



# **Engenharia de Requisitos e Processos de *Software***



**Cruzeiro do Sul Virtual**  
Educação a distância



# Material Teórico



Introdução ao Processo de *Software*

**Responsável pelo Conteúdo:**

Prof. Me. Artur Marques

**Revisão Textual:**

Prof.<sup>a</sup> Dr.<sup>a</sup> Selma Aparecida Cesarin



# UNIDADE

## Introdução ao Processo de *Software*



- Definição de Processo de *Software*;
- *SDLC: Software Development Life Cycle* (Ciclo de Vida do Desenvolvimento de *Software*);
- Modelos Tradicionais de Processo de *Software*;
- Conceito de Agilidade de *Software* ;
- *Scrum*;
- *XP, TDD e FDD*.



### OBJETIVOS DE APRENDIZADO

- Introduzir os conceitos e os Processos de *software* tradicionais e ágeis;
- Saber diferenciá-los e qual utilizar conforme a necessidade.





# Orientações de estudo

Para que o conteúdo desta Disciplina seja bem aproveitado e haja maior aplicabilidade na sua formação acadêmica e atuação profissional, siga algumas recomendações básicas:



Assim:

- ✓ Organize seus estudos de maneira que passem a fazer parte da sua rotina. Por exemplo, você poderá determinar um dia e horário fixos como seu “momento do estudo”;
- ✓ Procure se alimentar e se hidratar quando for estudar; lembre-se de que uma alimentação saudável pode proporcionar melhor aproveitamento do estudo;
- ✓ No material de cada Unidade, há leituras indicadas e, entre elas, artigos científicos, livros, vídeos e sites para aprofundar os conhecimentos adquiridos ao longo da Unidade. Além disso, você também encontrará sugestões de conteúdo extra no item **Material Complementar**, que ampliarão sua interpretação e auxiliarão no pleno entendimento dos temas abordados;
- ✓ Após o contato com o conteúdo proposto, participe dos debates mediados em fóruns de discussão, pois irão auxiliar a verificar o quanto você absorveu de conhecimento, além de propiciar o contato com seus colegas e tutores, o que se apresenta como rico espaço de troca de ideias e de aprendizagem.

# Definição de Processo de Software

Um Processo de *Software*, ou se você preferir, Metodologia de *Software* é um conjunto de atividades inter-relacionadas que leva à produção/construção do *software*.

Essas atividades podem levar ao desenvolvimento do *software* a partir do zero ou à modificação de um sistema existente/legado.

É interessante notar que o Processo de *software* realmente é conhecido no Mercado com muito, mas muitos nomes mesmo, a ponto de deixar você confuso.

Vamos conhecer alguns sinônimos:

- O próprio Processo de *Software*;
- Ciclo de Vida de Desenvolvimento de *Software*;
- Ciclo de Vida de Desenvolvimento de Sistemas (SDLC);
- Processo de Desenvolvimento de *Software*;
- Modelo de Processos de *Software*;
- Ciclo de vida do *Software*;
- Paradigmas de Desenvolvimento de *Software*.

Trata-se, portanto, de um modelo escolhido para gerenciar a criação de *software* desde o início, isto é, desde os desejos do cliente, ou seja, dos requisitos, até a liberação do produto acabado, testado e validado.

Claro, há definições canônicas dos grandes nomes da Engenharia de *Software*.

Vamos conhecer algumas:

Para Peters e Pedrycz (2001, p. 29): “Uma sequência de etapas com *feedback* que resultam na produção e na evolução de softwares”.

Pfeegler (2004, p. 36) entende que se trata de “[...] tarefas que são realizadas geralmente na mesma ordem todas as vezes, ordenadamente, envolvendo várias etapas que possuem atividades, restrições, recursos para alcançar a saída desejada”.

Isso realmente significa que se você tem dois Processos que constroem *software*, é o que eles têm em comum, todo o restante desenvolve um novo caminho, uma nova abordagem, que vai levar você ao mesmo resultado, “fazer *software*”, mas levando em consideração cada situação específica.

Afinal, um produto de *software*, cujos requisitos requerem um desenvolvimento ágil, será muito mais trabalhoso e custoso se for feito utilizando um Processo tradicional e sequencial.

Isso não quer dizer que esse último não seja útil, e ele é muito, mesmo nos dias atuais!

Depende do escopo: “o que eu tenho que fazer e quanto complexo é isso para ser feito no tempo e com os recursos que posso” (definição do autor).

Sommerville (2011, p. 28) descreve Processo de *software* como sendo: “[...] um conjunto de atividades relacionadas que levam à produção de um produto de *software*”.

Qualquer Processo de *Software* deve incluir as quatro atividades a seguir:

- **Especificação de Software/Requisitos:** define as principais funcionalidades do *software* e as restrições em torno delas;
- **Projeto e implementação:** devemos projetar/planejar e criar/codificar o *software*;
- **Verificação e validação:** o que será produzido a partir da codificação deve estar de acordo/conforme as especificações e, claro, atender às necessidades do cliente;
- **Evolução e manutenção:** modificações realizadas para atender às mudanças nos requisitos do cliente e do Mercado.

Veja bem, trata-se de uma visão de Processo bastante sumarizada e simplista, porque, na vida real, há muitas subatividades, inclusive as que descrevem o próprio Processo, por exemplo:

- **Produtos:** os resultados de uma atividade, por exemplo: documento de requisitos, arquitetura de solução, modelagem de dados, matriz de riscos, cartões de histórias dos usuários etc.;
- **Papéis:** nesse caso, quero me referir às responsabilidades das pessoas envolvidas no Processo. Por exemplo: arquiteto de dados, Engenheiro de *Software*, scrum master, product owner, programadores, analistas de negócios, clientes e usuários etc.;
- **Pré/pós condições:** trata-se das condições iniciais que devem ser verdadeiras antes e as depois, ou seja, ao final de uma atividade, por exemplo, não posso fazer um documento de arquitetura de solução sem os requisitos e as regras de negócios elicitadas terem sido aprovadas pelo cliente.

## *SDLC: Software Development Life Cycle* **(Ciclo de Vida do Desenvolvimento de Software)**

Segundo Swersky (2018), o ciclo de vida de desenvolvimento de *software* SDLC é uma terminologia usada para explicar como o *software* é entregue a um cliente em uma série de etapas. Essas etapas levam o *software* da fase de ideação para a entrega.

O *software*, como todos os produtos, começa como uma ideia. Seja um documento, diagrama ou *software* funcional, o artefato criado em uma etapa torna-se a entrada para a próxima etapa. A sequência de etapas usada por esses métodos é comumente chamada de Ciclo de Vida de Desenvolvimento de *Software*.

A primeira versão de uma aplicação de *software* raramente é “finalizada”, ou dada como terminada, e entregue simplesmente. Quase sempre há recursos adicionais e correções de bugs esperando para serem projetadas, desenvolvidas e implementadas, e isso independe dos Processos: se tradicional ou ágil.

Dessa forma, **SDLC** é o termo mais genérico utilizado para dizer, Métodos de Desenvolvimento de *Software*, sendo que o que varia de um para o outro são as etapas e a ordem com que são executadas, mas, em todos eles, há em comum o conceito de ciclo que pode ser único e crescente como no Cascata ou os que trabalham em ciclo menores, porém em grande número para a conclusão, como é o caso dos ágeis.

Quando temos um Plano Estruturado para abordar desenvolvimento de *software* na forma de um **SDLC** de Mercado ou desenvolvido internamente, temos muitos benefícios:

- Um vocabulário comum para cada etapa do **SDLC**;
- Canais de Comunicação definidos entre as Equipes de Desenvolvimento e as partes interessadas (*stakeholders*);
- Funções e responsabilidades claras entre desenvolvedores, *designers*, analistas de negócios e Gerentes de Projeto;
- Entradas e saídas claramente definidas de um passo para o próximo;
- Uma “definição de feito”, determinação que pode ser usada para confirmar se uma etapa está realmente completa. Vale aqui informar que estamos tratando de algo que gera muita polêmica nos Meios de Desenvolvimento. Trata-se da *definition of done*, como acabamos de traduzir, definição de pronto, porque o que é pronto para um não é para o usuário, por exemplo. Para o usuário, algo está pronto quando o funcionamento é eficiente e eficaz; já para o pessoal de desenvolvimento, algo pronto pode ser a *sprint* ou a funcionalidade que passou pelos testes de unidade e foi encaminhada para validação pelo usuário. Claro que não é assim, necessariamente, mas se trata de um exemplo simples. Todavia, é fundamental que a Equipe do Projeto, juntamente com o sponsor e o usuário determinem a sua definição de pronto para que todos estejam na mesma página e no mesmo momento quando os momentos da verdade da qualidade chegarem.

Claro que, em se tratando de um Processo de fabricação de *software*, temos alta complexidade e que justifica a quebra em várias fases que devem ser devidamente controladas e alimentar outras entradas de Processos.

Essas etapas são muito parecidas de uma metodologia para outra. Elas tendem a ocorrer numa mesma ordem, embora também possam ser misturadas, de modo que várias etapas ocorram em paralelo. Por exemplo, os Processos Ágeis tendem a unir esses passos em um ciclo fechado e de repetição rápida, métodos formais já tendem a ser mais lineares e irem passo a passo.

Swersky (2018, p. 4-5) descreve as 7 fases do **SDLC** geral da seguinte maneira:

Planejamento: A fase de planejamento envolve aspectos de gerenciamento de projetos e produtos. Isso pode incluir:

- Alocação de recursos (humanos e materiais);
- Planejamento de capacidade;
- Agendamento de projetos;

- Estimativa de custo;
- Provisionamento.

Os resultados da fase de planejamento incluem:

- Planos de projeto;
- Cronogramas;
- Estimativas de custos; e
- Requisitos de aquisição.

O ideal é que os gerentes de projeto e a equipe de desenvolvimento colaborem com as equipes de operações e segurança para garantir que todas as perspectivas sejam representadas.

**Requisitos:** O negócio deve se comunicar com as equipes de TI para transmitir seus requisitos para novos desenvolvimentos e aprimoramentos. A fase de requisitos reúne esses requisitos de partes interessadas do negócio e especialistas no assunto.

- Arquitetos;
- Equipes de desenvolvimento; e
- Gerentes de produto.

Trabalham para documentar os Processos de negócios que precisam ser automatizados por meio de *software*. A saída dessa fase em um projeto tradicional tipo Cascata é geralmente um documento que lista esses requisitos. Processos ágeis, por outro lado, podem produzir um *backlog* de tarefas a serem executadas.

**Design e prototipagem:** Uma vez que os requisitos sejam compreendidos, os arquitetos e desenvolvedores de *software* podem começar a projetar o *software*. O Processo de *design* utiliza padrões estabelecidos para arquitetura de aplicativos e desenvolvimento de *software*.

Os arquitetos podem usar uma estrutura de arquitetura, como o TOGAF (*The Open Group Architecture Framework* é uma estrutura para arquitetura empresarial que fornece uma abordagem para projetar, planejar, implementar e governar uma arquitetura de tecnologia da informação corporativa. Normalmente, ele é modelado em quatro níveis: negócios, aplicativos, dados e tecnologia.), para compor um aplicativo a partir de componentes existentes, promovendo a reutilização e a padronização.

Os desenvolvedores usam Padrões de *Design Patterns* comprovados para resolver problemas algorítmicos de maneira consistente. Essa fase também pode incluir alguns protótipos rápidos, também conhecidos como *spikes*, para comparar soluções para encontrar o melhor ajuste. A saída desta fase inclui:

- Criar documentos que listam os padrões e componentes selecionados para o projeto;

- Código produzido por picos, usado como ponto de partida para o desenvolvimento.

**Desenvolvimento de Software:** Esta fase produz o *software* em desenvolvimento. Dependendo da metodologia, esta fase pode ser conduzida em “*sprints*” com *time-box* (Processos Ágeis) ou pode prosseguir como um único bloco de esforço (Processo Tradicional Cascata/*Waterfall*). Independentemente da metodologia, as equipes de desenvolvimento devem produzir *software* de trabalho o mais rápido possível. As partes interessadas do negócio devem estar envolvidas regularmente, para garantir que suas expectativas sejam atendidas. A saída dessa fase é um *software* funcional e testável.

**Teste:** A fase de testes do SDLC é indiscutivelmente uma das mais importantes. É impossível fornecer *software* de qualidade sem testes. Existe uma grande variedade de testes necessários para medir a qualidade:

- Qualidade do código:
  - » Teste unitário (testes funcionais);
  - » Teste de integração;
  - » Teste de performance;
  - » Testes de segurança.

A melhor maneira de garantir que os testes sejam executados regularmente e nunca ignorados por conveniência é automatizá-los. Os testes podem ser automatizados usando ferramentas de Integração Contínua, como *Codeship*, por exemplo. A saída da fase de teste é um *software* funcional, pronto para implementação em um ambiente de produção.

**Implantação:** A fase de implantação é, idealmente, uma fase altamente automatizada. Em empresas de alta maturidade, esta fase é quase invisível; O *software* é implantado no instante em que está pronto. Empresas com menor maturidade, ou em algumas indústrias altamente regulamentadas, o Processo envolve algumas aprovações manuais. No entanto, mesmo nesses casos, é melhor que a implantação seja totalmente automatizada em um modelo de implementação contínua. As ferramentas de automação de liberação de aplicativos (ARA) são usadas em empresas de médio e grande porte para automatizar a implantação de aplicativos em ambientes de produção. Os sistemas ARA geralmente são integrados às ferramentas de Implementação Contínua. A saída desta fase é o lançamento para a produção de *software* de trabalho.

**Operações e Manutenção:** A fase de operações e manutenção é o fim, por assim dizer. O Ciclo de Vida de Desenvolvimento de *Software* não termina aqui. O *software* deve ser monitorado constantemente para garantir uma operação adequada. Bugs e defeitos descobertos na Produção devem ser reportados e respondidos, o que muitas vezes alimenta o trabalho de volta ao Processo. Correções de bugs podem não fluir durante todo o ciclo, no entanto, pelo menos um Processo abreviado é necessário para garantir que a correção não introduza outros problemas (conhecidos como regressão). (SWERSKY, 2018, p. 4-5)

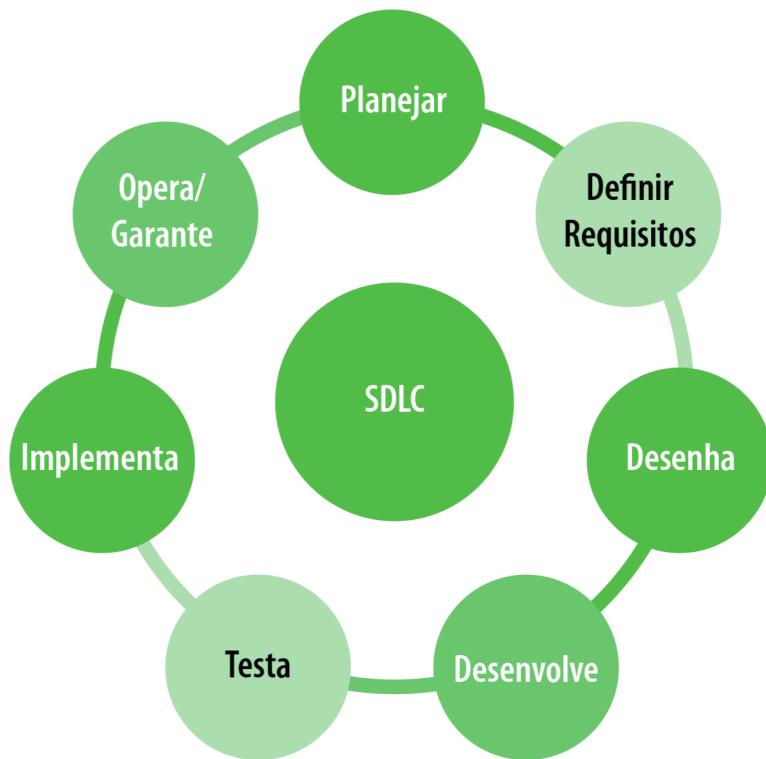


Figura 1 – Fases do SDLC. Durante a leitura de livros de Engenharia de *Software*, você poderá se deparar com SDLC com 5 fases (requisitos, desenho, desenvolvimento, testes e produção), há uma simplificação. Com o passar do tempo e o amadurecimento da Área, percebe-se que há necessidade de estarmos com mais 2 fases, como você pode notar, comparando o escrito nesse comentário versus a imagem acima

## Modelos Tradicionais de Processo de *Software*

### Cascata/Cachoeira ou Waterfall

O Modelo em Cascata é uma abordagem sequencial, em que cada atividade fundamental de um Processo é representada como uma fase separada, organizada em ordem linear.

No Modelo em Cascata, você deve planejar e agendar todas as atividades antes de começar a trabalhar nelas. Podemos chamar isso de um Processo orientado pelo Plano.

O Processo orientado pelo Plano é aquele em que todas as atividades são planejadas primeiro e o progresso é medido em relação ao plano, enquanto no Processo Ágil, o Planejamento é incremental e é mais fácil alterar o Processo para refletir as mudanças de requisitos.

As fases do modelo em cascata são: **Requisitos, Design, Implementação, Teste e Manutenção.**

Para tentar facilitar o seu entendimento nos Modelos Cascata, o escopo é fixo (plano), porém o tempo pode variar, às vezes muito, porque no caso de o *software* ser baseado em requisitos variáveis por causa de características de negócio, haverá maior dispêndio de tempo e de custos, portanto, caso os requisitos variem, a tendência é um *software* constantemente revisitado, muito ajustado e com grande correções, porém, caso o escopo e seus requisitos sejam perenes no tempo, é o modelo ideal para se desenvolver.

Ainda em dúvida... bom vamos melhorar o exemplo então!

Imagine que você trabalha na Indústria Automobilística como desenvolvedor de *software* e você foi incumbido de desenvolver um Sistema que estará embarcado nesse veículo e sua função é controlar a ignição e a mistura de combustível e ar de maneira ideal para que o motor funcione de maneira adequada. Claro que o pessoal de Engenharia Mecânica e Eletrônica colocou uma série de sensores e atuadores que fazem com que o carro tenha praticamente a mesma eficiência com etanol e gasolina.

Ficou assustado com o escopo inicial?!

Mas, não precisa!

Trata-se de um típico caso de desenvolvimento em cascata, porque os requisitos já foram definidos pela Engenharia e o *Hardware* (motor) deve se comportar de forma adequada constantemente.

Não vai haver nenhum usuário no meio do projeto querendo colocar um pistão a mais ou usar água de combustível. Portanto, tudo está muito claro, então, é só seguir as fases do **SDLC** em Cascata e fazer as coisas acontecerem.

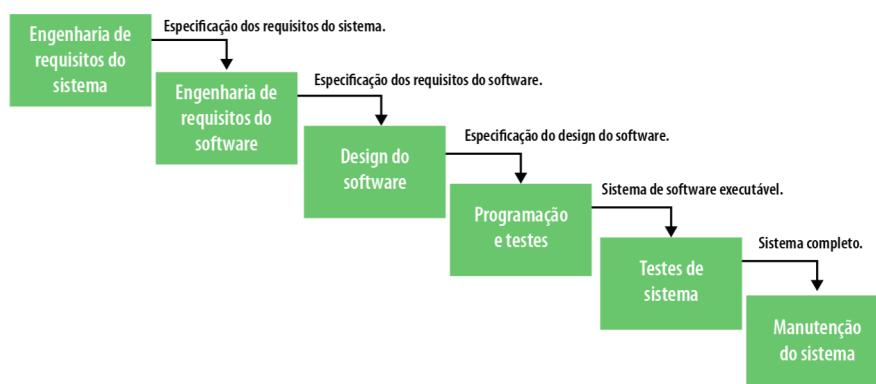


Figura 2 – Modelo em Cascata

Vantagens do modelo em cascata:

- É simples e fácil de entender e usar;
- É fácil de gerenciar: devido à rigidez do modelo, cada fase tem resultados específicos, bem como um Processo de revisão;

- Nesse modelo, as fases são processadas e completadas uma de cada vez. Fases não se sobrepõem;
- O modelo em cascata funciona bem para projetos menores, nos quais os requisitos são claramente definidos e muito bem compreendidos.

Desvantagens do modelo em cascata

- Uma vez que uma aplicação está em fase de teste, é muito difícil voltar atrás e mudar algo que não foi bem pensado na fase de conceito;
- Nenhum software de trabalho é produzido até bem mais tarde, durante o Ciclo de Vida;
- Altos níveis de risco e incerteza;
- Não é um bom modelo para projetos complexos e orientados a objetos com certeza;
- Não se trata de um bom modelo para projetos longos e/ou em andamento;
- Não é adequado para os projetos em que os requisitos apresentam um risco moderado a alto de alteração.

## ***Big Bang***

---

Esse modelo não possui uma técnica específica para trabalhar no projeto de desenvolvimento de software. O projeto pode começar com apenas uma quantia básica de dinheiro e recursos, portanto, envolve princípios menos planejados e nenhum método formal é seguido.

Podemos utilizá-lo em casos em que o cliente não tem certeza sobre seus desejos e os requisitos não são bem analisados ou definidos.

Os requisitos são entendidos e implementados conforme começam a chegar. Trata-se, porém, de um modelo que tende a ser muito mais arriscado do que outros modelos.

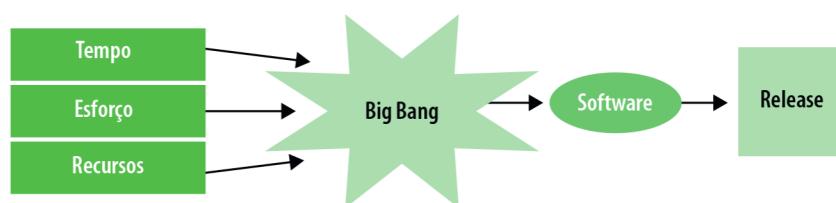


Figura 3 – Modelo de desenvolvimento de Software *Big Bang*

### **Vantagens:**

- O modelo é extremamente simples;
- Há muito pouco pré-planejamento que é necessário e, às vezes, o desenvolvimento começa sem nenhum planejamento;
- Os recursos necessários são muito poucos a princípio;
- É muito fácil de gerenciar;

- Usando esse modelo, tem-se uma exposição adequada para iniciantes que procuram desenvolver suas habilidades.

#### Desvantagens:

- É um modelo de alto risco e é por isso que não é recomendado para projetos complexos e grandes;
- Se os requisitos do cliente não forem compreendidos, o Projeto corre o risco de ser descartado e reiniciado, mesmo sendo pequeno e simples.

## Evolucionário/Evolutivo

O **modelo evolutivo** é uma combinação do Modelo Iterativo e Incremental do **SDLC**, e divide o ciclo de desenvolvimento em modelos em cascata menores e incrementais, nos quais os usuários podem obter Acesso em: ao produto no final de cada ciclo.

O *feedback* é fornecido pelos usuários no produto para o estágio de Planejamento do próximo ciclo, e a equipe de desenvolvimento responde, geralmente, alterando o produto, o plano ou o Processo. Portanto, o produto de *software* evolui com o tempo.

Sabemos que todos os modelos de *software* tradicionais têm desvantagens relativas à duração do tempo até a entrega final, porém, esse modelo aborda esses problemas de forma diferente.

O modelo evolutivo sugere a divisão do trabalho em partes menores, priorizando-as e as entregando ao cliente, uma a uma.

A principal vantagem é que a confiança do cliente aumenta à medida que ele constantemente obtém bens ou serviços quantificáveis desde o início do Projeto, para verificar e validar suas necessidades. O modelo permite a alteração de requisitos, bem como todo o trabalho dividido em blocos sustentáveis:

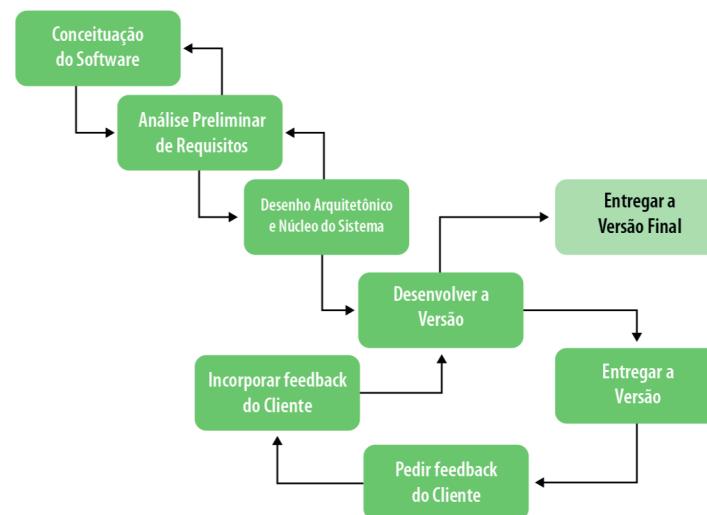


Figura 4 – Representação gráfica do modelo evolutivo

Pankaj explica que a aplicação do modelo evolucionário se aplica em circunstâncias como:

Grandes projetos onde você pode encontrar facilmente módulos para implementação incremental. O modelo evolutivo é comumente usado quando o cliente quer começar a usar os recursos principais em vez de esperar pelo *software* completo.

Também é usado no desenvolvimento de *software* orientado a objetos porque o sistema pode ser facilmente dividido em unidades em termos de objetos. (PANKAJ, 2019)

---

### Vantagens:

- Análise de risco é melhor; suporta mudanças nos requisitos. o tempo de funcionamento inicial é menor; mais adequado para Projetos grandes e de missão crítica;
- Durante o ciclo de vida, o *software* é produzido antecipadamente, o que facilita a avaliação ou *feedback* do cliente.

### Desvantagens:

- Não é adequado para Projetos de menor porte;
- Complexidade de Gerenciamento é maior;
- O final do Projeto pode não ser conhecido, o que é um risco;
- Recursos altamente qualificados são necessários para a análise de risco;
- O progresso do Projeto é altamente dependente da fase de análise de risco.

---

## Espiral

O modelo em espiral é um dos mais importantes Modelos de Ciclo de Vida de Desenvolvimento de e foi o primeiro a oferecer suporte para tratamento de riscos. Um risco é qualquer situação adversa que possa afetar a conclusão bem-sucedida de um Projeto de *Software*.

Esse modelo suporta o enfrentamento dos riscos, fornecendo o escopo para construir um protótipo em todas as fases do desenvolvimento do *software*.

Em sua representação, lembra uma espiral com muitos loops. O número exato de voltas da espiral é desconhecido e pode variar de Projeto para Projeto.

Cada loop da espiral é chamado de fase do Processo de Desenvolvimento de *Software*, e esse número exato de fases necessárias para desenvolver o produto pode ser variado pelo Gerente do Projeto, dependendo dos riscos do Projeto, portanto, ele determina dinamicamente o número de fases e isso nos remete ao Papel Fundamental do Gerente no Desenvolvimento de um *software*, usando o modelo espiral.

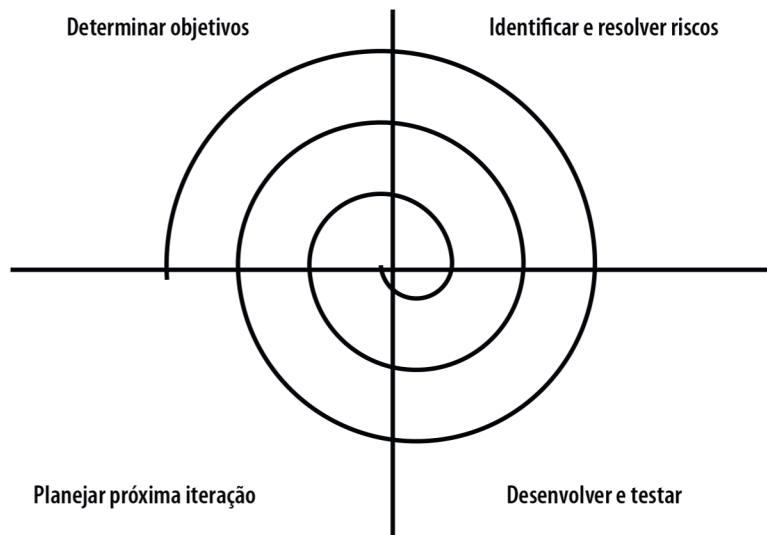


Figura 5 – Modelo de Engenharia de Software baseado na Espiral de Boehm. Em cada fase do Modelo de Espiral, as características do produto datadas e analisadas e os riscos nesse momento são identificados e resolvidos por meio de prototipagem

Esse modelo tem as seguintes fases:

- **Determinar os objetivos e identificar as soluções:** os requisitos são coletados dos clientes e os objetivos são identificados, elaborados e analisados no início de cada fase. Então, soluções alternativas possíveis para a fase são propostas;
- **Identificar e resolver os riscos:** durante essa fase, todas as possíveis soluções são avaliadas para selecionar a melhor possível. Em seguida, os riscos associados a essa solução são identificados e os riscos são resolvidos usando a melhor estratégia possível. No final dessa fase, um protótipo é construído;
- **Desenvolver e testar a próxima versão do produto:** durante essa fase, os recursos identificados são desenvolvidos e verificados por meio de testes. No final, a próxima versão do software estará disponível;
- **Planejar a próxima iteração:** os clientes avaliam a versão até então desenvolvida do software e, ao final, o Planejamento para a próxima fase é iniciado.

#### Vantagens do Modelo Espiral:

- **Manipulação de Risco:** os projetos com muitos riscos desconhecidos que ocorrem à medida que o desenvolvimento prossegue. Nesse caso, é o melhor modelo de desenvolvimento a ser seguido devido à análise e o manuseio de risco em todas as fases;
- **Bom para grandes projetos:** recomenda-se usar esse modelo em Projetos grandes e complexos;
- **Flexibilidade nos requisitos:** as solicitações de mudança nos requisitos na fase posterior podem ser incorporadas com precisão usando esse modelo;

- **Satisfação do cliente:** o cliente pode ver o desenvolvimento do produto na fase inicial do desenvolvimento do software e, assim, habituou-se ao Sistema, usando-o antes de concluir o produto total.

#### Desvantagens do Modelo atual:

- **Complexidade:** é muito mais complexo que outros modelos **SDLC**;
- **Caro:** não é adequado para pequenos projetos, pois é caro;
- **Muito confiável na análise de risco:** a conclusão bem-sucedida do Projeto é muito dependente da análise de risco. Sem experiência muito altamente experiente, será um fracasso desenvolver um Projeto usando esse Modelo;
- **Dificuldade no Gerenciamento do tempo:** como o número de fases é desconhecido no início do Projeto, a estimativa de tempo é muito difícil.

## Prototipagem

---

É um Processo de Desenvolvimento de Sistemas no qual um protótipo, ao qual nos referimos como sendo uma aproximação antecipada de um sistema ou produto final, é construído, testado e então retrabalhado conforme necessário, até que um protótipo aceitável seja finalmente obtido a partir do qual o Sistema ou o produto completo agora pode ser desenvolvido.

Esse modelo funciona melhor em cenários em que nem todos os requisitos do Projeto são conhecidos em detalhes com antecedência. É um Processo Iterativo de tentativa e erro, que ocorre entre os desenvolvedores e os usuários.

Conforme os escritos de Rouse, esse Processo possui as seguintes etapas:

- Os novos requisitos do sistema são definidos com o máximo de detalhes possíveis. Isso geralmente envolve entrevistar vários usuários que representam todos os departamentos ou aspectos do sistema existente.
- Um *design* preliminar é criado para o novo sistema.
- Um primeiro protótipo do novo sistema é construído a partir do projeto preliminar. Este é geralmente um sistema reduzido e representa uma aproximação das características do produto final.
- Os usuários avaliam cuidadosamente o primeiro protótipo, observando seus pontos fortes e fracos, o que precisa ser adicionado e o que deve ser removido. O desenvolvedor coleta e analisa as observações dos usuários.
- O primeiro protótipo é modificado, baseado nos comentários fornecidos pelos usuários, e um segundo protótipo do novo sistema é construído.
- O segundo protótipo é avaliado da mesma maneira como foi o primeiro protótipo.
- As etapas anteriores são iteradas quantas vezes forem necessárias, até que os usuários estejam satisfeitos de que o protótipo representa o produto final desejado.

- O sistema final é construído com base no protótipo final.
- O sistema final é cuidadosamente avaliado e testado. A manutenção de rotina é realizada de forma contínua para evitar falhas em grande escala e minimizar o tempo de inatividade. (ROUSE, 2018, p. 2)

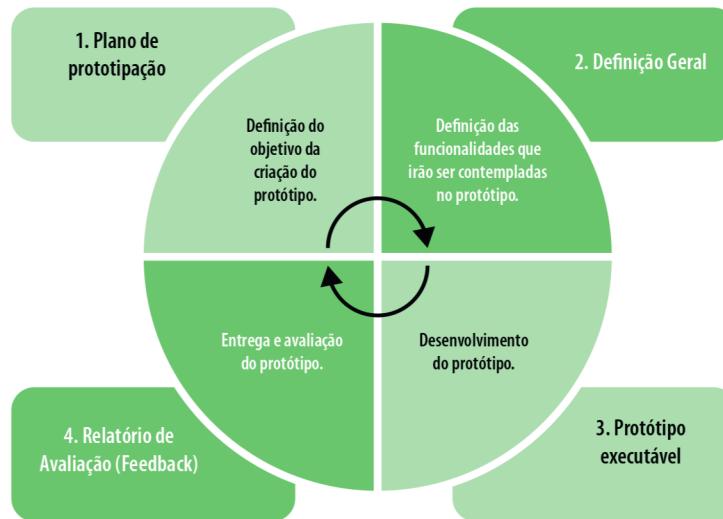


Figura 6 – Processo de Prototipagem e suas fases

#### Vantagens do Processo de Prototipagem:

- **Prototipagem envolve seu usuário:** uma das vantagens mais notáveis da prototipagem é que ela inclui o usuário. Inerente à prototipagem é o envolvimento do usuário. Eles desfrutam da experiência de estarem envolvidos no desenvolvimento, além de poderem participar com uma versão operacional de seu Projeto;
- Protótipos reduzem os mal-entendidos e evitam expectativas não atendidas. Além disso, eles ajudam a acumular *feedback* do usuário desde as primeiras fases do desenvolvimento. Com protótipos, os usuários podem rapidamente fornecer seus pensamentos, solicitar uma alteração no Projeto e modificar detalhes do modelo;
- **Prototipagem economiza dinheiro:** a prototipagem ajuda a poupar, melhorando a capacidade de detectar alterações necessárias no início do Projeto. Consequentemente, o tempo e os custos gerais de um projeto são reduzidos. Como resultado da prototipagem, podem ser previstas áreas de custo que não foram percebidas antecipadamente. As mudanças que precisam ser feitas podem ser resolvidas antes que o Processo de ajuste se torne grande e caro. Problemas, tanto imediatos quanto de longo prazo, podem ser isolados e corrigidos.

#### Desvantagens do Processo de Prototipagem:

- **Os usuários podem pensar que o protótipo é a versão final:** talvez a maior frustração com Processo de Prototipagem é que, às vezes, os usuários entendem mal que seja o produto final. Os usuários podem ficar irritados com uma versão inicial do produto e não querer usar uma cópia posterior aprimorada e depois reaprimorada, porque acham que têm as mesmas falhas da versão que usaram;

- Frequentemente, os protótipos são usados para projetar e construir Projetos físicos. Em muitas circunstâncias, porém, os Projetos são na verdade um *software*, que possui inúmeras variáveis de ajuste e sistematizações indeterminadas, por exemplo, esse é o caso de sites e aplicativos móveis e grandes portais. Esse modelo pode ser um esforço importante e benéfico para essas situações.

## Incremental

O Modelo Incremental é um Processo de Desenvolvimento de *Software* em que os requisitos são divididos em vários módulos independentes do Ciclo de Desenvolvimento de *Software*. É feito em etapas desde o projeto de análise, implementação, teste e verificação e manutenção.

Para efeito didático, vamos chamar cada incremento de iteração e cada iteração passa obrigatoriamente pelas fases de análise de requisitos, *design*, codificação e teste, sendo que cada lançamento subsequente deve adicionar essa função à iteração anterior, portanto, é outra versão. Isso se repete até que toda a funcionalidade esteja implementada e, portanto, o Sistema terminado.

O Sistema é colocado em produção quando o primeiro incremento é entregue. Esse primeiro incremento é, geralmente, um produto principal em que os requisitos básicos são abordados e os recursos complementares são adicionados nos próximos incrementos.

Uma vez que o produto principal é analisado pelo cliente, há desenvolvimento de Plano para o próximo incremento.



Ilustração do Modelo Incremental. Disponível em: <http://bit.ly/2nZ44GT>



Podemos notar pela figura acima que o Modelo de Processo Incremental aplica sequências lineares (como no Modelo Cascata) de forma escalonada, à medida que o tempo for avançando. Cada uma das sequências lineares gera um incremento do *software*.

O que você precisa entender desse Processo é que o desenvolvimento do Sistema é dividido em muitos miniprojetos. Então, Sistemas Parciais são sucessivamente construídos para produzir a solução final, sendo que os requisitos de prioridade mais alta são atacados primeiro; quando desenvolvido, ele é paralisado na iteração, ou seja, não se mexe mais nele.

### Vantagens:

- O *software* será gerado rapidamente durante o **SDLC**;
- É flexível e menos dispendioso para alterar requisitos e/ou escopo;
- Esse modelo é menos dispendioso em comparação ao Cascata;

- Um cliente pode responder a cada iteração: fraciona-se o risco porque o foco fica com um cliente por vez;
- Erros são fáceis de serem identificados.

### Desvantagens:

- Requer um bom Planejamento;
- Os problemas podem aparecer por causa da forma com que o Processo é desenvolvido, afinal, é por iteração, e você pode fracionar por Área ou Setor, de tal forma que nem todos os requisitos sejam coletados;
- Cada fase de iteração é rígida e não se sobrepõe;
- A correção de um problema em uma unidade requer correção em todas as unidades e isso consome muito tempo.

## Iterativo

É um Processo que desenvolve o software de tal forma que se concentra, principalmente, no crescimento preliminar e no *design* e vai ganhando velocidade lentamente, até que atenda aos requisitos, para que se consiga chegar ao final de um produto de *software* completamente construído e funcional. Portanto, trata-se aqui, de uma abordagem focada em segmentar qualquer grande Processo de Desenvolvimento de *Software* em partes menores. Então, não há o objetivo de estabelecer um plano de especificação completo, mas é feito para iniciar com requisitos mínimos, especificando e implementando apenas uma parte do *software*. Temos aqui, novamente, a figura do protótipo que é, então, revisado para inserção de requisitos adicionais, se for o caso.

Na prática, então, assume uma forma iterativa para criar uma nova versão do aplicativo, e assim sucessivamente, até seu final.

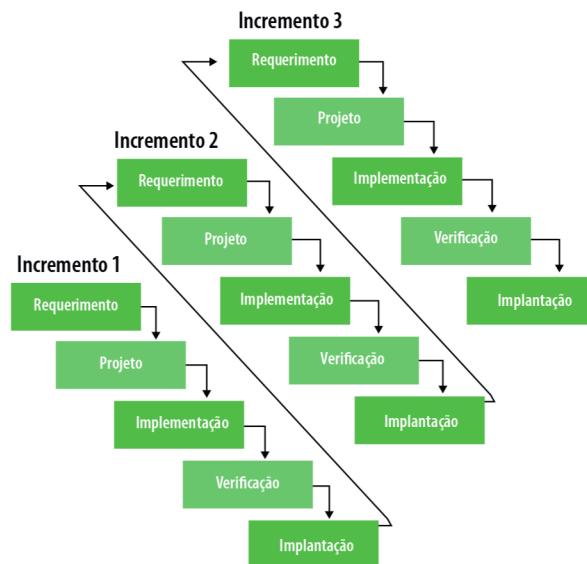


Figura 7 – Ciclo de vida do Desenvolvimento Iterativo



Na prática, cada iteração deve entregar uma parte funcional do *software*, que passará por todas as etapas de desenvolvimento, desde a elaboração dos requisitos até a implantação. Como a cada incremento é possível gerar uma versão praticável do *software* para o cliente, fica mais fácil e mais rápido obter *feedbacks* constantes.

As fases para esse Processo são:

- **Requisitos:** as informações relacionadas ao sistema são coletadas e analisadas. Depois disso, os requisitos coletados são planejados de acordo com o desenvolvimento do sistema;
- **Projeto:** a solução de *software* está preparada para atender às necessidades do projeto. O *design* do Sistema pode ser novo ou a extensão de um anterior/legado;
- **Implementação e Teste:** o Sistema é desenvolvido por meio da codificação e da construção da interface de usuário e módulos, que são, então, incorporados e testados;
- **Revisão:** o *software* é estimado e verificado de acordo com o requisito atual. Em seguida, outros requisitos são revisados discutidos e revisados para propor uma atualização na próxima iteração. Por fim, surgirá uma situação em que, ao implementar esse modelo, os requisitos do Sistema serão concluídos e, em seguida, o *software* poderá ser distribuído ou entregue ao cliente.

Onde esse modelo se aplica?

- Os requisitos do Sistema podem ser classificados e compreendidos de forma clara;
- Se uma nova Tecnologia precisar de compreensão prévia, esse modelo pode ser útil para conhecer a Tecnologia mais recente e incrementar ou atualizar o modelo de acordo;
- É útil quando existem altos riscos nas características e metas do Sistema;
- Situações em que os recursos com conjuntos de habilidades exigidos não estão acessíveis, e o sistema precisa ser desenvolvido em uma base contratual.

#### Vantagens:

- Produz *software* funcional rapidamente, ao longo do **SDLC**;
- Fornece maior flexibilidade e aprimora a base de requisitos;
- É simples de testar e reparar com uma pequena iteração, quando necessário.

#### Desvantagens:

- Para se implementar esse modelo, são necessários mais recursos humanos (pleno e sênior) e materiais;
- Não serve para projetos de *software* pequenos;
- Também é fortemente dependente da análise de riscos;
- Definir incrementos por requerer uma definição completa do Sistema;
- Certamente, maior atenção com relação ao Gerenciamento é necessária.

## RUP

**RUP**, significa em português, Processo Unificado do *Rational* e se trata de um Processo de Desenvolvimento de Software da *Rational* que foi comprada e hoje é uma divisão da IBM, desde 2003.

Ele divide o Processo de Desenvolvimento em quatro fases distintas, cada uma envolvendo modelagem de negócios, análise e *design*, implementação, teste e implantação.

As quatro fases são:

- **Iniciação:** temos a ideia para o Projeto declarada, bem como a Equipe de Desenvolvimento determina se vale à pena fazer o Projeto e quais recursos serão necessários;
- **Elaboração/Planejamento:** quando a arquitetura do Projeto e os recursos necessários são avaliados, e os desenvolvedores consideram possíveis aplicações do software e os custos associados ao desenvolvimento;
- **Construção:** o Projeto é desenvolvido e concluído, ou seja, o software é projetado, escrito e testado;
- **Transição:** é quando o software é liberado para a produção e também quando os ajustes finais ou atualizações são realizadas com base no *feedback* dos usuários finais

RUP fornece uma maneira estruturada para as empresas fazerem software, porque fornece um plano específico para cada etapa do Processo de Desenvolvimento, e ajuda a evitar que os recursos sejam desperdiçados, trabalhando fortemente na redução de custos inesperados de desenvolvimento.

Pense que, nesse caso, **RUP** não é um modelo de desenvolvimento concreto, mas destina-se a ser adaptável às necessidades específicas de um projeto, time ou Empresa.



Temos uma imagem que exibe cada uma das fases e também as disciplinas do RUP. O ciclo iterativo ao qual nos referimos consiste em passar por cada uma das fases. Em cada fase, há disciplinas mais ou menos atuantes, conforme ilustram as cores de cada disciplina na imagem. As fases do RUP são: *Concepção/Iniciação*, *Elaboração*, *Construção* e *Transição*. Vários ciclos nessas fases se repetem até o fim do projeto do sistema. De cada um desses ciclos, espera-se a criação de um *release*, um produto liberado para uso. Essa liberação pode ser interna ou externa, mas espera-se que seja funcional. Veja o RUP e suas Fases (representadas como colunas) e Disciplinas (representadas pelas linhas). Disponível em: <http://bit.ly/2nJlbeF>

Há várias práticas embarcadas nesse Processo, como, por exemplo:

- Desenvolver software iterativamente;
- Gerenciamento forte de requisitos;
- Usar arquitetura baseada em componentes;

- Modelagem de software visual usando UML;
- Preocupação com a verificação da qualidade do software;
- Controle de alterações no software, portanto, versionamento e gestão de mudança.

**Vantagens:**

- Capacidade adaptativa em lidar com requisitos em constante mudança;
- Enfatiza a necessidade de documentação essencial e precisa.

**Desvantagens:**

- Precisa de membros da Equipe com bastante experiência, preferencialmente especialistas;
- A integração ao longo do Processo pode ser prejudicial durante os testes ou outras fases, pois podem se tornar conflitantes e atrapalhar atividades tidas como mais fundamentais;
- É um modelo conhecido por sua complexidade, portanto é um desafio para times pequenos, por vezes, não se mostrando funcional.

## Conceito de Agilidade de Software

Conforme os escritos de Powell-Morse (2017) por volta de 2001 surgiu a partir de várias reuniões de um grupo de 17 desenvolvedores, uma metodologia inovadora de desenvolvimento de software.

A partir de uma assembleia, surgiu o que mais tarde veio a se chamar de Manifesto Ágil de Software.

Em linhas gerais, eles pregavam o seguinte:

- Enfatiza a necessidade de cada projeto ser tratado de maneira diferente com base nas necessidades individuais do Projeto, cronograma e no time;
- Se concentra menos em requisitos e muito mais nas abstrações para permitir maior flexibilidade ou agilidade durante o SDLC.

E isso se fundamenta nos seguintes valores pregados pelo manifesto:

- Indivíduos e interações ao invés de enfatizar sistemas e ferramentas;
- Software de trabalho, embora a documentação possa certamente ser muito benéfica durante o desenvolvimento. Portanto, software funcional;
- Colaboração com o cliente;
- Resposta rápida à mudança.

## Scrum

O Scrum é um Processo de Gerenciamento de Projetos de *Software* que faz uso da Metodologia de Desenvolvimento de *Software* Ágil. Essa estrutura é flexível e incentiva formas colaborativas de gerenciar um projeto de *software*.

Um dos benefícios mais reconhecidos dessa Metodologia de Desenvolvimento de *Software* é que ela permite que você se mova rapidamente e também mude facilmente quando necessário.

Ciclos de *feedback* mais rápidos e a capacidade de reconhecer problemas precocemente tornaram o Scrum uma das metodologias mais populares do mundo.

O desenvolvimento se divide em várias fases, sendo que cada uma delas resulta em um produto pronto para uso e, ao final de cada etapa, um produto funcional de *software* sempre é entregue a um cliente.

Chamamos esse ciclo ou essas etapas de entrega funcional ao término de *Sprint*. O *feedback* do cliente ajuda a revelar possíveis problemas ou a alterar o Plano Inicial, se necessário, porque é fato que há um cliente localmente alocado junto ao time Scrum.

Os principais papéis em uma equipe Scrum são:

- **Dono do produto (Product Owner):** ele cuida dos interesses do usuário final;
- **Mestre Scrum (Scrum Master):** deve coordenar todo o Processo e também garantir que o Scrum seja usado corretamente. Ele também realiza as reuniões Scrum com o time;
- **Time Scrum:** são os desenvolvedores do produto de *software* e suas principais tarefas são programação, análise e testes, entre outros.

Scrum possui como fases principais:

- **Criação do backlog do produto:** trata-se de uma lista de recursos a serem implementados durante o desenvolvimento, segue uma ordem de prioridade sendo que cada um desses recursos possui o nome de história do usuário;
- **Planejamento de sprint e criação de backlog da sprint:** aqui é determinada a duração de uma *sprint*, sendo a média de cada *sprint*, em termos de Mercado, 15 dias. Depois, selecionam-se as histórias de usuários relevantes e que serão desenvolvidos nessa *sprint*. Dessa forma, temos o backlog da *sprint*;
- **Desenvolvimento da sprint e reuniões:** codificação, uso de quadros como o Kanban para, visualmente, sabermos o progresso do desenvolvimento, assim como todos os dias, são realizadas reuniões chamadas de *Daily Scrum Meeting*, que são feitas para serem curtas e eficazes. 15 minutos, todos em pé, com perguntas do tipo: o que você já fez, o que você fará hoje e quais os problemas;
- **Teste e validação do produto da sprint:** o resultado do desenvolvimento de cada *sprint* é *software* funcional, portanto, ele é apresentado e validado;
- **Retrospectiva e planejamento da próxima sprint:** aqui se discutem os resultados obtidos e como melhorar com as lições aprendidas para a próxima etapa.



Ciclo de um Processo SCRUM de desenvolvimento:. Disponível em: <http://bit.ly/2oYNZky>



*Sprints* podem possuir variação de tempo de 1, 2, 3 ou 4 semanas, mas uma vez definida, ele pereniza. Todavia, *Scrum* não permite alterações, uma vez definido o *sprint backlog* e, depois que a reunião de planejamento de *sprint* é concluída e um compromisso é feito para entregar um conjunto de itens de *backlog* de produto, esse conjunto de itens permanece inalterado até o final do *sprint*. Quem prioriza o *backlog* do produto é o *Product Owner*, mas o time de desenvolvimento determina a sequência de desenvolvimento.

## XP, TDD e FDD

### Programação Extrema – XP

É um método flexível que permite que alterações sejam feitas nos requisitos. Baseia-se em 4 atividades principais, que são **Audição** (entendimento de histórias de usuários), **Design, Codificação e Testes**. Todos eles se reúnem para criar um software que se beneficia com os requisitos do cliente.

**XP** é melhor quando é necessário criar um software em um ambiente instável, portanto, desenvolvedores que trabalham com **XP** são capazes de entregar em prazos mais curtos do que a maioria das outras metodologias, além de incentivar o feedback de seus clientes que são compilados quando estamos testando o Sistema e, a partir de entradas de outros desenvolvedores trabalhando no mesmo projeto, além, é claro, diretamente do cliente.

#### Vantagens

- Reduzir os custos envolvidos no desenvolvimento de software;
- O envolvimento do cliente é considerado importante para o sucesso do XP;
- Enfatiza os planos práticos e os cronogramas;
- Processo simplificado de como os desenvolvedores dedicam seu tempo ao projeto;
- Está alinhado aos melhores métodos de desenvolvimento atuais;
- Os desenvolvedores podem criar softwares que atendam a todos os padrões.

#### Desvantagens

- O custo de alterar os requisitos iniciais em um estágio posterior pode ser muito alto;
- Sucesso muito dependente das pessoas envolvidas no Processo, quanto mais comprometida e mais clara a visão, melhor o resultado final;
- Exige que os vários níveis de pessoas envolvidas se encontrem e revisem com frequência o que foi feito, o que é um dreno no orçamento;

- Mudanças de desenvolvimento regulares podem ser difíceis para um desenvolvedor médio ou iniciante lidar;
- O esforço necessário e os requisitos são difíceis de estimar no início do Projeto.

## Desenvolvimento Orientado por Recurso - FDD

---

Mohan (2017) utiliza o método iterativo, juntamente com a tecnologia baseada em objetos. Isso tem favorecido equipes maiores de desenvolvedores, que trabalham em projetos que estão progredindo do trabalho em fases para essa abordagem iterativa.

Esse modelo trabalha acompanhando os marcos (data de entrega de funcionalidades) que são mantidos para cada Estágio do progresso no desenvolvimento de *software* com suas 5 etapas:

- Desenvolver o modelo completo;
- Construir a lista de recursos;
- Planejar;
- Projetar;
- Desenvolver por característica.

O **FDD** começa por uma revisão do escopo do Sistema antes que os modelos de domínio sejam criados em grande detalhe para cada recurso e depois revisados novamente. Os modelos de domínio selecionados são, então, mesclados em um modelo único geral.

### Vantagens:

- É capaz de mover projetos maiores facilmente com um grande grau de sucesso;
- Segue os padrões estabelecidos pela indústria de desenvolvimento de *software* e, portanto, incorpora as melhores práticas da indústria.

### Desvantagens:

- Esforço empreendido não é viável para pequenos projetos;
- Desenvolvedores individuais não podem trabalhar em um padrão tão complexo de desenvolvimento;
- Clientes não conseguem obter o *software* por escrito dos desenvolvedores, impedindo-os de obter provas de seu *software*.

## Desenvolvimento Orientado a Teste – TDD

---

O TDD pode ser definido como uma prática de programação que instrui os desenvolvedores a escrever um novo Código apenas se um teste automatizado falhar. Isso evita a duplicação de código. E torna o código mais claro, simples e livre de erros. Ele inicia com o *design* e o desenvolvimento de testes para cada pequena funcionalidade de um aplicativo, então, primeiro, o teste é desenvolvido, especifica

e valida o que o código fará, o que é uma abordagem de trás para frente, já que, num Processo normal, primeiro geramos o código e depois o testamos.

Como se trata de um teste, pode falhar, até porque, nessa abordagem, os testes são desenvolvidos antes mesmo da etapa de desenvolvimento propriamente dita.

Para passar no teste, a equipe de desenvolvimento precisa desenvolver e refatorar o código, sendo que refatorar aqui significa alterar algum código sem afetar seu comportamento:

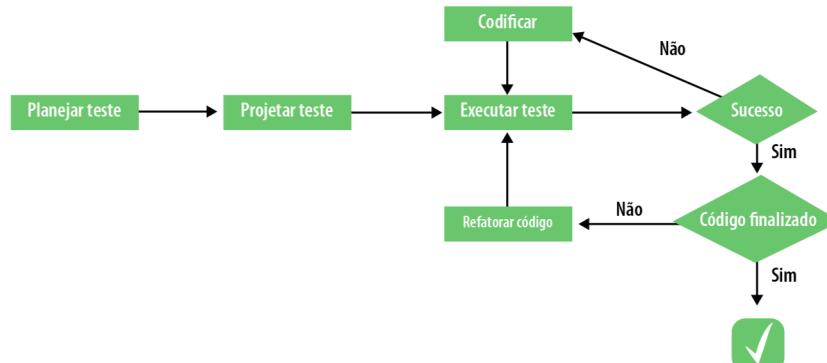


Figura 8 – Fluxo do desenvolvimento com teste no Processo TDD

A  
Z

**Planejar teste:** o planejamento dos testes permite a identificação dos itens e funcionalidades que deverão ser testados, quem são os responsáveis e quais os riscos envolvidos, ou seja, permite definir o escopo, o custo e o prazo para as atividades de teste do projeto. Para a realização desta etapa, os responsáveis devem tomar como referência os requisitos e suas prioridades. Ainda na etapa de planejamento serão definidas estimativas, estratégias e técnicas de teste.

**Projetar teste:** com base nos requisitos que descrevem a funcionalidade a ser criada e no plano de teste, o testador deve projetar o teste imaginando tanto o fluxo normal de execução, quanto os fluxos de exceção. Para a realização de tal atividade, o testador deve escrever o caso de teste e o roteiro de testes automáticos, detalhando as instruções contidas no projeto. Finalizado os documentos, deve-se submeter o teste à execução.

**Executar teste:** a execução do teste deve ser preferencialmente automática. Caso seja a primeira execução desse teste, necessariamente, ele precisa falhar, vez que foi escrito antes da funcionalidade a ser implementada.

**Codificar:** é a etapa na qual a funcionalidade será implementada pelo(s) programador(es). A equipe de desenvolvimento deve estar focada nos requisitos da funcionalidade, e apenas o código necessário para o teste ser executado com sucesso deve ser produzido.

**Refatorar código:** nesse ponto o código produzido deve passar por uma “limpeza”, com o objetivo de otimizar o código de maneira que seu comportamento não seja alterado (p. ex.: eliminando duplicações de código). Uma vez realizada a refatoração do código, os testes automáticos devem ser executados para garantir que esta não inseriu erros no código da funcionalidade.

# Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

## ▶ Vídeos

### **Modelo Cascata**

SOUZA, A. D. Modelo Cascata. 2016.

[https://youtu.be/ZMHcpn\\_MYFA](https://youtu.be/ZMHcpn_MYFA)

### **Aprenda Scrum em 9 minutos**

VIEIRA, D. Aprenda Scrum em 9 minutos.

<https://youtu.be/XfvQWnRgxG0>

### **O que é TDD?**

DEVMEDIA. O que é TDD? 2017.

<https://youtu.be/U-s0ldYia7w>

### **Extreme Programming (XP)**

CANAL TI Extreme Programming (XP). 2017.

<https://youtu.be/ALvpFMcl-dl>

## Referências

COVENTRY, T. **Gerenciamento de requisitos** – Planejamento para o sucesso: técnicas para acertar ao planejar requisitos. 2015.

GLOBAL CONGRESS 2015 – EMEA. PMI®. **Requirements management – Planning for success!**, Londres, Inglaterra. Newtown Square, PA: Project Management Institute. Disponível em: <<https://www.pmi.org/learning/library/requirements-management-planning-for-success-9669>>. Acesso em: 30 jul. 2019.

PANKAJ, P. P. **Engenharia de Software** – Modelo Evolutivo. 2019. Disponível em: <<https://www.geeksforgeeks.org/software-engineering-evolutionary-model/>>. Acesso em: 18 ago. 2019.

PETERS, J. F.; PEDRYCZ, W. **Engenharia de Software** – Teoria e Prática. Rio de Janeiro: Campus, 2001.

PFLEEGER, S. L. **Engenharia de Software** – Teoria e Prática. 2.ed. São Paulo: Pearson. 2004.

ROUSE, M. **Modelo de prototipagem**. 2018. Disponível em: <<https://searchcio.techtarget.com/definition/Prototyping-Model>>. Acesso em: 15 ago. 2019.

SEBoK – **Guide to the System Engineering Body of Knowledge**. 2011. Disponível em: <[https://www.sebokwiki.org/wiki/Guide\\_to\\_the\\_Systems\\_Engineering\\_Body\\_of\\_Knowledge\\_\(SEBoK\)](https://www.sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK))>. Acesso em: 19 jul. 2019.

SOMMERVILLE, I. **Engenharia de Software**. 9.ed. São Paulo: Pearson Prentice Hall, 2011. p. 26; 65-9.

SWERSKY, D. **O SDLC**: 7 fases, modelos populares, benefícios e muito mais. 2018. Disponível em: <<https://raygun.com/blog/software-development-life-cycle/>>. Acesso em: 12 ago. 2019.

WAZLAWICK, R. S. **Análise e Projeto Orientado a Objetos para Sistemas de Informação** – Modelando com UML, OCL e IFML. Amsterdã: Elsevier, 2014. p. 29-57.



**Cruzeiro do Sul**  
Educacional