF) = avaliar 8 fatores ambientais. Cada um recebe uma nota de 0 a 5.

FATOR AMBIENTAL	PESO	FATOR AMBIENTAL	PESO
Familiaridade com o processo de desenvolvimento	1,5	Experiência com a aplicação	0,5
Experiência com orientação a objetos	1	Capacidade do analista líder	0,5
Motivação	1	Estabilidade de requisitos obtida historicamente	2
Equipe em tempo parcial	-1	Dificuldade com a linguagem de programação	-1

Fonte: Elaborado com base em Wazlawick, 2013, p. 172

$$EF = 1,4 - (0,03 * \text{Total da soma dos fatores ambientais})$$

Total de Pontos de Casos de Uso Ajustados (UCP) = multiplicar o total de UUCP por TCF e EF.

Pontos de Histórias:

Estimativa de esforço preferida de métodos ágeis como Scrum e XP. Os desenvolvedores sentem no coração o tempo necessário para construirão um software e comunicam esse prazo. Com base nisso, multiplica-se o número de pessoas pela quantidade de dias para se obter o total de pontos de história. Por exemplo, se 2 pessoas levariam 6 dias para implementar determinada história de usuário, então atribuem-se à história: $2 \times 6 = 12$ pontos de história.

Source Lines Of Coud - SLOC:

Corresponde ao número de linhas do código, uma estimativa com base na opinião de especialistas e no histórico de projetos passados. Há também KSLOC para milhares de linhas, MSLOC para milhões de linhas e GSLOC para bilhões de linhas de código. Como em geral a equipe não chega a um valor único, devem ser considerados pelo menos três valores: sloc otimista (número mínimo de linhas em condições favoráveis), sloc pessimista (número máximo de linhas em condições desfavoráveis) e sloc esperado (número de linhas em situação de normalidade). O sloc é calculado então: $SLOC = (4 \times SLOC \text{ esperado} + SLOC \text{ otimista} + SLOC \text{ pessimista}) / 6$.

COCOMO:

Baseado em sloc, apresenta-se em três formas de implementação, de acordo com a complexidade e grau de informações a respeito do sistema a ser desenvolvido:

Implementação básica = quando a única informação sobre o sistema é o número estimado de linhas de código.

Implementação intermediária = quando fatores relativos a produto, suporte computacional, pessoal e processo são conhecidos.

Implementação avançada: quando necessário subdividir o sistema em subsistemas e distribuir as estimativas de esforço por fases e atividades.

Para o cálculo do esforço todas as implementações consideram também o tipo de projeto a ser desenvolvido, que pode ser:

Modo orgânico = o sistema a ser desenvolvido é de baixa complexidade e a equipe está acostumada. Baixo risco tecnológico e de pessoal.

Modo semi destacado = o sistema é novidade para a equipe. Médio risco tecnológico e de pessoal.

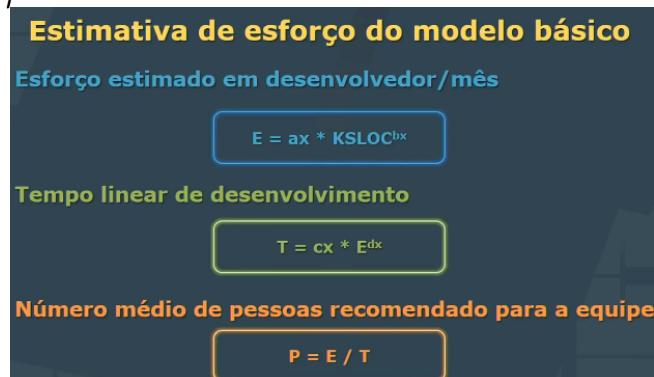
Modo embutido = o sistema tem alto grau de complexidade ou é embarcado, e a equipe tem dificuldade. Alto risco tecnológico e de pessoal.

Assim, é possível conhecer:

E = esforço estimado em desenvolvedor/mês;

T = tempo linear de desenvolvimento;

P = número de pessoas para a equipe.



TIPO	ax	bx	dx	
Orgânico	2,4	1,05	2,5	0,38
Semidestacado	3,0	1,12	2,5	0,35
Embutido	3,6	1,2	2,5	0,32

Estimativa de esforço dos modelos intermediário e avançado

- Considera 15 fatores influenciadores de custo

Fatores influenciadores de custo	Acrônimo	Muito baixa	Baixa	Média	Alta	Muito alta	Extra alta
Relativos ao produto							
Nível de confiabilidade requerida	RELY	0,75	0,88	1,00	1,15	1,40	
Dimensão da base de dados	DATA		0,94	1,00	1,08	1,16	
Complexidade do produto	CPLX	0,70	0,85	1,00	1,15	1,30	1,65
Supporte computacional							

Estimativa de esforço dos modelos intermediário e avançado

Cálculo dos influenciadores de custo

$$EAF = RELY * DATA * CPLX * TIME * ... * SCED$$

Estimativa de esforço

$$E = ai * KSLOC^{bi} * EAF$$

TIPO	ai	bi
Orgânico	2,8	1,05
Semidestacado	3,0	1,12
Embutido	3,2	1,2

Testes de Software:

Processo para saber se o software está funcionando como deveria.

Erro (error) = diferença entre o resultado obtido de um processo e o resultado correto ou esperado.

Defeito (fault) = linha de código, bloco ou conjunto de dados incorretos, provocam um erro.

Falha (failure) = não funcionamento do software, possivelmente por um defeito.

Engano (mistake) = ação que produz um defeito no software.

Verificação = analisar o software para ver se está de acordo com o especificado.

Validação = analisar o software para ver se atende às necessidades dos interessados.

Teste = permite realizar a verificação e validação do software.

Em maioria, o teste define um conjunto de dados de entrada a ser testado com os resultados esperados de cada dado, executa o programa e compara os resultados obtidos com os resultados esperados.

Dados de Teste = os dados inseridos no programa para realizar o teste.

Caso de Teste = os dados inseridos no programa para realizar o teste e o resultado esperado, correto.

Teste de Funcionalidade:

Vê se as funções implementadas estão corretas, dividindo-se em testes de unidade, integração, sistema e aceitação.

Testes de Unidade = verificar se as unidades foram implementadas corretamente.

Testes de Integração = integrar as unidades para gerar uma nova versão do sistema. Há a integração big-bang (diferentes componentes são construídos separadamente e integrados no final), integração bottom-up (integra os módulos de nível mais baixo, que não dependem de nenhum outro, e depois os de nível imediatamente mais alto), integração top-down (integra inicialmente os módulos de nível mais alto, deixando os mais básicos para o fim), integração sanduíche (integra os módulos de nível mais alto em top-down e os de nível mais baixo em bottom-up).

Testes de Sistema = verifica se a atual versão do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário e é capaz de obter os resultados esperados.

Testes de Aceitação = teste realizado pelo cliente para validação da aplicação desenvolvida.

Teste Estrutural:

Também chamado teste de caixa branca, os testes são executados com código-fonte. Detecta grande quantidade de possíveis erros pela garantia de ter executado pelo menos uma vez todos os comandos e condições do programa. É dividido em critérios baseados em complexidade (complexidade ciclomática e caminhos linearmente independentes) e em fluxo de controle.

Complexidade Ciclomática =

Métrica que mede a complexidade do programa. A complexidade ciclomática é o número de estruturas de seleção e repetição no programa + 1. As estruturas de seleção são if (1 ponto), elif (1 ponto), for (1 ponto), repeat (1 ponto), case (1 ponto por opção, exceto otherwise), or ou and nas estruturas citadas (1 ponto), e a pontuação se divide em fácil (<= 10), médio (11 a 20), alto risco (21 a 50) e não testáveis (>= 50).

Caminhos Linearmente Independentes =

Qualquer caminho do programa com um conjunto de instruções ou nova condição. Para análise, representamos como grafo de fluxo de controle (GFC), obtido colocando todos os comandos em nós e os fluxos de controle em arestas. Comandos em sequência podem ser colocados em um único nó, e estruturas de seleção e repetição devem ser representadas por meio de nós distintos com arestas que indiquem a decisão e a repetição, quando for o caso. O valor da complexidade ciclomática diz o número máximo de caminhos para exercitar todos os comandos do programa.

Fluxo de Controle =

Usa GFC em todos-nós, todas-arestas e todos-caminhos. Se não for possível um caso de teste que alcance um nó, uma aresta ou um possível caminho, um defeito potencial foi identificado.

Todos-Nós = a execução do programa passa em cada vértice do GFC, cada comando é executado ao menos uma vez.

Todas-Arestas = a execução do programa passa em cada aresta do GFC, cada desvio de fluxo de controle é executado ao menos uma vez.

Todos-Caminhos = a execução do programa passa em cada caminho do GFC.

Teste Funcional:

Também chamado teste de caixa preta, os testes são feitos com as entradas e saídas, sem conhecimento do código-fonte.

Particionamento Em Classes De Equivalência:

O particionamento de equivalência considera a divisão das entradas de um programa:

Entradas válidas num intervalo de valores (exemplo, 10 a 20) = 1 conjunto válido (10 a 20) e 2 inválidos (< 10 e > 20).

Entradas válidas como quantidade de valores (exemplo, 5 elementos) = 1 conjunto válido (5 elementos) e 2 inválidos (-5 elementos e +5 elementos).

Entradas válidas num conjunto de valores que podem ser tratados de forma diferente (exemplo, strings "masculino" e "feminino") = 1 conjunto válido para cada um e 1 conjunto inválido para outros valores quaisquer.

Entradas válidas numa condição do tipo "deve ser de tal forma" (exemplo, data final posterior à inicial) = 1 conjunto válido (quando a condição é verdadeira) e 1 inválido (quando a condição é falsa).

Análise De Valor-Limite =

Considerar valores fronteirícios com outras classes de equivalência:

Condicão de entrada com intervalo de valores = teste imediatamente subsequentes, explore classes inválidas vizinhas.

Condicão de entrada com quantidade de valores = teste com +1 e -1 valor do que o permitido.

Error-Guessing = os testes que o desenvolvedor vai fazendo intuitivamente enquanto programa o software.

Teste Baseado em Defeitos:

Um dos critérios mais eficazes na detecção de defeitos, consiste em criar um clone do software e inserir um erro proposital nele.

O resultado é testado: se o software original apresentar o mesmo resultado, é porque ele está com um erro.

Teste de Mutação = alterações no programa testado, cada alteração simula um defeito no sistema, criando um conjunto de programas mutantes. Cada alteração no programa dá origem a um novo programa mutante. As etapas são:

Geração dos mutantes = são feitos os mutantes nos programas clones. Devem ser tantos quanto for possível.

Execução do programa em teste = executar o programa em teste e verificar se o seu comportamento é o esperado.

Execução dos mutantes = executar todos os programas mutantes, e comparar o resultado com o obtido no programa original.

Análise dos mutantes vivos = analisar o que houve de errado no programa que teve resultados iguais ao mutante e consertar.

DevOps:

Dev - desenvolvimento = equipe responsável pela identificação dos requisitos com o cliente, análise, projeto, codificação e testes.

Ops - operações = equipe responsável pela implantação em produção, monitoramento e solução de incidentes e problemas.

DevOps é uma cultura colaborativa entre as equipes para entregar o software de forma ágil, segura e estável. Se apoia em quatro pilares:

Colaboração = trabalhar com pessoas com diferentes experiências e um propósito comum.

Afinidade = construir relações fortes entre os times, todos juntos nos objetivos, empatia e aprendizagem contínua.

Ferramentas = são como um acelerador, impulsionando a mudança com base na cultura atual.

Escala = leva em conta como os pilares podem ser aplicados às mudanças da organização, em questões técnicas e culturais.

As equipes ganham capacidade de responder às necessidades dos clientes, aumentar a confiança nos aplicativos e cumprir as

metas mais rapidamente. Isso influencia as fases do planejamento, desenvolvimento, entrega e operação:

Planejamento = idealizar, definir e descrever recursos e funcionalidades dos sistemas. Criar lista de pendências, acompanhar bugs, gerenciar o desenvolvimento de software ágil, usar quadros, visualizar o progresso com dashboards.

Desenvolvimento = codificação (gravação, teste, revisão, integração e compilação do código). Inovar sem sacrificar a qualidade, estabilidade e produtividade. Usar ferramentas produtivas, automatizar etapas e manuais e iterar pequenos incrementos..

Entrega = implantação do aplicativo e configuração da infraestrutura, de maneira consistente e confiável. Definir gerenciamento de versão com estágios claros.

Operação = manter, monitorar e solucionar problemas dos aplicativos. Garantir confiabilidade do sistema, alta disponibilidade e tempo de inatividade igual a zero, reforçando segurança e governança. Identificar problemas antes que afetem o cliente.

Princípios do DevOps:

Desenvolver e testar sistemas como os em produção;

Implantar o aplicativo com processos confiáveis, ágeis e repetíveis;

Monitorar e validar a qualidade operacional;

Amplificar os loops de feedback.

Integração Contínua:

Prática do desenvolvimento em que cada participante do time integra seu trabalho diariamente. Cada integração é verificada por um build automatizado, que detecta erros de imediato e permite que o desenvolvimento tenha mais qualidade e agilidade.

Princípios da integração contínua = repositório de origem único; automatização; autoteste de construção (testes unitários); todos fazem commit (gravar o software com as implementações) diário; todo commit é centralizado em uma máquina de integração; corrigir quebra de código imediatamente; manter a construção rápida; testar em ambiente o mais próximo possível do de produção; todos têm o executável mais recente; todos vêem o que acontece; automatizar a implantação.

Práticas da integração contínua = integrar o código diariamente; iniciar automaticamente a segmentação de implementação, a cada mudança de código, e executar validações, análise estática de padrões de codificação e testes; se uma versão falhar, corrigir em até 10 minutos.

Entrega Contínua = evolução natural da integração contínua. Disponibiliza mudanças de forma segura e rápida, garantindo que o código esteja sempre pronto para implantação, mesmo diante dos desenvolvedores fazendo alterações diariamente.

Implantação Contínua = evolução natural da entrega contínua. Implantação automática após execução com sucesso dos testes automatizados e validações previstas. Se tá funcionando, já pode entregar. Usada muito em aplicações web e aplicativos.

Recursos DevOps:

Conduzir = estabelecer objetivos e ajustá-los com base no feedback dos clientes. Inclui planejamento contínuo do negócio.

Desenvolver/Testar = envolve desenvolvimento colaborativo (profissionais trabalham juntos, com práticas e plataforma comum para criar e entregar o software, o centro é a integração contínua) e teste contínuo (testar cedo e continuamente o software, resulta em custos reduzidos, ciclos de teste curtos e feedback contínuo).

Implantar = liberação e implantação contínuas, lançar novos recursos para clientes e usuários o mais rápido possível.

Operar = envolve monitoramento contínuo (fornecer dados e métricas pro time de operações, desenvolvimento, controle de qualidade

e linhas de negócios) e feedback e otimização contínua do cliente (permite melhorar os aplicativos e a experiência do cliente).

Implantando Devops:

Abrange toda a organização, incluindo proprietários, arquitetura, design, desenvolvimento, qualidade, operações, segurança, parceiros e fornecedores, cada um com predisposições, experiências e preconceitos únicos. É necessário calma e cuidado ao implementar uma nova cultura. As seis técnicas de implantação do DevOps são:

Melhoria contínua = visa resultados cada vez melhores. Para que aconteça, é necessário focar no que melhorar, ter métricas para medir o desempenho e padronizar as tarefas.

Planejamento de liberação = visa oferecer recursos aos clientes em prazos menores e exatos, com práticas ágeis que auxiliam no planejamento desses prazos e o foco na qualidade.

Integração contínua = reduz o risco e identifica os problemas no início do ciclo de vida do desenvolvimento do software.

Entrega contínua = visa entregar resultados aos clientes sempre que possível.

Teste contínuo = há três testes que possibilitam teste contínuo: de provisionamento do ambiente e configuração; de gerenciamento de dados; de integração, função, desempenho e segurança.

Monitoramento e feedback contínuos = aparece de diferentes formas, por meio de solicitações, reclamações e classificações.

As maiores fontes de ineficiências se resumem em:

Sobrecarga desnecessária = necessidade de comunicar várias vezes a mesma informação.

Retrabalho desnecessário = defeitos descobertos em testes ou na produção, forçando a equipe a voltar ao desenvolvimento.

Superprodução = funcionalidades desenvolvidas que não foram requeridas.