



Engenharia de Requisitos e Processos de *Software*



Cruzeiro do Sul Virtual
Educação a distância

Material Teórico



Fundamentos de Engenharia de Requisitos

Responsável pelo Conteúdo:

Prof. Me. Artur Marques

Revisão Textual:

Prof. Me. Luciano Vieira Francisco

UNIDADE

Fundamentos de Engenharia de Requisitos



- Engenharia de Requisitos;
- Engenharia de Requisitos de *Software*;
- Requisitos do Sistema;
- Análise de Problemas – Requisitos;
- Classificação de Requisitos;
- Partes e Questionamentos em uma Especificação de Requisitos de *Software*.



OBJETIVO DE APRENDIZADO

- Conhecer os conceitos da engenharia de requisitos para a construção de *software*.



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.

Engenharia de Requisitos

A engenharia de requisitos preocupa-se em entender um sistema – *software* ou não – que o cliente tenha solicitado. Fornece a base na qual o *design* e desenvolvimento de um produto ou *software* pode prosseguir. É importante ressaltar que se os criadores/desenvolvedores não entenderem adequadamente os requisitos, é provável que o resultado não atenda às necessidades do cliente. Isso torna a compreensão dos requisitos do cliente importantes ao sucesso do projeto de desenvolvimento.

Como exemplo, você pode pensar em um veículo que apresente uma série de problemas de *design*, mecânica, consumo e conforto, e que não atende às necessidades de seus donos, tendo fracassado exatamente porque falhou na compreensão do que estes clientes, em especial, queriam. Por outro lado, quando pretendemos fabricar um *software*, é de uma importância decisiva entender o que o cliente necessita para produzir um artefato de *software* capaz de proporcionar o tipo de serviço adequado ao que foi solicitado, levando em consideração outros fatores, já que o **tempo é curto**, normalmente por praxe de mercado, e **os valores, altos**. Portanto, *software* bom é *software* com requisitos bem feitos e, principalmente, bem compreendidos.

Portanto, a engenharia de requisitos **concentra-se em descobrir o que deve ser desenvolvido – e não como deve ser desenvolvido**.

Engenharia de Requisitos de Software

A engenharia de requisitos é o **processo de adequação dos projetos** a um conjunto de requisitos básicos de *software*. Isto é extremamente importante para criar resultados precisos em engenharia de *software*.

Consiste em um conjunto de atividades cujo propósito é identificar e comunicar a finalidade de um sistema de *software* e os contextos em que serão utilizados. Portanto, atua como ponte entre as necessidades reais de usuários, clientes e outras partes interessadas e afetadas por um sistema de *software*, e as capacidades e oportunidades oferecidas por tecnologias intensivas exploradas através dos softwares.

Na engenharia de requisitos de *software* é analisado um conjunto de dados referentes às metas e aos objetivos do *software*: como funcionará? Quais são as qualidades e propriedades que deve ter para fornecer os resultados necessários? Trabalhamos a partir destas questões e destes dados para analisar diferentes soluções de construção que suportam e propiciam esses resultados.

Quando mencionamos metas e objetivos, temos por trás de tudo isso o que chamamos de **propósito do software!**

Há, dentro da indústria de requisitos, isto que chamamos atualmente de **complexidade de propósito**. O design de sistemas intensivos de *software* pertence a uma classe de problemas conhecidos como **heréticos** – em referência à dificuldade de se definir bem a qual domínio pertence o problema em questão: atualmente, há muitas áreas “cinzentas” em *software*, e outras tantas estão contidas nas pessoas e formas de clientes ou de usuários. Quando dizemos “herético”, referimo-nos às seguintes situações:

- Não há formulação definitiva do problema, porque diferentes partes interessadas têm, cada uma, a sua própria concepção do problema;
- Não há nenhuma regra de parada, porque cada solução é suscetível de levar a novas percepções sobre o problema, e o problema nunca é suscetível de ser resolvido inteiramente. Por isso, após o término de um projeto de *software*, há tantas reuniões, tantos pedidos de partes interessadas, solicitando saber o que será feito no futuro para complementar a versão entregue do *software*. Por isso também tanto **feature road map** ou, se preferir, **mapas de entrega de funcionalidades**, de tempos em tempos, permitindo ao usuário saber, com antecedência, o que será entregue, para saber se agora será atendido;
- As soluções não são certas ou erradas, mas simplesmente melhores ou piores;
- Não há nenhum teste objetivo para determinar quão boa é uma solução, de modo que o teste envolve tão somente o julgamento subjetivo das várias partes envolvidas e interessadas;
- Não há nenhum conjunto pré-existente de soluções potenciais, nem um conjunto bem descrito das propriedades de tais soluções. Devem ser descobertas durante a análise de problemas, que nos levará à descoberta dos requisitos;
- Cada problema é suficientemente complexo, de forma que nenhum outro problema será idêntico a esse. Além disso, cada problema pode ser considerado o **sintoma** de um outro problema, o que significa que é difícil isolar o problema e também escolher um nível apropriado de abstração para descrevê-lo;
- Problemas complexos tendem a nos induzir a erros, o que muitas vezes têm profundas implicações políticas, éticas e/ou profissionais: as partes interessadas tendem a ser **intolerantes** em relação a erros; e isso a despeito de todos os esforços despendidos pela área de Recursos Humanos (RH) das empresas, e também das escolas de Administração e Empreendedorismo, que dizem ser melhor errar **logo**, pois assim aprendemos e podemos fazer as coisas melhores;
- O excesso de trabalho, quando um engenheiro de requisitos precisa chegar a uma declaração acordada do problema, é o **próprio problema**. Esse tipo de situação não apenas é mal definido por falta de experiência, mas também porque tende a desafiar a própria descrição.

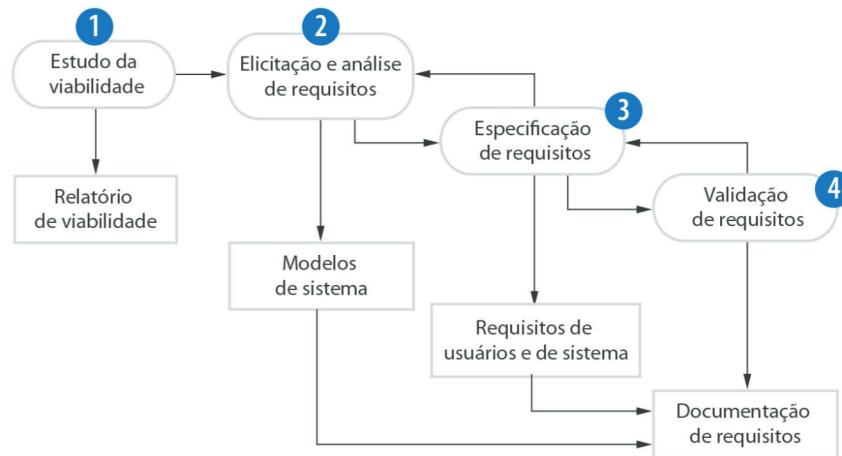


Figura 1

Fonte: Adaptado de SOMMERVILLE, 2011

Visão geral do processo da engenharia de requisitos e suas quatro fases:

i) **Estudo de viabilidade**: é feita uma estimativa acerca da possibilidade de se satisfazerem as necessidades do usuário identificado e usando as tecnologias atuais de *hardware* e *software*. Verifica se o sistema será rentável e se pode ser desenvolvido considerando as restrições orçamentárias; ii) **Elicitação e análise de requisitos**: observação e análise dos sistemas existentes, discussão com os potenciais usuários e compradores, análise das tarefas etc. Pode compreender o desenvolvimento de um protótipo; iii) **Especificação dos requisitos**: traduzir as informações obtidas na fase de análise em um documento que defina um conjunto de requisitos; iv) **A validação dos requisitos**: verifica os requisitos quanto ao realismo, à consistência e completude. Podem ser descobertos erros e o documento deve ser modificado para a correção desses problemas. A análise de requisitos é contínua, pois novos requisitos podem emergir durante o processo.

Requisitos do Sistema

Conforme o **SEBoK – System Engineering Body of Knowledge, Corpo de Conhecimentos em Engenharia de Sistemas** (2011 p. 1) –, os requisitos do sistema são todos os requisitos, no nível do sistema, que descrevem as funções que deve, como um todo, cumprir para satisfazer às necessidades e aos requisitos das partes interessadas. É expresso por uma combinação apropriada de declarações textuais, visualizações e requisitos não funcionais. Este último expressa os níveis de segurança, confiabilidade etc., que serão necessários.

Os requisitos do sistema são importantes porque formam a **base da arquitetura do sistema**. Atividades referentes à integração e verificação são referências para a validação e aceitação das partes interessadas, e são, por fim, um meio de comunicação entre os técnicos que precisam interagir durante todo o ciclo de vida do desenvolvimento do sistema.

Como você pôde notar, após a leitura e análise do relatório de necessidades – resultante de um processo termo de abertura de um projeto e de descrições iniciais –, a primeira parte importante é a engenharia de requisitos do *software*. Define-se por duas atividades, ilustradas por duas questões: primeiramente, quais soluções serão adotadas? Em segundo lugar, quais especificações serão escritas para que uma solução, que possa ser comprovada a partir de testes, seja comprehensível, permitindo a manutenção e alongando o seu ciclo de vida com qualidade?

Análise de Problemas – Requisitos

A engenharia de requisitos oferece uma gama de técnicas para lidar com a **complexidade** dos problemas, tais como:

- **Abstração** envolve ignorar detalhes para que se possa ter uma visão geral e abrangente. Por exemplo, faz-se abstração sempre que se toma algum conjunto de atividades realizadas por seres humanos, descrevendo-o em termos de **sistema**;
- **Decomposição** envolve “quebrar” um conjunto de fenômenos em partes menores para que possamos analisá-las separadamente umas das outras. Tais processos de decomposição nunca são perfeitos devido ao **acoplamento** entre as partes, mas uma decomposição – o mesmo que dizer **análise** – bem-feita ainda nos oferece **insights** sobre como as coisas funcionam;
- **Projeção** significa adotar uma visão ou perspectiva específica que pertença a outrem, tentando imaginar e descrever – “projetar” – tudo aquilo que seria relevante para este ou aquele indivíduo, este ou aquele grupo de pessoas. Ao contrário do que ocorre na decomposição, as perspectivas **não** se destinam a serem – tratadas como – independentes, mas naturalmente atreladas umas às outras.

Assim, a primeira coisa à qual nos referimos aqui é a palavra **problema** que, no âmbito da engenharia de requisitos, é sinalizado pela própria palavra: **requisito**.

Requisitos só existem porque há alguém que quer alguma coisa – um produto, serviço, *software* –; esse alguém é um cliente e, portanto, em alguns casos/projetos, os requisitos são entendidos e descritos como uma lista de recursos ou funções, propriedades, restrições e tantas outras coisas exigidas pelo cliente.

Na prática, raramente há um único cliente, mas antes um **conjunto** diversificado de pessoas que serão afetadas, de alguma forma, pelo sistema. De modo geral, para a simplificação em projetos de *software*, entende-se por **cliente** a pessoa/área ou empresa que deseja o *software* e que pagará pelo qual; o usuário, por sua vez, é aquele que efetivamente **utilizará o software** desenvolvido a partir da lista de requisitos entregue pela engenharia de requisitos. Portanto, **quem o utilizará sabe muito bem do que precisa** – mas não menospreze ou subestime as expectativas e interesses de quem colocará a mão no bolso!

Todas essas pessoas e grupos de pessoas podem ter objetivos e interesses os mais variados e, muitas vezes, conflitantes. Podem não ser explícitos, ou ainda

difícies de articular e comunicar objetivamente. Podem **não saber o que querem**. Nestas circunstâncias, simplesmente perguntar **o que necessitam** não costuma ser muito frutífero, infelizmente.



Um requisito de *software* é uma descrição dos principais recursos de um produto de *software*, o seu fluxo de informações, comportamento e atributos. Em suma, um requisito de *software* fornece uma estrutura básica para o desenvolvimento de um produto de *software*. O grau de compreensibilidade, precisão e rigor da descrição fornecida por um documento de requisitos tende a ser diretamente proporcional ao grau de qualidade do produto final (PETERS; WITOLD, 2001, p. 102).

Os principais componentes – e os seus respectivos fatores – de uma análise de requisitos, com vista à excelência do resultado final, são os seguintes:

- **Funcional:** serve para identificar as atividades do sistema, as ações;
- **Comportamental:** demonstra a sequência e se há sobreposição de funções do sistema em uma hierarquia de atividades de controle; tal sequenciamento serve para perceber e controlar as funções do sistema em diversos patamares – são os controles;
- **Não comportamental:** toda a parte de engenharia de pessoas e garantia de qualidade; ações e atributos que influenciam na excelência do resultado final.

O que se espera, com tudo isto, é um **termo**, isto é, uma especificação de requisitos, uma descrição do sistema e um plano de garantia da qualidade, bem como dos atributos que precisam ser seguidos pelo time de desenvolvimento.



Importante!

A ISO 2007 define um requisito como uma declaração que identifica um produto ou processa uma característica ou restrição operacional, funcional ou de projeto, que é inequívoca, testável ou mensurável, e necessária para a aceitabilidade do produto ou processo.

Para que possamos identificar todas essas variáveis, há uma sequência de etapas bem conhecida da engenharia de requisitos:

1. **Concepção:** aqui são definidos o **escopo** e a **natureza** do sistema;
2. **Elicitação:** começa-se a **reunir** e **organizar** os requisitos para o *software*;
3. **Elaboração:** refinar os requisitos que foram reunidos e minimamente organizados;
4. **Negociação:** são determinadas as prioridades de cada, anotados os requisitos essenciais e – o que é mais importante – solucionados os conflitos entre os diferentes requisitos;
5. **Especificação:** agora, os requisitos são reunidos em um único produto, sendo o resultado da engenharia de requisitos;

6. Validação: estabelece-se a **qualidade** dos requisitos: se são inequívocos, consistentes, completos etc.;

7. Gerenciamento: por fim, são gerenciadas as **mudanças** que os requisitos devem sofrer ao longo do tempo de vida do projeto.

A análise dos requisitos ou dos problemas, como já nos referimos, serve para identificar o ambiente, os itens produzidos, as principais funções executadas pelas pessoas e os equipamentos utilizados para a produção de um produto, os seus métodos e o seu cronograma operacional.

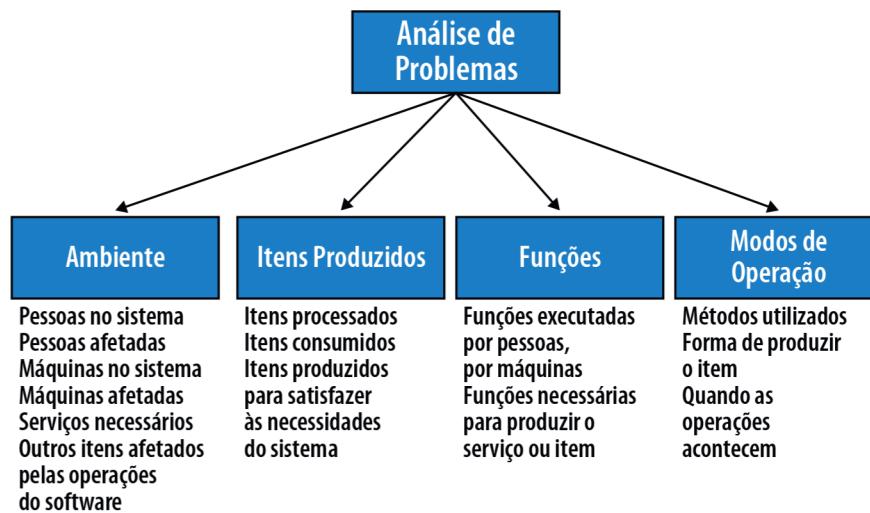


Figura 2 – Componentes que comumente são levados em consideração ao se fazer uma análise de problemas/requisitos

Classificação de Requisitos

Elaborar e definir **requisitos** é uma tarefa complexa que inclui uma série de processos, tais como **elicitação, análise, especificação, validação e gerenciamento**. É necessário saber como são classificados antes de conhecer os principais tipos de requisitos para produtos de software:

- **Requisitos de negócios:** incluem declarações de alto nível de objetivos e necessidades;
- **Requisitos das partes interessadas:** são as necessidades de determinados grupos que formam as partes interessadas discretas. É necessário especificá-los para definir o que cada parte interessada espera em termos de **solução e resultado final**;
- **Requisitos da solução:** as propriedades que um produto deve apresentar para atender da melhor maneira possível às necessidades das partes interessadas e do próprio negócio como um todo:

- » **Requisitos não funcionais:** são as características gerais de um sistema. São também conhecidos como **atributos de qualidade**;
- » Requisitos funcionais descrevem como um produto deve se comportar, os seus **recursos** e as suas **funções**. Ou seja, tudo aquilo que o faz funcionar como se espera que opere. Qualquer requisito funcional **não cumprido** impactará severamente o funcionamento, a ponto de fazer com que, no limite, simplesmente **não funcione**.
- **Requisitos de transição:** é um conjunto **adicional** de requisitos empregados em momentos de **transição** entre operações ou sistemas, definindo o que é necessário, da parte de uma organização, para conduzir-se com **êxito** de seu estado atual para o estado desejado a partir do novo produto/software.

Essa classificação de requisitos é a mais comumente utilizada, e você, como profissional de TI – Tecnologia da Informação –, conseguirá identificá-los com relativa facilidade. Todavia, o SEBoK, que é o acervo mais atualizado de engenharia de software, além de ser a fonte mais confiável, tornou-se um **padrão** em nossa indústria. Portanto, oferecemos para estudo a seguinte Tabela:

Tabela 1 – Exemplo de classificação de requisitos do sistema

Tipos de requisitos do sistema	Descrição
Requisitos funcionais	Descrevem qualitativamente as funções ou tarefas do sistema a serem executadas em operação.
Requisitos de desempenho	Definem quantitativamente a extensão, ou quão bem e sob quais condições uma função ou tarefa deve ser executada – por exemplo, taxas, velocidades etc. São requisitos quantitativos de desempenho do sistema e são individualmente verificáveis. Observe que pode haver mais de um requisito de desempenho associado a uma única função, requisito funcional ou tarefa.
Requisitos de interface	Definem de que modo o sistema se faz necessário para interagir ou trocar material, energia ou informações, com sistemas externos – <i>interface externa</i> –, ou como os elementos do sistema dentro do sistema, incluindo elementos humanos, interagem uns com os outros – <i>interface interna</i> . Os requisitos de <i>interface</i> incluem conexões físicas – <i>interfaces físicas</i> – com sistemas externos ou elementos internos do sistema que suportam interações ou trocas.
Requisitos operacionais	Definem as condições operacionais ou propriedades necessárias para o sistema operar ou existir. Esse tipo de requisito inclui fatores humanos, ergonomia, disponibilidade, facilidade de manutenção, confiabilidade e segurança.
Requisitos de modos e/ou estados	Definem os vários modos operacionais do sistema e os eventos que conduzem a transições de modos.
Restrições físicas	Definem as restrições de peso, volume e dimensão aplicáveis aos elementos do sistema.
Restrições de design	Definem os limites das opções disponíveis para o projetista de uma solução, impondo limites imutáveis: por exemplo, o sistema deve incorporar um legado ou elemento do sistema fornecido, ou determinados dados devem ser mantidos em um repositório <i>on-line</i> ?

Tipos de requisitos do sistema	Descrição
Condições ambientais	Definem as condições ambientais a serem encontradas pelo sistema em seus diferentes modos operacionais. Isto deve abordar o ambiente natural – por exemplo, o vento, a chuva, temperatura, fauna, o sal, a poeira, radiação etc. –, efeitos ambientais induzidos ou autoinduzidos – por exemplo, o movimento, choque, ruído, eletromagnetismo, térmico etc. –, e ameaças ao ambiente social – por exemplo, as legais, políticas, econômicas, sociais, comerciais etc.
Requisitos logísticos	Definem as condições logísticas necessárias para a utilização contínua do sistema. Esses requisitos incluem a manutenção – o fornecimento de instalações, suporte de nível, pessoal de suporte, as peças sobressalentes, o treinamento, a documentação técnica etc. –, a embalagem, o manuseio e transporte.
Políticas e regulamentos	Definem políticas organizacionais relevantes ou aplicáveis, ou requisitos regulatórios que possam afetar a operação ou o desempenho do sistema – por exemplo, políticas trabalhistas, relatórios para agência regulatória, critérios de saúde ou segurança etc.
Restrições de custo e cronograma	Definem, por exemplo, o custo de um único exemplar do sistema, a data de entrega esperada do primeiro exemplar etc.

Como vimos, os **requisitos de solução** são importantes, pois dizem respeito diretamente ao produto de *software* – sendo importante aprofundar-se neste ponto.

Requisitos Funcionais

Os requisitos funcionais são as **operações** desejadas e esperadas de um programa ou sistema, conforme definido no desenvolvimento de *software* e na engenharia de sistemas. Descrevem a função **final** desejada de um sistema operando conforme os parâmetros normais, de modo a assegurar que o projeto seja adequado para gerar o produto almejado, assim como o produto final atinja o seu potencial de projeto para atender às expectativas do usuário.

Comumente, um requisito funcional é uma funcionalidade básica ou um comportamento desejado, documentado de forma clara e quantitativa. Os requisitos funcionais, no que se refere a softwares, são complementados por requisitos de **qualidade** e requisitos **técnicos**, denominados **requisitos não funcionais**.

Segundo Waslawick (2014), os requisitos funcionais podem ser identificados como:

- **Evidentes:** funções executadas com o conhecimento do usuário. Esses requisitos geralmente correspondem à troca de informações entre o usuário e sistema, tais como consultas e entradas de dados que fluem pela interface do sistema;
- **Ocultos:** funções desempenhadas pelo sistema **sem** o conhecimento explícito do usuário. Geralmente estas funções são operações matemáticas e atualizações de dados, realizadas ocultamente pelo sistema, sem conhecimento explícito do usuário – mas como consequência de outras funções desempenhadas pelo usuário. Requisitos funcionais ocultos são executados **internamente** pelo sistema. Assim, embora não apareçam explicitamente como casos de uso, devem estar adequadamente associados aos quais para serem lembrados no momento do

projeto e da implementação. Neste caso, o usuário não solicita explicitamente que o sistema execute determinada operação. Como se trata de atividade executada automaticamente pelo sistema, é por princípio um requisito **oculto**.



Importante!

Quando um produto falha em requisitos funcionais, geralmente significa que o produto é de baixa qualidade e possivelmente inútil.

Alguns dos requisitos funcionais mais típicos incluem – mas não se limitam a:

- Regras do negócio;
- Correções de transação, ajustes e cancelamentos;
- Funções administrativas;
- Autenticações – usuário e senha;
- Níveis de autorização – alçada;
- Acompanhamento de auditoria – logs;
- Interfaces externas;
- Requisitos de certificação;
- Requisitos de relatório;
- Data histórica;
- Requisitos legais ou regulamentares.

Os requisitos funcionais geralmente são descritos **em texto**, ainda que se trate de um projeto tradicional ou ágil. No entanto, podem ser também visuais, apresentados no formato de:

- **Documento de especificação de requisitos de software:** contém descrições de funções e recursos que o produto deve fornecer. O documento também define restrições e suposições. Inclusive pode ser um único documento que comunica requisitos funcionais, ou pode acompanhar outras documentações de software, tais como histórias de usuários e casos de uso, nesse caso em metodologia ágil;
- **Casos de uso:** trata-se de um artefato comportamental da *UML – Unified Modeling Language* – que descreve a interação entre o sistema e os usuários, levando à obtenção de metas/objetivos específicos;
- **Histórias de usuários:** é uma descrição documentada de um recurso de **software visto da perspectiva do usuário final**. Portanto, deve descrever o que exatamente o usuário deseja do sistema para que este o atenda como esperado. Em projetos ágeis, as histórias de usuários são organizadas em um *backlog* – coisas para fazer: uma lista ordenada de funções do produto. Atualmente, as histórias de usuários são consideradas o melhor formato para itens de *backlog*;

- **Estrutura de decomposição do trabalho, ou EAP** – Estrutura Analítica do Projeto –, ou seja, uma decomposição **funcional**. É um documento visual que ilustra de que maneira os processos complexos se dividem em suas componentes mais simples. EAP é uma abordagem eficaz e que permite uma análise independente de cada parte, além de ajudar a capturar a imagem completa do projeto;
- **Protótipos:** em se tratando de *software*, podemos entender este quesito de forma abrangente, para diferentes tipos de entregas em estágio inicial, criadas para mostrar como os requisitos devem ser implementados. Os protótipos ajudam a dirimir as lacunas de visão e permitem que as partes interessadas e equipes clarifiquem áreas complicadas de produtos em vias de desenvolvimento. Tradicionalmente, os protótipos apresentam como a solução funcionará, fornecendo exemplos de como os usuários interagirão em suas tarefas. Comumente os clientes pedem que o protótipo seja submetido à grande quantidade de usuários para que possam ver como a camada de apresentação e distribuição de elementos aparecem na tela do computador. Com isso busca-se melhorar a **usabilidade**, ordenação de campos e/ou arquitetura da informação;
- **Modelos e diagramas:** quaisquer outros artefatos gráficos que possam ser úteis para a visualização da proposta de solução, desde que definidos como padrão e que sejam aceitos tanto interna como externamente – fábricas de *software*, de teste de *software*, escritórios de gerenciamento de projetos e todos os demais.

Como exemplo de requisito funcional entre produto/*software*, especificamente para uma embalagem, pode ser a capacidade de conter fluido sem vazamento.

Ademais, podem ser requisitos funcionais para um *software* os seguintes aspectos:

- O acesso ao sistema deverá ser realizado mediante identificador de usuário e senha pessoal e intransferível;
- O usuário deve ser capaz de pesquisar qualquer conjunto inicial de bancos de dados, ou selecionar um subconjunto dentro desse;
- O sistema deve fornecer visualizadores apropriados para que o usuário leia os documentos na área de armazenamento do documento;
- A cada pedido deve ser alocado um identificador único que o usuário deve ser capaz de copiar para a área de armazenamento permanente da conta.

Requisitos Não Funcionais

Os requisitos não funcionais descrevem como um sistema deve se comportar e estabelece restrições para a sua funcionalidade – **requisito funcional**. Esse tipo de requisito também é conhecido como **atributo de qualidade do sistema**. É essencial para garantir a usabilidade e eficácia de todo sistema de *software*. Falhar em atender aos requisitos não funcionais pode resultar em sistemas que não satisfazem às necessidades do usuário. Ademais, permite **impõr restrições** no sistema.

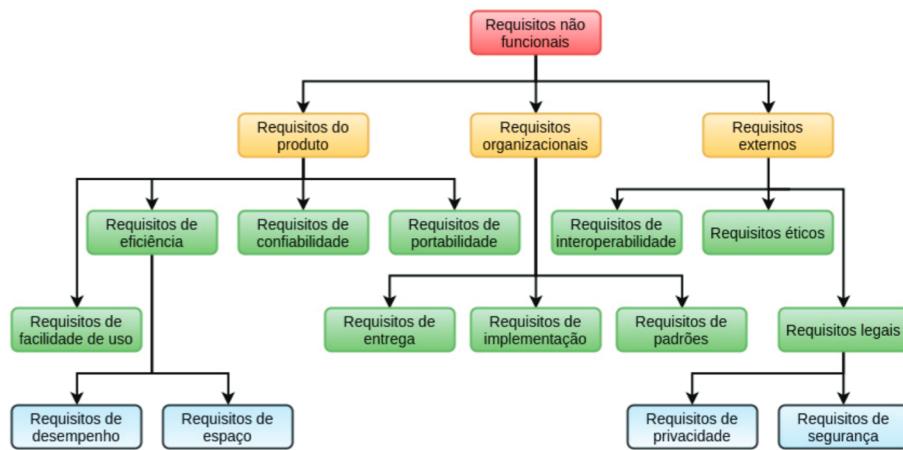


Figura 3 – Tipos de requisitos não funcionais

Fonte: Wikimedia Commons/github.com

De acordo com Antonio Mendes da Silva Filho (2008, p. 8-12), para requisitos não funcionais associados a produtos há os seguintes elementos:

Usabilidade: é um dos atributos de qualidade ou requisitos não funcionais de qualquer sistema interativo, ou seja, no qual ocorre interação entre o sistema e seres humanos. A noção de usabilidade vem do fato de que qualquer sistema projetado para ser utilizado pelas pessoas deveria ser fácil de aprender e fácil de usar, tornando, assim, fácil e agradável a realização de qualquer tarefa. Requisitos de usabilidade especificam tanto o nível de desempenho quanto a satisfação do usuário no uso do sistema;

Manutenibilidade: é geralmente empregado quando nos referimos às modificações feitas após o sistema de *software* ter sido disponibilizado para uso. Na realidade, o termo **manutenibilidade** é um tanto abrangente já que ele envolve tanto a atividade de reparo (de algum defeito existente no sistema de *software*), quanto a atividade de alteração/evolução de características existentes ou adição de novas funcionalidades não previstas ou capturadas no projeto inicial;

Confiabilidade: é a probabilidade de o *software* não causar uma falha num sistema durante um determinado período de tempo sob condições especificadas. A probabilidade é uma função da existência de defeitos no *software*. Assim, os estímulos recebidos por um sistema determinam a existência ou não de algum defeito. Em outras palavras, a confiabilidade de *software*, geralmente definida em termos de comportamento estatístico, é a probabilidade de que o *software* irá operar como desejado num intervalo de tempo conhecido. Também, a confiabilidade caracteriza-se um atributo de qualidade de *software* que implica que um sistema executará suas funções como esperado;

Disponibilidade: é uma medida de quão disponível o sistema estaria para uso, isto é, quão disponível o sistema estaria para efetuar um serviço solicitado por algum usuário. Por exemplo, um serviço de um sistema de *software* terá uma disponibilidade de 999/1.000. Isto significa que dentre um conjunto

de 1.000 solicitações de serviço, 999 deverão ser atendidas. Esta métrica é muito importante em sistemas de telecomunicações, por exemplo.

- **Taxa de ocorrência de falha:** é uma medida da frequência na qual o sistema falha em prover um serviço como esperado pelo usuário, ou seja, a frequência na qual um comportamento inesperado é provável de ser observado. Por exemplo, se temos uma taxa de ocorrência de falha de 2/1.000, isto significa que 2 falhas são prováveis de acontecerem para cada 1.000 unidades de tempo;
- **Probabilidade de falha durante fase operacional:** é uma medida da probabilidade que o sistema irá comportar-se de maneira inesperada quando em operação. Esta métrica é de suma importância em sistemas críticos que requerem uma operação contínua;
- **Tempo médio até a ocorrência de falha, ou Mean Time to Failure (MTTF):** é uma medida do tempo entre falhas observadas. Note que esta métrica oferece um indicativo de quanto tempo o sistema permanecerá operacional antes que uma falha aconteça.

Desempenho: é um atributo de qualidade importante para sistemas de *software*. Considere, por exemplo, um sistema de uma administradora de cartões de crédito. Em tal sistema, um projetista ou engenheiro de *software* poderia considerar os requisitos de desempenho para obter uma resposta de tempo para autorização de compras por cartão. Note que os requisitos de desempenho têm impacto mais global sobre o sistema e, por essa razão, estão entre os requisitos não funcionais mais importantes.

- **Requisitos de resposta:** especificam o tempo de resposta de um sistema de *software* aceitável para usuários. Neste caso, um projetista poderia especificar que o sistema deveria responder à solicitação de um serviço específico de um usuário dentro de um intervalo de 2 segundos;
- **Requisitos de processamento (throughput):** estes requisitos especificam a quantidade de dados que deveria ser processada num determinado período de tempo. Um exemplo seria exigir que o sistema de *software* possa processar, no mínimo, 6 transações por segundo;
- **Requisitos de temporização:** este tipo de requisito especifica quão rapidamente o sistema deveria coletar dados de entrada de sensores antes que outras leituras de dados de entrada, feitas posteriormente, sobrescrevam os dados anteriores;
- **Requisitos de espaço:** em alguns casos, os requisitos de espaço podem ser considerados. Aqui, podemos nos referir à memória principal ou secundária. Por exemplo, a memória principal para executar uma aplicação poderia ser considerada como um requisito de desempenho uma vez que ela está relacionada ao comportamento do sistema em tempo de execução.

Portabilidade: pode ser definida como a facilidade na qual o *software* pode ser transferido de um sistema computacional ou ambiente para outro. Em outras palavras, o *software* é dito portável se ele pode ser executado em ambientes distintos. Note que o termo ambiente pode referir-se tanto

à plataforma de *hardware* quanto a um ambiente de *software* como, por exemplo, um sistema operacional específico;

Reusabilidade: uma característica das engenharias é fazer uso de projetos existentes a fim de reutilizar componentes já desenvolvidos, objetivando minimizar o esforço em novos projetos. Dessa forma, componentes que já tenham sido desenvolvidos e testados podem ser reutilizados. Considere os elevados níveis de reusabilidade que encontramos tanto na indústria de automóveis quanto de aparelhos eletrônicos. Na indústria de automóveis, por exemplo, um motor é geralmente reutilizado de um modelo de carro para outro;

Segurança: em um sistema de *software*, este requisito não funcional caracteriza a segurança de que acessos não autorizados ao sistema e dados associados não serão permitidos. Portanto, é assegurada a integridade do sistema quanto a ataques intencionais ou acidentes. Dessa forma, a segurança é vista como a probabilidade de que a ameaça de algum tipo será repelida.

A seguir, exemplos de requisitos não funcionais genéricos:

- Os usuários devem alterar a senha de *login* original imediatamente após o primeiro *login*. Além disso, a senha inicial nunca deve ser reutilizada;
- Os funcionários não podem atualizar as próprias informações de salário. Tal tentativa deve ser relatada ao administrador de segurança;
- Toda tentativa malsucedida de um usuário para acessar um item de dados deve ser registrada em uma trilha de auditoria;
- Um *site* deve ter capacidade suficiente para lidar com 20 milhões de usuários sem que isso afete o seu desempenho;
- O *software* deve ser portável. Assim, mudar de um sistema operacional para outro não criará nenhum problema;
- A privacidade da informação, a exportação de tecnologias restritas, os direitos de propriedade intelectual etc., tudo isto deve ser auditado.

Vejamos exemplos de requisitos não funcionais de produto (SOUZA, 2017):

Requisitos de produtos:

Tabela 2 – Requisitos de usabilidade

RNF01	O sistema deverá ser operável por usuários sem a necessidade de treinamento prévio.
RNF02	O sistema deverá fazer uso de <i>design</i> responsivo na implementação de suas <i>interfaces</i> gráficas, de modo a se comportar adequadamente em navegadores acessados via computador, <i>smartphone</i> e <i>tablet</i> .
RNF03	O sistema deverá ser disponibilizado em português.

Tabela 3 – Requisitos de confiabilidade

RNF04	Apenas usuários-administradores devem possuir acesso à camada de administração do sistema.
RNF05	A senha do usuário será gravada/trafegada utilizando-se o algoritmo PBKDF2 com uma hash SHA256.
RNF06	As únicas informações que o sistema poderá exibir, no tocante a cartões de crédito salvos previamente na conta do usuário, são os últimos 4 dígitos, a data de validade e operadora.

Tabela 4 – Requisitos de portabilidade

RNF07	O sistema deverá funcionar nos navegadores <i>Google Chrome</i> , <i>Mozilla Firefox</i> e <i>Safari</i> .
-------	--

Requisitos organizacionais:

Tabela 5 – Requisitos de entrega

RNF08	O desenvolvimento do sistema deverá ser feito de maneira incremental com os PI seguindo a fundamentação do SAFe.
RNF09	Os entregáveis relacionados à arquitetura do sistema e ao nível gerencial de equipe deverão ser elaborados de acordo com as datas estimadas no <i>Kanban</i> do projeto.
RNF10	A priorização das <i>features</i> para cada entrega de incremento do produto deverá ser feita com a estimativa e os critérios de aceitação recomendados pelo SAFe.

Tabela 6 – Requisitos de implementação

RNF11	O sistema será desenvolvido utilizando a linguagem <i>Python</i> sob o framework <i>Django</i> .
RNF12	Todas as variáveis de entrada terão “vazio”, ou equivalentes, como valor <i>default</i> .

Tabela 7 – Requisitos padrões

RNF13	O sistema será desenvolvido utilizando o paradigma de programação orientado a objetos.
-------	--

Requisitos externos:

Tabela 8 – Requisitos éticos

RNF14	O sistema deverá se comunicar com o banco de dados <i>SQLite 3</i> .
RNF15	O sistema não apresentará aos usuários quaisquer dados de caráter privativo, tais como informações pessoais de outros usuários.

Tabela 9 – Requisitos legais

RNF16	O sistema deverá atender às normas legais no que se refere à comercialização <i>on-line</i> de produtos.
-------	--

Partes e Questionamentos em uma Especificação de Requisitos de Software

Quando escrevemos uma especificação de requisitos de *software*, comumente lidamos com cinco questões básicas, listadas na Figura a seguir:

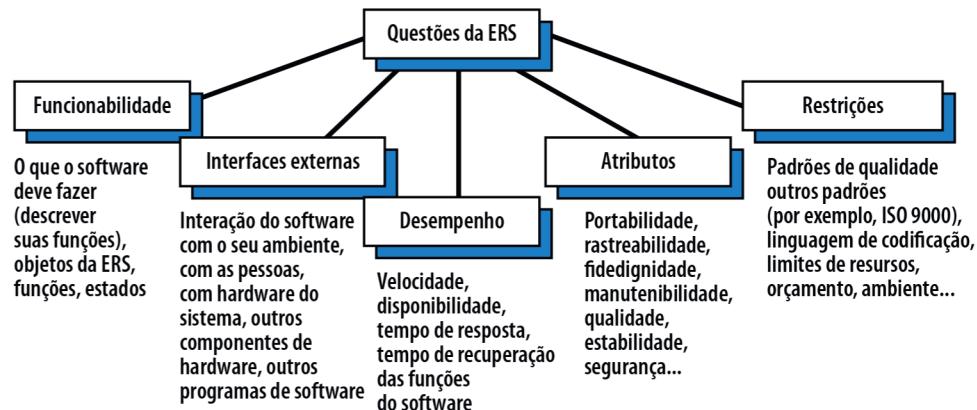


Figura 4 – Questões básicas consideradas ao se escrever uma especificação de requisitos

É importante termos em mente que os requisitos geram um documento que deve ser mantido e atualizado, pois, independentemente do paradigma de engenharia que utilizemos – como, por exemplo, cascata, espiral, que são tradicionais, ou ágeis, tais como *Scrum*, *XP*, *FDD* ou *TDD* –, devemos gerar a documentação e administrá-la, uma vez que os requisitos sofrem mudanças, atualizações, versionamentos, acréscimos e exclusões. A seguir você verá as partes de uma especificação através de seu índice – esta estrutura em questão foi adaptada do DoD USA – Departamento de Defesa dos Estados Unidos da América:



Figura 5 – Partes de uma especificação de requisitos de *software*

Especificar os requisitos é tão fundamental para o sucesso de um projeto de software que é feito um acompanhamento dos maiores problemas relatados em termos de **insucesso**. Eis o que relaciona Pfleeger (2004, p. 112):

- Requisitos incompletos com 13,1%;
- Falta de envolvimento por parte do usuário com 12,4%;
- Falta de recursos com 10,6%;
- Expectativas não realistas com 9,9%;
- Falta de apoio dos executivos com 9,3%;
- Modificações nos requisitos e nas especificações com 8,7%;
- Falta de planejamento com 8,1%;
- O sistema não era mais necessário com 7,5%.

Neste momento, é importante refletir sobre esses dados e trabalhar com as possibilidades **mais destruidoras** de uma boa especificação de requisitos, que exigirá de você muito trabalho e treino para superá-las.

- Falta de conhecimento do negócio pelo analista;
- Uso inadequado do pouco tempo que possui para fazer a especificação;
- Falta de método e organização da informação coletada;
- Subestimar a inteligência e o conhecimento do usuário e cliente;
- Inexperiência associada à prepotência;
- Falta de intimidade com ferramentas de apoio à especificação de requisitos.

Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

▶ Vídeos

O que é requisito

<https://youtu.be/oo06hyLuFNU>

Requisitos funcionais e não funcionais

<https://youtu.be/hsAtfdfk5N4>

📄 Leitura

A (suma) importância da engenharia de requisitos

<http://bit.ly/38j396l>

Processo de engenharia de requisitos aplicado a requisitos não funcionais de desempenho – um estudo de caso

<http://bit.ly/2Pseclc>

Referências

DA SILVA FILHO, A. M. **Engenharia de software 3** – requisitos não funcionais. 2008. Disponível em: <<https://www.devmedia.com.br/artigo-engenharia-de-software-3-requisitos-nao-funcionais/9525>>. Acesso em: 19 jul. 2019.

ISO-SEI 2007. *Requirements management process area and requirements development process area. In: Capability Maturity Model Integrated (CMMI) for development*, version 1.2. Pittsburgh, PA, USA: SEI/CMU, 2007.

PETERS, J. F.; WITOLD, P. **Engenharia de software**. 3. ed. São Paulo: Campus, 2001.

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**. 8. ed. [S.l.: s.n.], 2011.

SOUZA, R. F. **Requisitos não funcionais**. Github, 2017. Disponível em: <https://github.com/Desenho-2-2017/Ecom_merci/wiki/Requisitos-n%C3%A3o-Funcionais>. Acesso em: 19 jul. 2019.

WAZLAWICK, R. S. **Análise e projeto orientado a objetos para sistemas de informação – modelando com UML, OCL e IFML**. New York: Elsevier, 2014.



Cruzeiro do Sul
Educacional



Engenharia de Requisitos e Processos de *Software*



Cruzeiro do Sul Virtual
Educação a distância

Engenharia de Requisitos e Processos de *Software*



Cruzeiro do Sul Virtual
Educação a distância

Material Teórico



Ferramentas e Exemplos de Engenharia de Requisitos

Responsável pelo Conteúdo:

Prof. Me. Artur Marques

Revisão Textual:

Prof.^a Dr.^a Selma Aparecida Cesarin

UNIDADE

Ferramentas e Exemplos de Engenharia de Requisitos



- Requisitos Ágeis – Histórias dos Usuários;
- *Planning Game*;
- Principais Ferramentas para Gestão de Requisitos;
- Método Tradicional – Cascata.



OBJETIVO DE APRENDIZADO

- Familiarizar o aluno com as práticas, as ferramentas e as representações de artefatos gráficos para requisitos de sistemas.



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.

Requisitos Ágeis – Histórias dos Usuários

Os requisitos ágeis, ou seja, as histórias dos usuários nas metodologias ágeis em geral e, notadamente, no processo XP, é a unidade fundamental das atividades dos times de desenvolvimento.

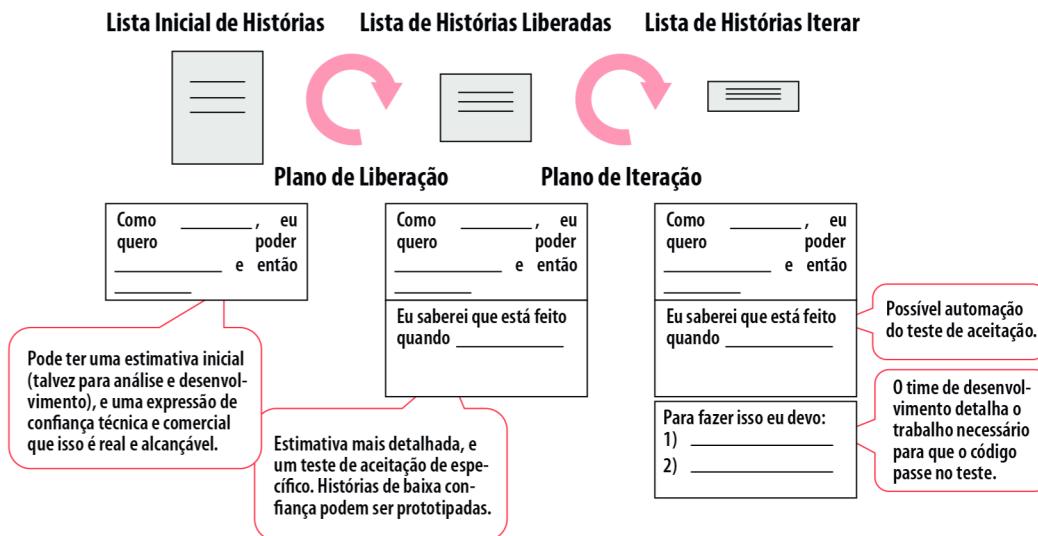


Figura 1 – Histórias e seu fluxo pelo XP, por Naresh Jain – Agile FAQs

As histórias de usuários são pequenas, muito menores do que outros artefatos de requisitos, como casos de uso ou cenários de uso. É importante reconhecer que todas as declarações feitas num *Index Card* (Cartão de Índice ou Cartão de Histórias do Usuário) representam uma única história de usuário.

Mas, como escrevemos histórias de usuários?

- Os estudantes podem comprar passes de estacionamento mensais *on-line*;
- Os passes de estacionamento podem ser pagos por meio de cartões de crédito;
- Os passes de estacionamento podem ser pagos via *PayPal*;
- Professores podem inserir as notas dos alunos;
- Os estudantes podem obter o cronograma atual do seminário;
- Os alunos podem solicitar transcrições oficiais;
- Os alunos só podem se inscrever em seminários para os quais tenham pré-requisitos;
- Transcrições estarão disponíveis *on-line* por meio de um navegador padrão.

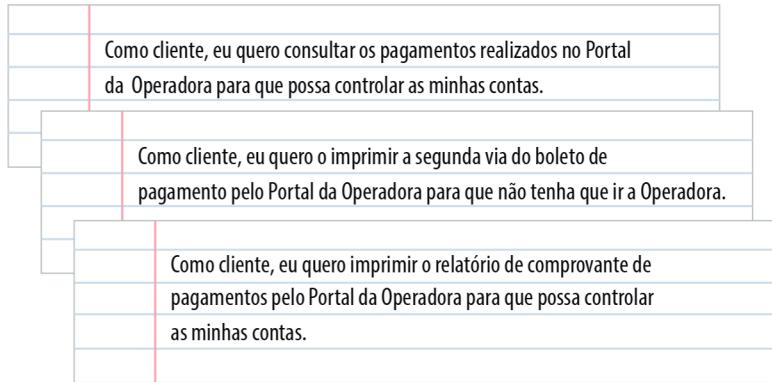


Figura 2 – Exemplo de História Inicial do Usuário formal

Há situações importantes para as partes interessadas ou a área de negócios escreverem histórias de usuários:

- **“As partes interessadas escrevem histórias de usuários:** Um conceito importante é que os envolvidos no projeto escrevem as histórias do usuário, não os desenvolvedores. As histórias de usuários são simples o suficiente para que as pessoas aprendam a escrevê-las em poucos minutos, por isso faz sentido que os especialistas do domínio (as partes interessadas) as escrevam;
- **Use a ferramenta mais simples:** As histórias de usuários geralmente são escritas em fichas de índice. Os cartões de índice são muito fáceis de trabalhar e, portanto, são uma técnica de modelagem inclusiva;
- **Lembre-se dos requisitos não funcionais:** As histórias podem ser usadas para descrever uma ampla variedade de tipos de requisitos;
- **Indique o tamanho estimado:** Inclua uma estimativa para o esforço de implementar a história do usuário. Uma maneira de estimar é atribuir pontos de história do usuário (*planning poker*) a cada cartão, uma indicação relativa de quanto esforço levará um par de programadores para implementar a história. A equipe sabe então que, se atualmente, leva em média 2,5 horas por ponto da carta; portanto, a história do usuário fica muito mais fácil de estimar. Então $2,5h \times$ uma carta com 5 pontos teremos uma atividade cuja duração seria de aproximadamente 12,5h;
- **Indique a prioridade:** Os requisitos, incluindo os defeitos identificados como parte de suas atividades de testes paralelos independentes ou por suas operações e esforços de suporte, são priorizados pelos envolvidos no projeto e adicionado à pilha no local apropriado. Você pode facilmente manter uma pilha de requisitos priorizados movendo as cartas na pilha conforme apropriado. O cartão de história do usuário inclui uma indicação da prioridade. Podemos usar uma escala de um a dez, sendo 1 a prioridade mais alta e 10 a mais baixa;
- **Inclua um identificador exclusivo.** O cartão também deverá possuir um identificador exclusivo para a história do usuário. A única razão para isso é manter a rastreabilidade entre a história do usuário e outros artefatos, em particular testes de aceitação” (AMBLER, 2009, p. 3).

Podemos, ainda, ter histórias de usuários muito grandes que chamamos de epopeias e grupos de histórias que chamamos de temas.

Vejamos como são estruturadas:

- **Epopeias:** são grandes histórias de usuário, normalmente, aquelas que são grandes demais para serem implementadas em uma única iteração e, portanto, precisam ser desagregadas em histórias de usuário menores em algum momento. Normalmente, têm prioridade mais baixa. Não faz sentido desagregar um épico de baixa prioridade porque você estaria investindo tempo em algo que você nunca poderá abordar, a menos que uma parte do épico tenha alta prioridade e precise ser descartada;
- **Tema:** é uma coleção de histórias de usuários relacionadas. Por exemplo, para um sistema de registro universitário, pode haver temas em torno dos alunos, gerenciamento de cursos, geração de transcrição, administração de notas, processamento financeiro. São usados para organizar histórias em lançamentos ou organizá-las para que várias subequipes possam trabalhar nelas.

As histórias de usuários é o alfa e o ômega dos requisitos ágeis, e os desenvolvedores costumam escrever testes baseados nelas.

As histórias são noções básicas perfeitas para testes, porque são breves e caracterizam os recursos mais importantes do produto final. Os testes são geralmente escritos antes de começar a criação do código do produto. Essa abordagem do desenvolvimento visa a economizar tempo e a atender aos termos do projeto.

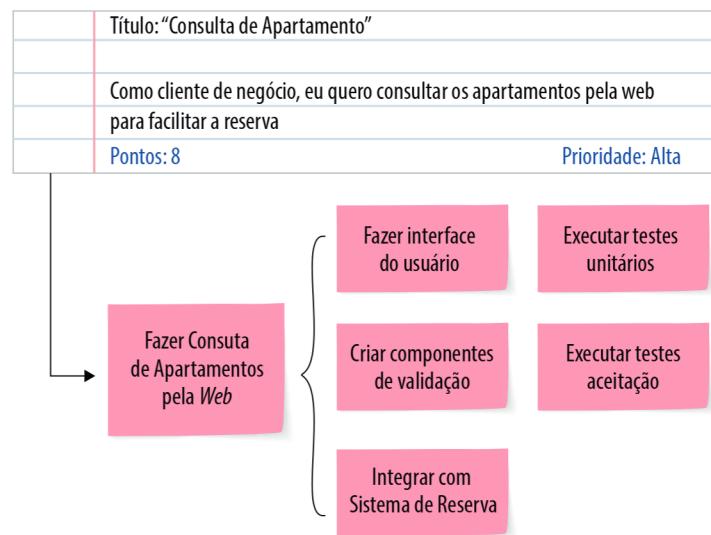


Figura 3 – Exemplo de Cartão de Índice (cartão de história de usuário)

As Histórias do Usuário são descrições de funcionalidades que têm valor para o cliente, não é uma lista de tarefas. O ato de dividir uma história em tarefas ajuda a torná-la compreensiva aos desenvolvedores.

No exemplo ilustrado na Figura 3, o cliente precisa que os usuários que navegam pelo seu site possam consultar e reservar os apartamentos. Para desenvolver esta funcionalidade, a equipe de desenvolvimento decidiu dividir a história nas seguintes

tarefas: ‘Fazer interface do usuário’, ‘Criar Componentes de validação’ e ‘Integrar com Sistema de Reserva’. E para garantir que estas tarefas irão atender a descrição da estória, criaram testes unitários e de aceitação, baseados no que o *Product Owner* escreveu no verso do cartão. A história tem valor agregado para o cliente porque representa a funcionalidade em questão. Por outro lado, as tarefas representam apenas uma parte do todo e não tem o mesmo valor.

Planning Game

O Jogo de Planejamento permite-nos encontrar rapidamente um plano aproximado e depois refiná-lo à medida que o projeto avança. Poderíamos dizer que o Jogo do Planejamento é uma reunião, é um ponto vital de interação entre cliente e desenvolvedor.

A reunião acontece com a equipe trabalhando em uma pilha de cartões de índice que contêm as histórias do usuário. Os requisitos do usuário são escritos em fichas de índice e são tratados durante o Jogo de Planejamento.

Cartões de índice são uma ferramenta altamente eficaz. A utilidade simples dos cartões conecta clientes e programadores para atingir seu objetivo comum. O uso de cartões de índice não é obrigatório no Jogo de Planejamento, e você pode descobrir que outras ferramentas, como aplicativos da Web, podem ser eficazes.

Quaisquer que sejam as ferramentas escolhidas, há uma clara separação de responsabilidades durante o Jogo de Planejamento.

Vejamos quem são os atores em XP:

Tabela 1 – Responsabilidades durante o jogo de planejamento

O negócio	Técnico
Definir o escopo do lançamento	Estimar quanto tempo cada história do usuário levará
Definir a ordem de entrega (quais histórias são feitas primeiro)	Comunicar os impactos técnicos da implementação de requisitos
Definir datas e horários de lançamento	Dividir as histórias do usuário em tarefas e aloque o trabalho

Fonte: Adaptado de informit.com

Você pode realizar as estimativas em cima desses cartões de histórias utilizando o *Planning Poker*, muito útil para as equipes ágeis. O planejamento acontece com frequência e é feito levando-se em consideração a expectativa de mudança. Mesmo com o nosso plano aproximado no jogo, temos uma imagem mais precisa do que a maioria, porque o cliente e a equipe de desenvolvimento trabalharam juntos para criá-lo.

O planejamento aborda duas questões-chave no desenvolvimento de software: prever o que será realizado até a data de vencimento e determinar o que fazer a seguir.

A ênfase está em direcionar o projeto. Duas etapas trabalham com esses conceitos, conforme Lacey:

- **O Planejamento de Liberação** é uma prática em que o Cliente apresenta os recursos desejados aos programadores e os programadores estimam sua dificuldade. Com as estimativas de custos em mãos e com o conhecimento da importância dos recursos, o Cliente estabelece um plano para o projeto. Os planos de lançamento inicial são necessariamente imprecisos: nem as prioridades nem as estimativas são verdadeiramente sólidas e, até que a equipe comece a trabalhar, não saberemos com que rapidez elas serão. Até mesmo o primeiro plano de lançamento é preciso o suficiente para a tomada de decisões, no entanto, e as equipes revisam o plano de lançamento regularmente.
- **Planejamento de Iteração** é a prática pela qual a equipe recebe orientação a cada duas semanas. As equipes de XP criam softwares em “iterações” de duas semanas, entregando softwares úteis ao final de cada iteração. Durante o planejamento de iteração, o cliente apresenta os recursos desejados para as próximas duas semanas. Os programadores dividem-nas em tarefas e estimam seu custo (em um nível mais refinado de detalhes do que no Planejamento da Liberação). Com base na quantidade de trabalho realizado na iteração anterior, a equipe se inscreve para o que será realizado na iteração atual. (LACEY, 2018)

O jogo de planejamento XP tem dois participantes no processo de planejamento: Negócios e Desenvolvimento. Isso pode ajudar a remover parte do calor emocional inútil da discussão. Não há como um simples conjunto de regras existir para lembrar a todos como eles gostariam de atuar, e eles fornecem uma referência comum quando as coisas não estão indo bem (BECK, 1999, p.38).

- **Desenvolvedores:** são as pessoas que irão realizar a implementação do que foi definido no cartão;
- **Negócio (cliente):** são as pessoas que vão dizer aos desenvolvedores o que eles querem que seja implementado. O cliente decide a prioridade do trabalho a ser feito e o que está dentro e fora do escopo. O cliente Agile garante a máxima satisfação do cliente do produto e o valor das organizações, garantindo quais são as histórias nas quais as equipes estão trabalhando, são priorizadas para gerar o máximo valor.

Segundo Beck, existem três fases do jogo a saber:

1.**Exploração:** Esta é uma maneira de tentar descobrir coisas novas que o sistema é capaz de fazer. O propósito da fase de exploração é dar aos jogadores uma apreciação do que todo o sistema deve eventualmente fazer. Exploração tem três movimentos:

- **Escreva uma história:** Negócios escreve uma história descrevendo algo que o sistema precisa fazer. As histórias são escritas em cartões de índice, com um nome e um parágrafo curto descrevendo o propósito da história;

- **Estimar uma história:** Desenvolvimento estima quanto tempo a história levará para implementar. Se o desenvolvimento não puder estimar a história, ela pode pedir aos negócios que esclareçam ou dividam a história;

- **Uma história é contada:** Se Desenvolvimento não puder estimar uma história completa ou se os negócios perceberem que parte de uma história é mais importante do que o restante, os negócios podem transformar uma história em duas ou mais histórias.

2. Compromisso: Será tomada a decisão quanto aos passos a serem seguidos na realização dos requisitos. O propósito da fase de compromisso é que as empresas escolham o escopo e a data da próxima versão e que o desenvolvimento se comprometa com a entrega. Possui 4 movimentos a saber:

- **Classificar por Valor:** Negócios classifica as histórias em 3 pilhas:

1. Aquelas sem as quais o sistema não funcionará;
2. Aquelas que são menos essenciais, mas fornecem valor comercial significativo;
3. Aquelas que seria bom ter.

- **Classificar por risco:** Desenvolvedores classificam as histórias 3 pilhas:

1. Aquelas que podem estimar com precisão;
2. Aquelas que podem ser razoavelmente estimadas;
3. Aquelas que não podem estimar.

- **Definir velocidade:** Desenvolvedores informam a rapidez com que a equipe pode programar em tempo de engenharia ideal por mês;

- **Escolher escopo:** A empresa escolhe o conjunto de cartões na liberação, seja definindo uma data para a engenharia estar completa e escolhendo cartões com base em sua estimativa e velocidade do projeto, ou escolhendo os cartões e calculando a data.

3. Steer (direção): Esta é a orientação dos desenvolvedores para o requisito do projeto. O propósito da fase de direção é atualização o plano com base no que é aprendido pelo desenvolvimento e pelos negócios. A fase de direção tem quatro movimentos:

1. **Iteração:** no início de cada iteração (de 1 até 3 semanas), Negócios selecionam uma iteração das histórias mais valiosas a serem implementadas. As histórias da primeira iteração devem resultar em um sistema que é executado de ponta a ponta;

2. **Recuperação:** Se o desenvolvimento perceber que ele superestimou sua velocidade, ele poderá solicitar a Negócios que é o conjunto de histórias mais valioso a ser retido no release atual com base na nova velocidade e nas estimativas;

3. **Nova história:** Se os negócios perceberem que precisa de uma nova história durante o meio do desenvolvimento de um lançamento, ele poderá escrever a história. O desenvolvimento estima a história, depois o negócio remove as histórias com a estimativa equivalente do plano restante e insere a nova história;

4. **Reestimativa:** Se o desenvolvimento achar que o plano não fornece mais um mapa preciso do que fazer, ele pode reestimar todas as histórias restantes e definir a velocidade novamente. (BECK, 1999)

Fowler (2012) afirmou que “Valores sem práticas são difíceis de aplicar e podem ser aplicados de muitas maneiras. É difícil saber por onde começar e práticas sem valores são como atividades sem propósito”.

O jogo de planejamento de maneira sumária se resume a:

- Relacionar os itens de trabalho que talvez precisem ser feitos (histórias de usuários que carregam implicitamente requisitos funcionais, não funcionais e regras de negócios aglutinadas para que formem uma *sprint* que tem propósito de entregar uma funcionalidade por si só ou que seja dependente de mais *sprints* e também quando estiverem concluídas gerarão um *release* que será liberado em produção, com entregas valiosas ao cliente);
- Estimar os itens;
- Definir um orçamento para o ciclo de planejamento;
- Concordar com o trabalho que precisa ser feito dentro do orçamento. Ao negociar, não altere as estimativas ou o orçamento.

Principais Ferramentas para Gestão de Requisitos

Muitas vezes, eliciar os requisitos de um sistema complexo e de natureza dinâmica gera desafios sérios para controlar o ciclo de vida dos requisitos. Por isso, as ferramentas nos auxiliam na organização, afinal, num sistema, muitas vezes centenas de requisitos devem ser cumpridos.

Aqui vão algumas ferramentas que vão auxiliar você nessa tarefa:

- **Helix RM:** ajuda as equipes a capturar, decompor e priorizar os requisitos, identificar o status de cada requisito dentro do processo de aprovação, realizar revisões de requisitos, manter-se atualizado com as alterações e colaborar com todas as partes interessadas. A ferramenta também permite que as empresas reutilizem os requisitos de outros projetos para reduzir o tempo de validação, o retrabalho e garantir a consistência entre os projetos. Os requisitos podem ser vinculados a outros requisitos, casos de teste e resultados ou código-fonte. O *Helix RM* oferece uma matriz de rastreabilidade para ajudar as equipes a identificar melhor as causas raiz. A análise de impacto para executar cenários hipotéticos, rastreamento de bugs e gerenciamento de tarefas pode ser concluída ao lado do Jira. Toda a colaboração com as partes interessadas pode ser feita em tempo real para que todos estejam atualizados sobre as mudanças conforme elas acontecem. Esta ferramenta não é gratuita;

» **Integrações:** O *Helix RM* integra-se ao *Atlassian Jira*, *Atlassian Bamboo*, Produtos Microsoft, Slack, Adobe, ActiveState Komodo, Apache Maven, Amazon Lumberyard, Autodesk 3ds Max, Autodesk Maya, Crytek CRYENGINE, DBmaestro TeamWork, Eclipse, ElectricFlow, GitHub, Go2Group ConnectALL, IBM Rational DOORS, IBM Aspera, JetBrains, Puppet, Prometheus, Thoughtworks e Unity.

- **Jira:** é uma das ferramentas mais reconhecidas usadas pelas equipes para gerenciamento de ciclo de vida de aplicativos (ALM) e gerenciamento de requisitos. O Jira ajuda as equipes a identificar e mapear os requisitos de negócios, colaborar com as partes interessadas, garantir que todas as tarefas se conectem diretamente a qualquer requisito de captura e fornecer às partes interessadas entregas de alta qualidade. O Jira facilita a visualização de como os requisitos de negócios e os problemas existentes estão vinculados, rastreia as tarefas do projeto relacionadas aos requisitos de negócios e determina como os requisitos diferem de especificações padrão. O Jira também permite que as partes interessadas criem e visualizem uma estrutura de hierarquia de requisitos. O Jira não é uma ferramenta gratuita;
- » **Integrações:** O Jira possui uma extensa lista de integrações, incluindo gerenciamento de projetos, ferramentas administrativas, utilitários, blueprints, gráficos e diagramas, CRM, painéis, codificação, e-mail, gerenciamento de documentos, suporte técnico, segurança, testes e muito mais. O site da *Atlassian* oferece uma lista completa das atuais integrações do Jira.
- **Orcanos:** A ferramenta de *Application Lifecycle Management (ALM)* e gerenciamento de requisitos (RM) da Orcanos é uma solução flexível e poderosa que fornece um único repositório para gerenciamento de requisitos em empresas de pequeno a grande porte. É compatível com dispositivos médicos e oferece rastreabilidade de ponta a ponta, análise de impacto, painéis em tempo real, alertas e notificações. A Orcanos oferece uma ferramenta de rastreabilidade de requisitos que facilita o rastreamento de cobertura e rastreabilidade de requisitos de sistema, hardware ou software, casos de teste, riscos e muito mais. As partes interessadas podem obter uma visão hierárquica de seus requisitos e colaborar através de e-mail, mensagens instantâneas, alertas ou notificações. A Orcanos também fornece um gerador de documentos *Microsoft Word* que suporta relatórios incorporados e modelos personalizados. Orcanos, não é uma ferramenta gratuita;
- » **Integrações:** Entre em contato com a Orcanos para obter uma lista de todas as integrações de aplicativos.

- **ReQtest:** A ferramenta de gerenciamento de requisitos *ReQtest* coloca o foco no gerenciamento de projetos de ponta a ponta com foco na experiência do usuário. O *ReQtest* oferece um módulo de gerenciamento de testes, rastreamento avançado de bugs e um dashboard intuitivo para rastrear e gerenciar tarefas. A ferramenta simplifica o processo de identificação e gerenciamento da interface comercial, de marketing, funcional, não funcional, de usuário ou

outros requisitos. Os requisitos de negócios são armazenados em uma estrutura semelhante a uma árvore, na qual as equipes podem ver, planejar e gerenciar tudo com mais eficiência. *ReQtest* não é uma ferramenta gratuita;

» **Integrações:** *ReQtest* integra-se ao Jira; integrações personalizáveis também são oferecidas via API.

- **Visure Requirements:** *Requirements* oferece uma ferramenta que fornece suporte essencial para o processo de requisitos de negócios de ponta a ponta. A ferramenta visa padronizar e aplicar processos claramente definidos e formalizar uma estrutura de especificação de requisitos comum. Ele gerencia as alterações durante todo o ciclo de vida do aplicativo e captura, analisa, valida, rastreia e permite a reutilização de requisitos. As partes interessadas podem criar famílias de produtos que compartilham um conjunto de requisitos, requisitos de padrões ou recursos comuns, juntamente com testes ou casos de uso. Como uma ferramenta multiusuário com recursos de controle de versão, várias partes interessadas podem fazer alterações no mesmo requisito, simultaneamente, sem nenhum problema. *Visure* não é uma ferramenta gratuita;
- » **Integrações:** integrações com ferramentas de terceiros incluem *Doors*, *HP ALM*, *RiskCAT*, *Jira* e *Enterprise Architect* (MOIRA, 2019).

Método Tradicional – Cascata

Durante o estágio de requisitos, os desenvolvedores anotam todos os requisitos possíveis de um sistema em um documento de requisitos. O documento define o que o sistema deve fazer, mas não necessariamente como ele funcionará. Os desenvolvedores basearão todo o desenvolvimento futuro do *software* no documento de requisitos.

Na próxima etapa da análise, os desenvolvedores usam o documento de requisitos para examinar e detalhar o *design* lógico ou teórico do sistema, sem levar em consideração suas tecnologias de *hardware* ou *software*.

Como sabemos, isso é muito útil, porque os desenvolvedores podem detectar erros de *design* durante os estágios de análise e *design*, o que os ajuda a evitar a geração de código com defeito durante o estágio de codificação. O projeto definiu metas claramente, para que os desenvolvedores possam trabalhar em direção a objetivos concretos e medir facilmente seu progresso.

No processo em cascata, todos os requisitos do sistema a serem desenvolvidos são capturados nessa fase de análise com base nas necessidades e nos problemas e estão em um documento chamado DRS ou Documento de Especificação de Requisitos.

Os requisitos devem ser claros e também de maneira detalhada. Esses requisitos são estabelecidos pela equipe, que inclui especialistas no assunto, usuários de negócios e analistas de negócios.

O analista entende o domínio do problema, corrige a captura e documenta os requisitos no documento de especificação de requisitos. O analista usa o método de perguntas e respostas para esclarecer as dúvidas e também para a atualização do documento de requisitos.

Os usuários e os analistas se valem, ainda, de reuniões presenciais, ou chamadas de vídeo e áudio, além, é claro, de se valerem, também, de questionários, folhas de entrevista, ferramentas de modelagem, quadro branco para capturar as entradas nos requisitos.

Primeiro, após a captura, são analisados os requisitos e se escrevem as especificações, depois que um documento de requisitos é concluído, ele é revisado pelos analistas e pelos usuários, para garantir que os requisitos esperados sejam capturados, e sejam claros e compreendidos com facilidade.

Após essa revisão, o documento de requisitos é assinado para ser usado nas outras fases do modelo Cascata.

Os requisitos são considerados uma linha de base para processos de controle de alterações para gerenciar a modificação feita. Escrever boas especificações de requisitos do Sistema é essencial para o sucesso de qualquer Projeto de Software.

Geralmente, a elaboração de especificações técnicas para o *software* ocorre após uma primeira discussão entre a equipe de desenvolvimento e o proprietário do produto, isso em qualquer tipo de processo, mas no Cascata é muito necessário.

As especificações servem como referência para a estimativa de custo e tempo. Como o documento de requisitos do sistema visa a descrever fielmente o *software* a ser desenvolvido, isso torna o processo de estimativa muito mais fácil e muito mais preciso. Lembre-se de que, no processo cascata, que não permite alteração, uma vez definido o escopo, é fundamental a precisão.

O documento de especificação de requisitos DRS também é utilizado como documento de especificação funcional DFD ou documento de requisitos do produto DRP, sendo que o DER descreve todas as funcionalidades e explica como a funcionalidade estará dentro de um determinado sistema como parte de um sistema maior ou como um sistema independente.

Na sequência, são apresentadas as perguntas chave para você responder durante o desenvolvimento do *software*:

- O que o aplicativo ou *software* deve fazer? Isso ajuda a identificar as principais funcionalidades e o objetivo principal do *software*;
- Como o *software* deve se comportar? Serve para entender como o *software* interage com o ambiente em que é implantado. Ele também define a especificação de *hardware* e define como o *software* interage com os usuários finais. O *software* a ser descrito pode ser um sistema inteiro ou, às vezes, faz parte de um sistema mais extenso. É, então, essencial definir como essa parte interage com um sistema maior e como os dois sistemas se comunicam. A especificação

de requisitos do sistema de CRM (Relacionamento com o Cliente) ou ERP (Gestão e Planejamento de Recursos da Empresa) são exemplos em que é essencial entender como o *software* deve se comportar;

- Quais são os requisitos em termos de desempenho? Fornecerá, por exemplo, informações sobre o tempo de resposta aceitável, com que rapidez deve responder e com que rapidez deve lidar com os problemas quando ocorrerem. Sim, às vezes, eles são requisitos não funcionais;
- Existem requisitos ou restrições que devem ser levados em consideração ou respeitados? Ele tem como objetivo determinar as restrições a serem levadas em consideração durante o *design*, o desenvolvimento e a implantação do Sistema.

O DRS, em sua construção, varia de empresa e de abordagem, portanto não existe um padrão, mas ao menos esses componentes são úteis em sua composição:

- **Introdução:** é importante que a equipe de desenvolvimento e os proprietários do produto definam e escrevam essa parte juntos. No final, podemos incluir uma breve visão geral do documento para dar aos leitores uma ideia do que eles podem esperar do documento. Não esqueça de incluir os termos técnicos;
- **Descrição Geral:** é importante explicar as diferentes funcionalidades do aplicativo. Nesta parte, as interfaces de *hardware* e de usuário são definidas. Em que dispositivo os usuários finais esperam acessar o aplicativo;
- **Requisitos específicos:** os requisitos são detalhados para facilitar o *design* do produto e validar o *software* de acordo com os requisitos. Aqui, é importante descrever todas as entradas que o *software* manipula e todas as saídas para definir melhor a interação com outros sistemas e facilitar a integração. As funcionalidades enumeradas na seção anterior serão detalhadas aqui;
- **Referências:** é importante incluir informações sobre o conteúdo, para que seja mais fácil encontrá-las quando necessário.

Segundo Osetskyi, são consideradas más práticas num DRE (Documento de Requisitos do Sistema):

- **Dicionário incompleto:** o DER pode incluir jargões que somente pessoas familiarizadas com a empresa podem entender. Uma especificação de requisitos visa proporcionar a todos os envolvidos no desenvolvimento do *software* uma melhor compreensão do que o *software* faz, etc. é importante que todos compreendam todos os termos usados no documento.
- **Misturando conceitos:** pode ser tentador lançar todas as informações que temos no mesmo local, mas isso leva a uma documentação ruim.
- **Inclua instruções de desenvolvimento:** é importante separar os requisitos de *software* da implementação técnica. Os proprietários do produto conhecem melhor suas necessidades e a equipe de desenvolvimento sabe como desenvolver o *software* que atende a essas necessidades.

- **Ação passiva:** Por exemplo: os relatórios são gerados clicando em um determinado botão. É importante saber que esperamos a geração de relatórios a partir do *software*, mas também é importante saber quem clicará no botão para gerar o relatório.
- **Documentação ambígua e incompleta:** algumas vezes algumas linhas nos requisitos podem levar a várias interpretações. Além disso, para cada funcionalidade ou situação descrita é importante que o documento não apresente aspectos ainda não determinados. (OSETSKYI, 2018, p. 4)



Veja um exemplo claro de um documento de requisitos, acessando: DA SILVA, F. R. Documento de Requisitos do Sistema – Módulo de Avaliação Acadêmica – Versão 0.1. 2016. Disponível em: <http://bit.ly/2FyJ6Ux>

Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

▶ Vídeos

Histórias de Usuário – Fatto Consultoria e Sistemas

<https://youtu.be/sEtiCJfXTE8>

3 formas de Quebrar Histórias de Usuário – Rodrigo Vieira

<https://youtu.be/OAVR9Zc5DeI>

O que é requisito – Fatto Consultoria e Sistemas. 2015

<https://youtu.be/oo06hyLuFNU>

Gerenciamento de requisitos sem mistério

<https://youtu.be/jajQyzOpLaE>

Referências

AMBLER, S. W. **Histórias de usuários: uma introdução ágil.** 2009. Disponível em: <<http://www.agilemodeling.com/artifacts/userStory.htm>>. Acesso em: 30 ago. 2019.

BECK, K. **Extreme Programming Explained: Embracing Change.** 1st edition. Boston: Addison-Wesley, 1999.

FOWLER, M. **O Modelo de fluência Ágil.** 2012. Disponível em: <<https://martin-fowler.com/articles/agileFluency.html>>. Acesso em: 30 ago. 2019.

LACEY, M. **Planning Game 2018.** Mitch Lacey & Associates, INC. EUA. Disponível em: <<https://www.mitchlacey.com/intro-to-agile/extreme-programming/planning-game>>. Acesso em: 30 ago. 2019.

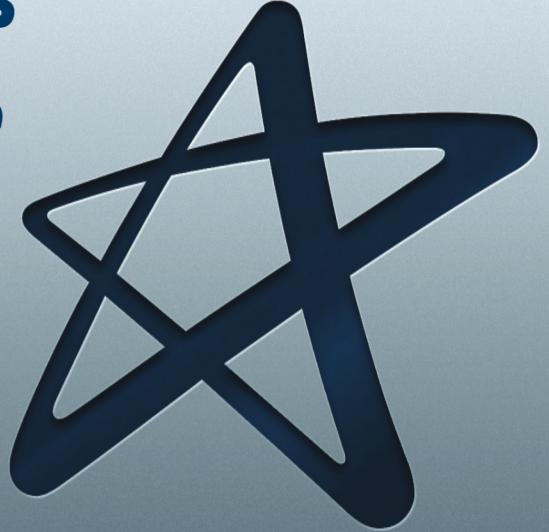
MOIRA, A. **5 principais ferramentas de gerenciamento de requisitos.** 2019. Disponível em: <<https://cio.com.br/5-principais-ferramentas-de-gerenciamento-de-requisitos/>>. Acesso em: 30 ago. 2019.

OSETSKYI, V. **Como escrever a especificação de requisitos do sistema para desenvolvimento de software.** 2018. Disponível em: <<https://dzone.com/articles/how-to-write-the-system-requirements-specification>>. Acesso em: 30 ago. 2019.



Cruzeiro do Sul
Educacional

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.^a Dr.^a 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)

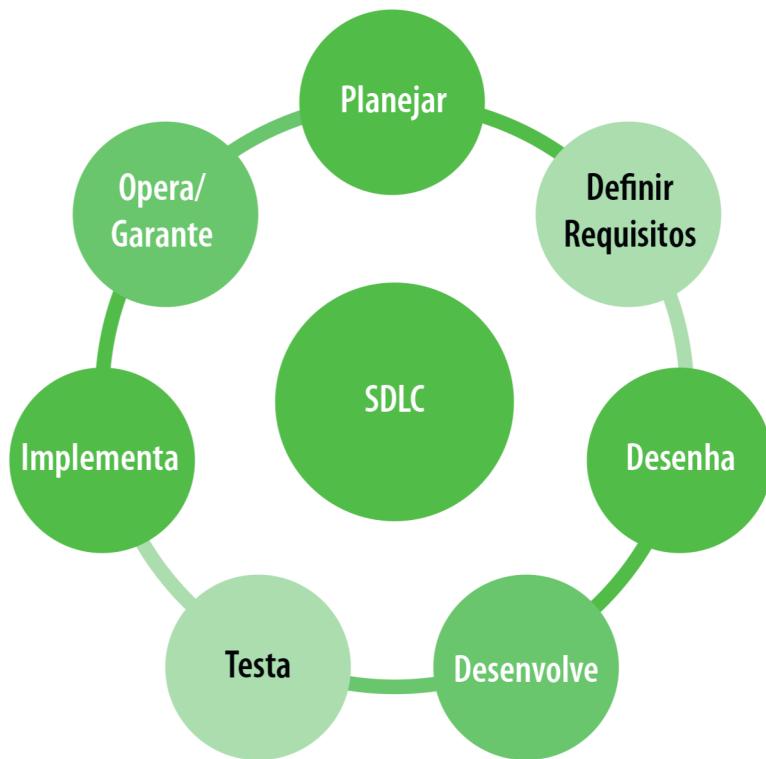


Figura 1 – Fases do SDLC. Durante a leitura de livros de Engenharia de *Software*, você poderá se deparar com SDLC com 5 fases (requisitos, desenho, desenvolvimento, testes e produção), há uma simplificação. Com o passar do tempo e o amadurecimento da Área, percebe-se que há necessidade de estarmos com 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.

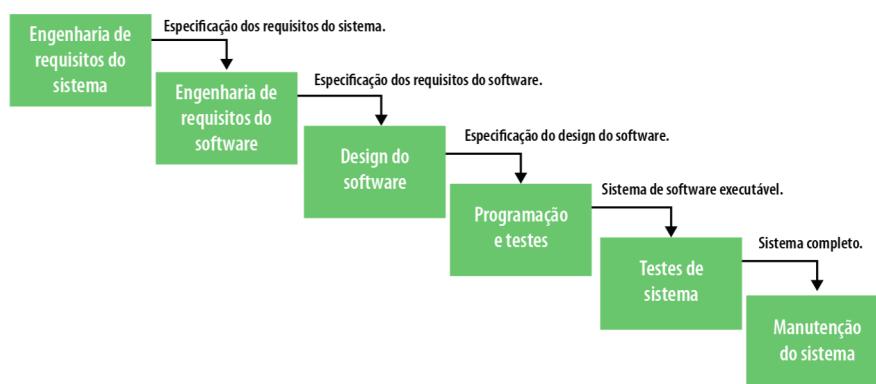


Figura 2 – Modelo em Cascata

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.

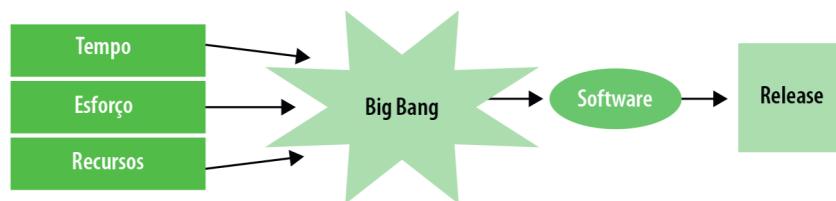


Figura 3 – Modelo de desenvolvimento de Software *Big Bang*

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:

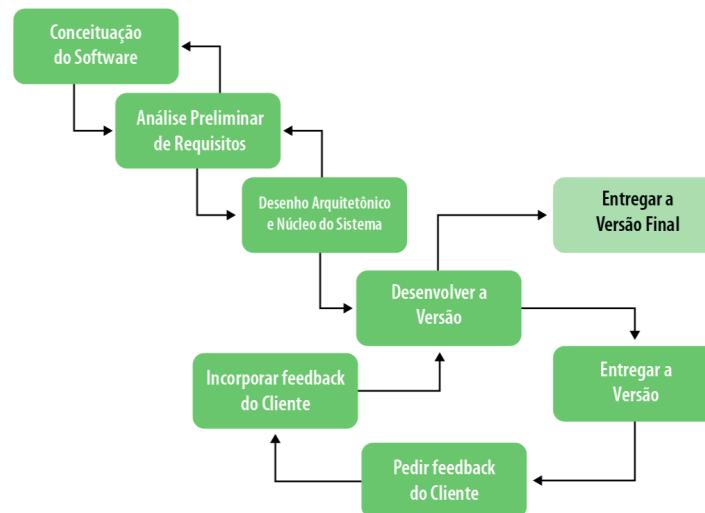


Figura 4 – Representação gráfica do modelo evolutivo

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.

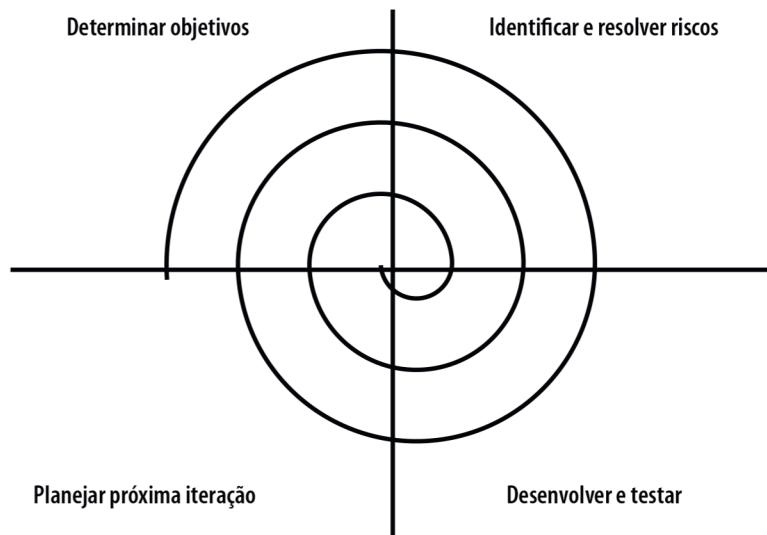


Figura 5 – Modelo de Engenharia de Software baseado na Espiral de Boehm. Em cada fase do Modelo de Espiral, as características do produto datadas e analisadas e os riscos nesse momento são identificados e resolvidos por meio de prototipagem

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)

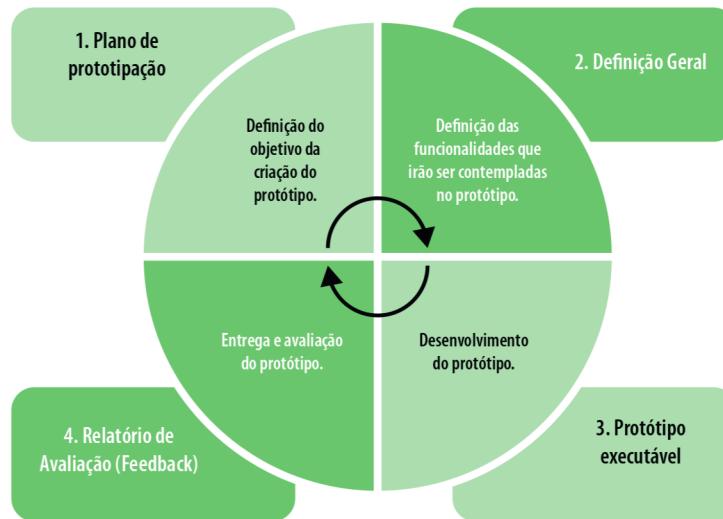


Figura 6 – Processo de Prototipagem e suas fases

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.

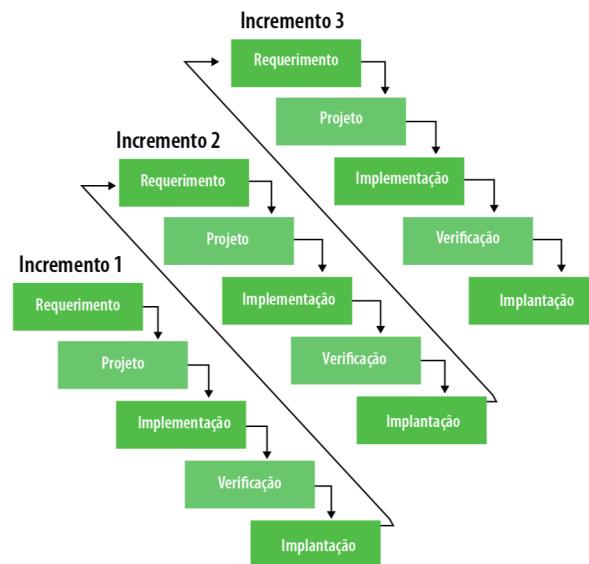


Figura 7 – Ciclo de vida do Desenvolvimento Iterativo



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:

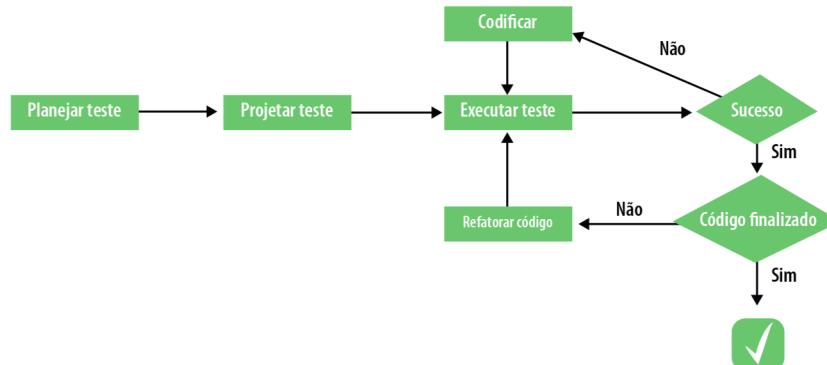


Figura 8 – Fluxo do desenvolvimento com teste no Processo TDD

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

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