Faculdade Pitágoras

FÁBIO DE FREITAS FERREIRA

ENGENHARIA DE SOFTWARE
DESENVOLVIMENTO DE SOFTWARE
BELO HORIZONTE – 06/12/2017

FÁBIO DE FREITAS FERREIRA

ENGENHARIA DE SOFTWARE

DESENVOLVIMENTO DE SOFTWARE

Trabalho de Conclusão de Curso apresentado à
(Faculdade Pitágoras), como requisito parcial
Para a obtenção do título de graduado em
(Sistema de Informação).

Orientador (a): Vanessa Silicani

Belo Horizonte – MG
2017

FÁBIO DE FREITAS FERREIRA

ENGENHARIA DE SOFTWARE
DESENVOLVIMENTO DE SOFTWARE

Trabalho de Conclusão de Curso apresentado à
(Faculdade Pitágoras), como requisito parcial
Para a obtenção do título de graduado em
(Sistema de Informação)

BANCA EXAMINADORA

_____

Prof.ª Vanessa Silicanni

_____

Prof.º

_____

Prof. (a)

Belo Horizonte, 06 de dezembro de 17

Dedico esse trabalho à senhora Tereza Fernandes de Freitas (mãe), que me motivou e ajudou financeiramente para que eu chegasse ao final do curso e aos professores que tiveram paciência e flexibilidade em ministrar todos os conteúdos referente aos anos letivos e duração do curso de Sistema de Informação. Também deixo meus agradecimentos aos colegas e amigos que estudamos juntos nesses 4 anos de graduação.

Obrigado e muita gratidão a todos os envolvidos.

Ferreira, Fábio de Freitas. **Engenharia de software:** desenvolvimento de software. 2017. 77 páginas. Trabalho de Conclusão de Curso (Graduação em Sistema de Informação) – Faculdade Pitágoras, Belo Horizonte, 2017.

Resumo

Trabalho de Conclusão de Curso (TCC), considero um dos trabalhos mais importantes da vida acadêmica em Bacharelado de Sistemas de Sistema de Informação. Baseando se no TCC I, dou continuidade a proposta base em Engenharia de Software, com subtema em Desenvolvimento de software. Nesse trabalho será abordado o que é engenharia de software, como é utilizada nas empresas, o que o profissional de engenharia de software faz em uma organização, melhores práticas, principais pontos, visões de autores diferentes e como é utilizada a engenharia de software no processo de desenvolvimento de software, englobando desde o início, colher e analisa requisitos, diagramas em geral e o desenvolvimento com suas várias etapas realizada em um ciclo de desenvolvimento de software, testes, homologação e o produto final, ou seja o que deve ser feito em um projeto de sistema computacional na organização. Esse trabalho, não somente servirá de aprimoramento do aprendizado de quem vos fala e cumprimento de exigência acadêmica de iniciação cientifica. Deixo claro que o material será teórico e não será abordado o tema: programação como códigos, interfaces e até mesmo o sistema funcional

Palavras-chaves: Engenharia 1; Desenvolvimento 2; Software 3; Projeto 4; Processo 5;

Ferreira Fábio de Freitas. **Software Engineering:** Software development. 2017.78 pages. Work of completion of course (graduation in information System) – Pythagoras College, Belo Horizonte, 2017

Abstract

Work of conclusion of course (TCC), consider one of the most important works of my academic life in Bachelor of Information System systems. Based in the TCC I, give continuation to the proposal based on Software Engineering, with subthemes in software development. In this paper I will discuss what is software engineering, as is used in companies, which make software engineering professional in an organization, best practices, key points, visions of different authors and as software engineering is used in the process of software development, including from the start, spoon and analyzes requirements, diagrams in General and the development with its various steps performed in a software development cycle, testing, approval and the final product, namely software developed tested and running ready for use in the organization. This work, not only serve as a reference for those who wish to join in the area of development, but also as a manual of good practices in Software development. Leave clear that material will be theoretical and will not be addressed the theme: codes, programming interfaces and even the functional system,

Keywords: Engineering 1; Development 2; Software 3; Project 4; Process 5;

SUMÁRIO

INTRODUÇÃO

O termo ‘engenharia de software’ ficou conhecido na conferência da OTAN, em 1968, para se discutir francamente os problemas relacionados à projetos de *softwares* complexos. Os tópicos mais importantes foram as linguagens de programação, orientação a objetos e documentação. Engenharia de *software* está totalmente ligada a esses tópicos. Porem com o crescimento e avanço das tecnologias computacionais, os engenheiros de *softwares* tiveram maior facilidade para desenvolver grandes projetos sistêmicos e grande dificuldade para acompanhar os avanços tecnológico que é mais rápida que a capacitação de alguns Engenheiros, ‘no Brasil’.

Trabalho de Conclusão de Curso, é de total importância de grade curricular e aprendizado em: engenharia de *software* para compreender o desenvolvimento de *software*, sejam eles simples, ou complexos, com foco em cumprir as exigências acadêmicas curriculares, como atividade avaliativa obrigatória.

Com objetivo em criar uma revisão bibliográfica utilizando de pesquisas em livros e sites referentes ao assunto de engenharia de *software*. Aprimorar os conhecimentos referentes à: processos, analise de requisitos, arquitetura de *software*, modelagem, qualidade de *software* e projetos.

O que é Engenharia de *Software*? Para que serve um projeto? Qual a finalidade de utilizar engenharia de *software* em um projeto? Essas e muitas outras perguntas serão respondidas ao longo de todo o desenvolvimento dessa monografia.

Nessa monografia, iremos descobrir a importância da engenharia de *software* em projetos sistêmicos, como utiliza lá, suas premissas, qual o papel do engenheiro de *software* em uma organização e como ele pode ajudar no crescimento e competitividade da organização.

Vamos entender o papel da engenharia de software nos projetos sistêmicos, arquitetura, modelagem e como colocar em pratica as orientações aqui existentes. Esse TCC tem como principal objetivo orientar quem deseja construir sistemas de qualidade e lidar com projetos de diferentes segmentos.

1 ENGENHARIA DE *SOFTWARE*

O que é Engenharia de *Software*?

Engenharia de *software* é uma área (disciplina) da computação voltada à especificação, desenvolvimento, manutenção e criação de sistemas de software, com a aplicação de tecnologias e práticas de gerênciamento de projetos e outras disciplinas visando, organização, produtividade e qualidade.

> O mundo moderno não poderia existir sem o software. Infraestruturas e serviços nacionais são controlados por sistemas computacionais, e a maioria dos produtos elétricos inclui um computador e um software que o controla. A manufatura e a distribuição industriais são totalmente informatizadas, assim como o sistema financeiro. A área de entretenimento, incluindo a indústria da música, jogos de computador, cinema e televisão, faz uso intensivo de software. Portanto, a engenharia de software é essencial para o funcionamento de sociedades nacionais e internacionais.
> (SOMMERVILLE, 2011, p.2).

> Quando um *software* é bem-sucedido – atende às necessidades dos usuários, opera perfeitamente durante um longo período, é fácil de modificar e, mais fácil ainda, de utilizar -, ele é realmente capaz de mudar as coisas para melhor. Porém, quando um software falha – quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil modifica-lo -, fatos desagradáveis podem e de fato, acontecem. Todos queremos construir softwares que facilitem o trabalho, evitando pontos negativos latentes nas tentativas mal-sucedidas. Para termos êxito, precisamos de disciplina no projeto e na construção do software. Precisamos de uma abordagem de engenharia (PRESSMAN, 2011, p.7).

> UML (*Unified Modeling Language*) é uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de software, particularmente daqueles construídos utilizando o estilo orientado a objetos (OO). Essa definição é um tanto simplificada. Na verdade, para diferentes pessoas a UML tem significados diferentes. Isso ocorre devido à sua própria história e às diferentes maneiras de ver o que compõe um processo de engenharia de software eficaz (FOWLER, 2005, p. 25)

> Os sistemas de software são abstratos e intangíveis. Eles não são restringidos pelas propriedades dos materiais, nem governados pelas leis da física ou pelos processos de manufatura. Isso simplifica a engenharia de software, porque não há limites naturais para o potencial do software. No entanto, devido a essa falta de restrições físicas, os sistemas de software podem se tornar extremamente complexos de modo muito rápido, difíceis de entender e caros para alterar (SOMMERVILLE, 2011, p. 2).

Nos anos 70, devido ao crescimento por procura de *softwares* para as tarefas complexas, a produção de *software* entrou em crise, justamente pelos problemas que se encontravam em construir *softwares* complexos. Foi nesse contesto que surgiu a Engenharia de Software com o objetivo de tratar e controlar o desenvolvimento de *softwares* complexos. Em outras palavras, a engenharia de *software* tem como objetivo principal: oferecer meios de gerenciar, planejar o processo de

desenvolvimento de *software*, dinamizando a produção, criando modelos de produção com métodos e ferramentas que possibilita o engenheiro de *software* desenvolver *softwares* com qualidade.

Engenharia de *Software*, tem como objetivo, apoiar o desenvolvimento de *software*, incluindo de especificação, projeto e evolução de programas. Engenharia de *software*, não trata especificamente do programa em si, mas de toda à documentação da arquitetura e dados de configuração para o bom funcionamento do sistema e auxilio ao usuário de como deverá utilizar o sistema.

A engenharia de *software* é responsavel por genrenciar o projeto de *software* e desenvolver as ferramentas, métodos e teorias apoiando a produção. A engenharia precisa garantir o resultado com qualidade dentro do prazo e orçamento estipulado no escopo do projeto. Engenharia tem tudo a ver com qual método será mais adequado para a circunstancia X ou Y.

Engenharia de *software* é importante por dois motivos:

1. Cada vez mais, indivíduos e sociedades dependem dos sistemas de *software* avançados. Temos de ser capazes de produzir sistemas confiáveis com econômia e rapidez.

2. Geralmente é mais barato, a longo prazo, usar métodos e técnicas da engenharia de *software* para sistemas de *software*, em vez de simplesmente escrever os programas como se fossem algum projeto pessoal. Para a maioria dos sistemas, a maior parte do custo é mudar o *software* depois que ele começa a ser usado.

Engenharia de *software* é uma abordagem sistemática para a produção de *software*; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtores do *software*

Graças a Engenharia de *Software* que se realizou a exploração do espaço, criação da Internet e as telecomunicações modernas, transporte de cargas e passageiros como monitoramento via satelit, carros inteligêntes, automatização em linha de produção e muitos outros produtos coexistentes e à de existir em um futuro próximo.

> Inúmeras pessoas escrevem programas. Pessoas envolvidas com negócios escrevem programas em planilhas para simplificar seu trabalho; cientistas e engenheiros escrevem programas para processar seus dados experimentais; e há aqueles que escrevem programas como hobby, para seu próprio interesse e diversão. No entanto, a maior parte do desenvolvimento de software é uma atividade profissional, em que o software é desenvolvido para um propósito específico de negócio, para inclusão em outros dispositivos ou como produtos de software como sistemas de informação, sistemas CAD etc.

> O software profissional, o que é usado por alguém além do seu desenvolvedor, é normalmente criado por equipes, em vez de indivíduos. Ele é mantido e alterado durante sua vida (SOMMERVILLE, 2011, p. 3).

1.1 ENGENHARIA DE SOFTWARE X ENGENHARIA DE INFORMAÇÃO X CIÊNCIA DA COMPUTAÇÃO

A Engenharia da Informação: aplica técnica estruturada na empresa como um todo, ou um departamento especifico, focando no desenvolvimento e evolução do sistema que tem o papel principal na organização, ou seja, engenheiros envolvidos em especificação sistemica, da arquitetura geral e intregração das partes dos sitema.

A Engenharia de *Software*: aplica técnica estruturada a um projeto, cuidando de todos os aspectos do sistema como *hardware* e *software* e principalmente preocupando se com as práticas de produção de *software*, ou seja, seu foco é o *software*.

> Tipos diferentes de sistemas necessitam de diferentes processos de desenvolvimento. Por exemplo, um software de tempo real em uma aeronave precisa ser completamente especificado antes de se iniciar o desenvolvimento. Em sistemas de comércio eletrônico, a especificação e o programa são, normalmente, desenvolvidos juntos. Consequentemente, essas atividades genéricas podem ser organizadas de formas diferentes e descritas em nível de detalhamento diferente, dependendo do tipo de software em desenvolvimento (SOMMERVILLE, 2011, p. 6).

Ciencia da computação: aborda as teórias e fundamentos, se preocupando como o *software* que é executado internamente em uma máquina.

É melhor fazer Ciência da computação ou Engenharia de *Software*? Qual a deferença entre os cursos e as carreiras?

Segundo Ben Soher (2013), pode se dizer que a engenharia de *software* se concentra na prática de produção de sistemas de *software*, e a ciência da computação nos duntamentos e aspectos computacionais.

Segundo Silva (2011), Curso de Engenharia da Computação abrange conhecimentos em Engenharia Eletrônica e Ciência da Computação, trabalhando a interface dos *softwares*.

> A Engenharia da Computação aproxima-se bastante do curso de Engenharia Eletrônica e dura cinco anos. A ênfase dos cursos de Ciência da Computação recai sobre a criação e desenvolvimento de softwares, baseados em modelos matemáticos, e em média duram quatro anos (Silva, 2011, p. 20).

Importância da Engenharia de *Software*

Cada vez mais, empresas, pessoas e sociedade em geral, dependem dos sistemas de *software*, seja ele avançado ou simples. Para uma organização é mais barato utilizar métodos de engenharia de *software* a longo prazo para não ter que fazer mudanças futuramente por causa do custo que pode ser muito maior, por esses motivos o engenheiro de *software* deve ser capaz de produzir sistemas confiáveis rapidamente.

> Em geral, os engenheiros de software adotam uma abordagem sistemática e organizada para seu trabalho, pois essa costuma ser a maneira mais eficiente de produzir software de alta qualidade. No entanto, engenharia tem tudo a ver com selecionar o método mais adequado para um conjunto de circunstâncias, então uma abordagem mais criativa e menos formal pode ser eficiente em algumas circunstâncias. Desenvolvimento menos formal é particularmente adequado para o desenvolvimento de sistemas Web, que requerem uma mistura de habilidades de software e de projeto (SOMMERVILLE, 2011, p. 5).

1.2 FUNDAMENTOS DA ENGENHARIA DE *SOFTWARE*

- Sistemas desenvolvidos com um processo compreendido e gerenciado.
- Confiança e desempenho são importante para todos os sistemas.
- Entender e gerenciar os requisitos e as especificações de *software*.
- Explorar os recursos existentes.

Qual o papel do Engenheiro de *Software* em uma organização?

> Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem de pessoas para tomar decisões e fazer julgamentos. Não existe um processo ideal, a maioria das organizações desenvolve os próprios processos de desenvolvimento de software. Os processos têm evoluído de maneira a tirarem melhor proveito das capacidades das pessoas em uma organização, bem como das características específicas do sistema em desenvolvimento. Para alguns sistemas, como sistemas críticos, é necessário um processo de desenvolvimento muito bem estruturado; para sistemas de negócios, com requisitos que se alteram rapidamente, provavelmente será mais eficaz um processo menos formal e mais flexível (SOMMERVILLE, 2011).

O engenheiro de *software* dedica-se exclusivamente ao desenvolvimento de *softwares* e programas computacionais, desenhando e testando novos programas, além de fazer sua manutenção e revisão. Cria aplicativos, jogos, plataformas digitais para atividades educacionais, sistemas embarcados e sistemas específicos, como: médicos e bancários, de forma a elevar seu desempenho e produtividade. O bacharel também pode trabalhar em empresas públicas ou privadas, na área de inovação em startups e em grandes companhias de tecnologia

> Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software. Essas atividades podem

> envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação como Java ou C, no entanto aplicações de negócios não são necessariamente desenvolvidas dessa forma. Atualmente, novos softwares de negócios são desenvolvidos por meio da extensão e modificação de sistemas existentes ou por meio da configuração e integração de prateleira ou componentes do sistema (SOMMERVILLE, 2011, p.18).

Engenheiros faz que o processo em desenvolvimento de *softwares* funcione como métodos, teorias e ferramentas aplicadas apropriadamente. Sem contar com o a motivação em descobrir as soluções dos problemas e sempre reconhecendo que deve se trabalhar de acordo com as restrições financeiras e regras de negócios da companhia.

> Engenharia de software é uma abordagem sistemática para a produção de software; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtores do software. A forma como essa abordagem sistemática é realmente implementada varia dramaticamente de acordo com a organização que esteja desenvolvendo o software, o tipo de software e as pessoas envolvidas no processo de desenvolvimento. Não existem técnicas e métodos universais na engenharia de software adequados a todos os sistemas e todas as empresas. Em vez disso, um conjunto diverso de métodos e ferramentas de engenharia de software tem evoluído nos últimos 50 anos (SOMMERVILLE, 2011, p. 7).

1.3 *SOFTWARE*

O que é um *software*?

Software é um produto criado pelo profissional de engenharia de *software* e ao qual se dar suporte ao logo de sua vida útil, esse produto é um programa de computador que segue uma sequência de instruções escritas e interpretadas em linguagem de máquina para executar uma tarefa especifica, *software* é a parte lógica, com função de fornecer instruções ao hardware (parte física do computador), *software* computacional é a tecnologia mais importante no cenário mundial, por ser utilizado em aplicações especificas ou em diferentes áreas e segmento que vai desde ao programa que controla sua agenda do celular até sistemas financeiros, petrolíferos, na visualização de um exame de ressonância magnética, à sistemas operacionais e controle bancário.

> Muitas pessoas pensam que software é simplesmente outra palavra para programas de computador. No entanto, quando falamos de engenharia de software, não se trata apenas do programa em si, mas de toda a documentação associada e dados de configurações necessários para fazer esse programa operar corretamente. Um sistema de software desenvolvido profissionalmente é, com frequência, mais do que apenas um programa; ele normalmente consiste em uma série de programas separados e arquivos de configuração que são usados para configurar esses programas. Isso pode incluir documentação do sistema, que descreve a sua estrutura;

> documentação do usuário, que explica como usar o sistema; e sites, para usuários baixarem a informação recente do produto (SOMMERVILLE, 2011, p. 3).

O produto *software* assume o papel de interação com a peça física (*hardware*), transmitindo as informações de forma gerencial de diversas fontes e dispositivos diversos, como celulares ou *mainframe*, os dados são adquiridos, exibidos, modificados e transmitidos como informações entre os dispositivos de forma simples (textos) ou complexas (multimídia). O *software* é usado como veículo para distribuir os dados dentro do computador através do sistema operacional ou através da rede, controlando outros programas, aplicações e ferramentas em geral, com objetivo crucial e mais importante em distribuir as informações de forma que o *software* recebe os dados e os transformam em informações úteis de fácil entendimento auxiliando as empresas se manterem competitivas no mercado e colaborando com as redes mundiais de informação (internet).

Software consiste em: programas de computadores que ao ser executado apresenta características, funções e desempenho; com estruturas de dados possibilitando manipulação das informações; descrevendo operações e uso dos programas, tanto virtual como impresso.

Existem vários tipos de sistemas de *software*, por isso é exigido diferentes abordagem. Desenvolver um sistema corporativo para uma organização qualquer, que controle todos os processos existentes na empresa, não é o mesmo que desenvolver um sistema controlador de voo, ou até mesmo o desenvolvimento de jogos computacionais. Todos esses sistemas, precisam de engenharia de *software* com técnicas diferentes.

Exemplos de tipos de produtos de *software*:

> 1. Produtos genéricos. Existem sistemas stand-alone, produzidos por uma organização de desenvolvimento e vendidos no mercado para qualquer cliente que esteja interessado em comprá-los. Exemplos desse tipo de produto incluem software para PCs, como ferramentas de banco de dados, processadores de texto, pacotes gráficos e gerenciamento de projetos. Também incluem as chamadas aplicações verticais projetadas para um propósito específico, como sistemas de informação de bibliotecas, sistemas de contabilidade ou sistemas de manutenção de registros odontológicos.
> 2. Produtos sob encomenda. Estes são os sistemas encomendados por um cliente em particular. Uma empresa de software desenvolve o software especialmente para esse cliente. Exemplos desse tipo de software são sistemas de controle de dispositivos eletrônicos, sistemas escritos para apoiar um processo de negócio específico e sistemas de controle de tráfego aéreo (SOMMERVILLE, 2011, p. 3).

1.4 CLASSIFICAÇÃO DE *SOFTWARE*

Software de Sistema: conjunto de processos do sistema interno computacional permitindo a interação dos usuários e os periféricos através da interface gráfica.

Software de Programação: ferramentas que o programador usa para criar programas, utilizando linguagem de programação em um ambiente virtual de desenvolvimento integrado.

Software de Aplicação: programas computacionais que o usuário utiliza para executar várias tarefas específicas em diversas áreas ao mesmo tempo.

> **Software científico/de engenharia** - tem sido caracterizado por algoritmos "number crunching" (para "processamento numérico pesado"). As aplicações vão da astronomia à vulcanologia, da análise de tensões na indústria automotiva à dinâmica orbital de ônibus espaciais, e da biologia molecular à fabricação automatizada. Entretanto, aplicações modernas dentro da área de engenharia/científica estão se afastando dos algoritmos numéricos convencionais. Projeto com auxílio de computador, simulação de sistemas e outras aplicações interativas começaram a ter características de sistemas em tempo real e até mesmo de software de sistemas (PRESSMAN, 2011, p. 34).
>
> Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum para um novo engenheiro de software trabalhar num programa mais velho que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por essa geração facilite o trabalho de futuros engenheiros de software. Ainda assim, novos desafios têm surgido no horizonte (PRESSMAN, 2011, p. 35).

Software Livre: programa que permite ao usuário ter acesso ao código fonte e altera-lo conforme sua necessidade.

Software Genérico: são os sistemas que pode ser adquirido por qualquer pessoa, baixando gratuitamente ou comprando.

Software Específico: são sistemas solicitado por uma pessoa específica, onde ela informa o que comportamento do software, quem pode utilizar e onde instalar.

> **Software de inteligência artificial** – faz uso de algoritmos não numéricos para solucionar problemas complexos que não são passíveis de computação ou de análise direta. Aplicações nessa área incluem: robótica, sistemas especialistas, reconhecimento de padrões (de imagem e de voz), redes neurais artificiais, prova de teoremas e jogos (PRESSMAN, 2011, p.35).

1.5 TIPOS DE SISTEMAS DE *SOFTWARE*

Tabela 1.1 Perguntas frequentes sobre *software*

Pergunta	Resposta
O que é *software***?**	Softwares são programas de computador e documentação associada. Produtos de *software* podem ser desenvolvidos para um cliente específico ou para o mercado em geral.
Quais são os atributos de um bom *software*?	Um bom *software* deve prover a funcionalidade e o desempenho requeridos pelo usuário; além disso, deve ser confiável e fácil de manter e usar.
O que é engenharia de *software*?	É uma disciplina de engenharia que se preocupa com todos os aspectos de produção de *software*.
Quais são as principais atividades da engenharia de *software*?	Especificação de *software*, desenvolvimento de *software*, validação de *software* e evolução de *software*
Qual a diferença entre engenharia de *software* e ciência da computação?	Ciência da computação foca a teoria e os fundamentos; engenharia de *software* preocupa-se com o lado prático do desenvolvimento e entrega de *softwares* úteis
Qual a diferença entre engenharia de *software* e engenharia de sistemas?	Engenharia de sistemas se preocupa com todos os aspectos do desenvolvimento de sistemas computacionais, incluindo engenharia de *hardware*, *software* e processo. Engenharia de software é uma parte específica desse processo mais genérico.
Quais são os principais desafios da engenharia de *software*?	Lidar com o aumento de diversidade, demandas pela diminuição do tempo para entrega e desenvolvimento de *software* confiável.
Quais são os custos da engenharia de *software*?	Aproximadamente 60% dos custos de *software* são de desenvolvimento; 40% são custos de testes. Para *software* customizado, os custos de evolução frequentemente superam os custos de desenvolvimento.
Quais são as melhores técnicas e métodos da engenharia de *software*?	Enquanto todos os projetos de software devem ser gerenciados e desenvolvidos profissionalmente, técnicas diferentes são adequadas para tipos de sistemas diferentes. Por exemplo, jogos devem ser sempre desenvolvidos usando uma série de protótipos, enquanto sistemas de controle críticos de segurança requerem uma especificação analisável e completa. Portanto, não se pode dizer que um método é melhor que outro.
Quais diferenças foram feitas pela Internet na engenharia de *software*?	A Internet tornou serviços de *software* disponíveis e possibilitou o desenvolvimento de sistemas altamente distribuídos baseados em serviços. O desenvolvimento de sistemas baseados em Web gerou importantes avanços nas linguagens de programação e reuso de software.

Fonte: Sommerville (2011, p. 4)

Sistemas Legados: são *softwares* muito antigos escritos em linguagem defasadas acarretando um grande custo em sua manutenção. Exemplos: sistemas bancários, petrolíferos ou de entidades públicas.

Sistemas de Tempo Real: são sistemas com tempo de respostas muito rápido e, ou seja, tempo de resposta curto. Por exemplo: jogos.

Sistemas Embarcados ou Embutidos: são *softwares* criados para funcionar fora do ambiente computacional, com processos restritos a executar somente uma função específica e são encontrados em eletrodomésticos. Por exemplo: Forno Micro-ondas, relógios digitais, ar condicionado e etc...

Sistemas Web: são aplicações que funcionam somente conectados à internet, utilizando tráfego de dados através de um servidor. Exemplo: *Whatzapp*, Uber e etc...

Sistemas Científicos: sistemas exclusivos para área cientificas, como geoprocessamento.

Sistemas de Controle embutidos: controla e gerencia dispositivos de *hardware*. Exemplo: sensor de estacionamento de um carro, o antitravamento dos freios ABS.

Sistemas de Processamento de Lotes: *software* que processa dados em grandes lotes. Processando as entradas e criando as saídas especificas de cada entrada. Exemplo: sistema de cobrança de cartão de credito.

Sistemas de entretenimento: *software* de uso pessoal que entreter o usuário. Exemplos: jogos.

Sistemas para Modelagem e Simulação: sistemas modelados e utilizado por engenheiros e cientistas para modelar processos. Exemplo: SGAD.

Sistemas de Coleta de Dados: *softwares* que coletam dados através de sensores enviando os dados para outro *software* processar. Exemplo: carro da *Google*.

Tabela 1.2- tabela de características essenciais para um sistema de *software*

Características do produto Descrição	**Descrição**
Manutenibilidade	O *software* deve ser escrito de forma que possa evoluir para atender às necessidades dos clientes. Esse é um atributo crítico, porque a mudança de software é um requisito inevitável de um ambiente de negócio em mudança.
Confiança e proteção	A confiança do *software* inclui uma série de características como confiabilidade, proteção e segurança. Um software confiável não deve causar prejuízos físicos ou econômicos no caso de falha de sistema. Usuários maliciosos não devem ser capazes de acessar ou prejudicar o sistema.
Eficiência	O *software* não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. Portanto, eficiência inclui capacidade de resposta, tempo de processamento, uso de memória etc.
Aceitabilidade	O software deve ser aceitável para o tipo de usuário para o qual foi projetado. Isso significa que deve ser compreensível, usável e compatível com outros sistemas usados por ele.

Fonte: Sommerville (2011, p. 5)

1.6 PROCESSOS DE SOFTWARE

Um processo de *software* é um conjunto de atividades relacionadas que levam à produção de um produto de *software*. Essas atividades podem envolver desenvolvimento de *software* a partir do zero ou como nos dias de hoje; os novos softwares são desenvolvidos por extensões ou modificações de *softwares* já existentes.

> Existem muitos tipos de software. Não existe um método ou uma técnica universal de engenharia de software que se aplique a todos. No entanto, há três aspectos gerais que afetam vários tipos diferentes de software:
> 1. Heterogeneidade. Cada vez mais se requer dos sistemas que operem como sistemas distribuídos através das redes que incluem diferentes tipos de computadores e dispositivos móveis. Além de executar nos computadores de propósito geral, o software talvez tenha de executar em telefones móveis. Frequentemente, você tem de integrar software novo com sistemas mais antigos, escritos em linguagens de programação diferentes. O desafio aqui é desenvolver técnicas para construir um software confiável que seja flexível o suficiente para lidar com essa heterogeneidade.
> 2. Mudança de negócio e social. Negócio e sociedade estão mudando de maneira incrivelmente rápida, à medida que as economias emergentes se desenvolvem e as novas tecnologias se tornam disponíveis. Deve ser possível alterar seu software existente e desenvolver um novo software rapidamente. Muitas técnicas tradicionais de engenharia de software consomem tempo, e a entrega de novos sistemas frequentemente é mais demorada do que o planejado. É preciso evoluir para que o tempo requerido para o software dar retorno a seus clientes seja reduzido.
> 3. Segurança e confiança. Pelo fato de o software estar presente em todos os aspectos de nossas vidas, é essencial que possamos confiar nele. Isso se torna verdade especialmente para sistemas remotos acessados através de uma página Web ou uma interface de web service. Precisamos ter certeza de que os usuários maliciosos não possam atacar nosso software e de que a proteção da informação seja mantida (SOMMERVILLE, 2011, p.6).

Especificação de *software*, Projeto e implantação de *software*, Validação de *software* e Evolução de *software*; são atividades fundamentais para a engenharia de software.

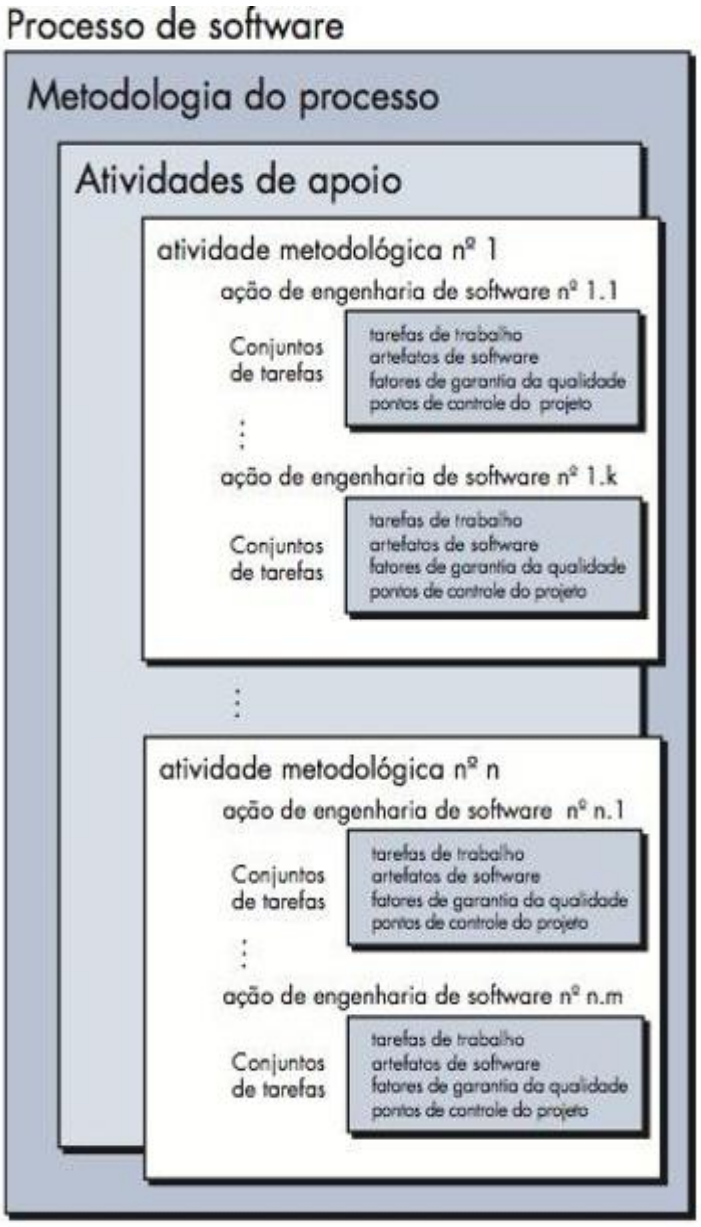
Os processos de *software* são complexos e dependem de pessoas para tomar decisões. Não existe um processo ideal, a maioria das organizações desenvolve seu próprio processo de *software*. O processo de desenvolvimento de software visa tirar o melhor proveito das capacidades das pessoas em uma organização e principalmente quando se está desenvolvendo um *software*. Os processos de *software* são categorizados como dirigidos a planos: são os que todas as atividades são planejadas com antecedência, e avaliado por comparação com o planejamento inicial, ou processos ágeis: onde o planejamento é gradativo e fácil de alterar o processo de maneira a refletir as necessidades de mudanças. Não existe processo de *software*

ideal, mas muitas organizações abrem espaços para melhorias no processo de *software*. Organizações nas quais a diversidade de processos de *software* é reduzida, os processos podem ser melhorados através da padronização, possibilitando melhor comunicação e redução no período de treinamento, tornando mais econômico o apoio ao processo automatizado.

> Existem muitos processos de software diferentes, mas todos devem incluir quatro atividades fundamentais para a engenharia de software:
> 1. Especificação de software. A funcionalidade do software e as restrições a seu funcionamento devem ser definidas.
> 2. Projeto e implementação de software. O software deve ser produzido para atender às especificações.
> 3. Validação de software. O software deve ser validado para garantir que atenda às demandas do cliente.
> 4. Evolução de software. O software deve evoluir para atender às necessidades de mudança dos clientes (SOMMERVILLE, 2011, p.18).

Processo de software é representado esquematicamente na figura a baixo.

Figura 1 – Processo de software



Fonte: Sommerville (2011)

1.7 MODELOS DE PROCESSOS DE *SOFTWARE*

1.7.1 MODELO EM CASCATA

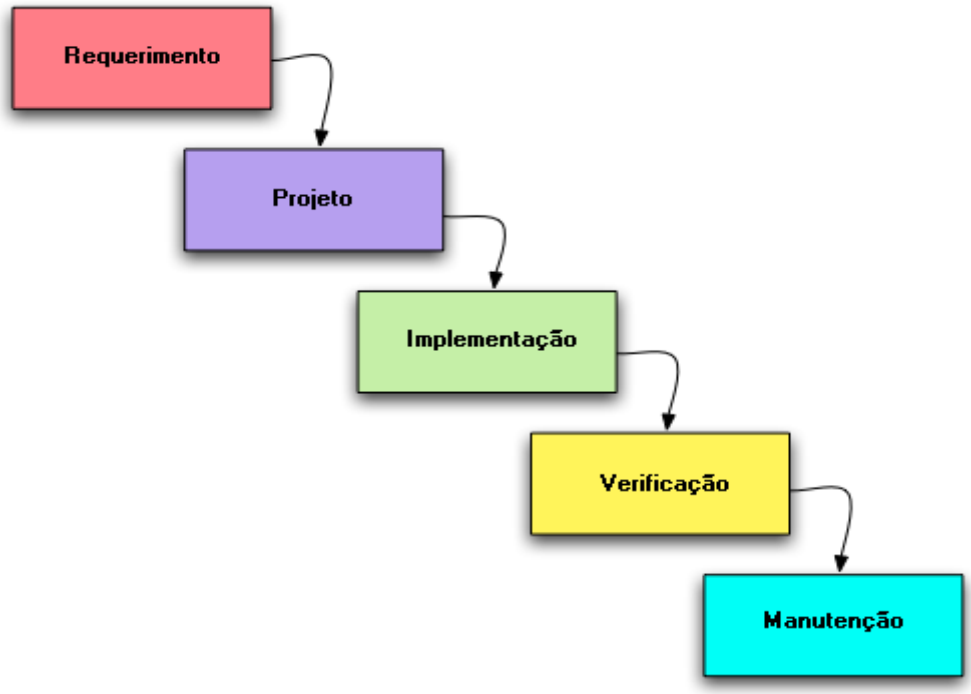
Desenvolve *softwares* sequencialmente de modo que flui sempre para frente, como se fosse uma cascata que segue sempre o curso da água constantemente para frente de acordo com as fases de análise de requisitos, projeto, implantação, teste, integração e manutenção do *software*. Na modelagem em cascata o prosseguimento de uma fase para a próxima segue sempre de forma sequencial, movendo de uma fase para outra somente quando a fase anterior estiver totalmente concluída.

> Os principais estágios do modelo em cascata refletem diretamente as atividades fundamentais do desenvolvimento:
> 1. Análise e definição de requisitos. Os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.
> 2. Projeto de sistema e software. O processo de projeto de sistemas aloca os requisitos tanto para sistemas de hardware como para sistemas de software, por meio da definição de uma arquitetura geral do sistema. O projeto de software envolve identificação e descrição das abstrações fundamentais do sistema de software e seus relacionamentos.
> 3. Implementação e teste unitário. Durante esse estágio, o projeto do software é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação.
> 4. Integração e teste de sistema. As unidades individuais do programa ou programas são integradas e testadas como um sistema completo para assegurar que os requisitos do software tenham sido atendidos. Após o teste, o sistema de software é entregue ao cliente.
> 5. Operação e manutenção. Normalmente (embora não necessariamente), essa é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em uso. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do ciclo de vida, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos (SOMMERVILLE, 2011, p.20).

O modelo em cascata é mais utilizado quando os requisitos de algum problema são compreendidos claramente, por isso o modelo em cascata é mais utilizado para fazer adaptações ou aperfeiçoamento de um *software* já existente. Exemplo: quando uma lei é criada ou modificada deve se fazer adaptações no sistema para se adequar à nova legislação.

Modelo em cascata é o mais importante dos modelos de desenvolvimento de *software*, por servir de referência aos demais modelos e de base para vários projetos, sem mencionar que é utilizado para a documentação abrangendo arquivos de texto e representações gráficas.

Figura 1.2 - Ilustração do modelo em cascata



Fonte: Ferreira (2017, p. 18)

> Em princípio, o resultado de cada estágio é a aprovação de um ou mais documentos (‘assinados’). O estágio seguinte não deve ser iniciado até que a fase anterior seja concluída. Na prática, esses estágios se sobrepõem e alimentam uns aos outros de informações. Durante o projeto, os problemas com os requisitos são identificados; durante a codificação, problemas de projeto são encontrados e assim por diante. O processo de software não é um modelo linear simples, mas envolve o feedback de uma fase para outra. Assim, os documentos produzidos em cada fase podem ser modificados para refletirem as alterações feitas em cada um deles (SOMMERVILLE, 2011, p.21).

1.7.2 PROTOTIPAÇÃO

É o processo que tem como objetivo facilitar o entendimento dos requisitos, apresentando conceitos e funcionalidades do *software*, propondo soluções adequada para o problema do cliente aumentando sua percepção de valor.

O processo de prototipação além de auxiliar o entendimento dos requisitos também nos ajuda a entender o propósito do *software* que será desenvolvido, o negócio do cliente, propor melhorias, minimizar riscos e maximizar lucros.

> Um protótipo é uma versão inicial de um sistema de software, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções. O desenvolvimento rápido e iterativo do protótipo é essencial para que os custos sejam controlados e os *stakeholders* do sistema possam experimentá-lo no início do processo de software (SOMMERVILLE, 2011, p.30).

Protótipos são grandes aliados das metodologias ágeis de desenvolvimento, garantindo maior alinhamento entre a equipe e o cliente. Protótipos são desenvolvidos em níveis diferentes de fidelidade. Protótipo de alta fidelidade leva mais tempo para ser criado ou modificado. O protótipo varia de acordo com o nível de entendimento do negócio, complexidade dos requisitos, prazo e orçamento para elaboração.

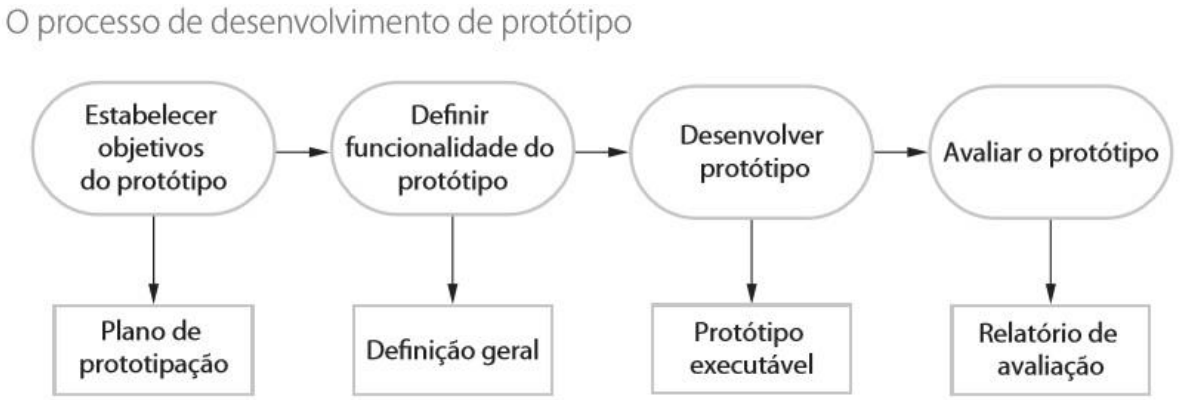
> Um protótipo de software pode ser usado em um processo de desenvolvimento de software para ajudar a antecipar as mudanças que podem ser requisitadas:
> 1. No processo de engenharia de requisitos, um protótipo pode ajudar na e licitação e validação de requisitos de sistema.
> 2. No processo de projeto de sistema, um protótipo pode ser usado para estudar soluções específicas do software e para apoiar o projeto de interface de usuário (SOMMERVILLE, 2011, p.30).

A importância da prototipação está ligada ao meio rápido e eficiente de validar uma ideia, abordagem ou novo conceito.

Prototipação de ser dividida em três categorias:

1. *Wireframes &* Rascunhos; são protótipos de baixa fidelidade, rápidos de se desenvolver e modificar. *Wireframes* não mostram detalhes visuais ou interações de tela, mas ajuda a validar requisitos e regras de negócio de maneira eficiente. É a melhor escolha para representar cenários complexos onde um fluxo ou processo precisa ser compreendido.
2. Protótipos visuais; utiliza programas de edição gráfica em sua criação. É uma ótima opção para telas com maior ênfase em estética e usabilidade, quando os requisitos já foram entendidos, não possuem interações de tela e leva mais tempo para fazer ajustes e melhorias.
3. Protótipos interativos; são completos e representativos. Além da parte visual, englobam uma série de detalhes de estética e efeitos de interação, proporcionando uma experiência rica e realista. Ajuda a equipe identificar novos requisitos, oportunidades e futuros problemas. Protótipo interativo demanda uma equipe de maior conhecimento técnico e demora mais tempo para ser criado. Por outro lado, teremos lucros maiores a longo prazo e riscos menores.

Figura 1.3 – O processo de desenvolvimento de protótipo



Fonte: SOMMERVILLE (2011, P.30)

1.7.3 DESENVOLVIMENTO ITERATIVO E INCREMENTAL

Desenvolvimento Incremental: é um planejamento onde várias partes do sistema são desenvolvidas em paralelo e integradas quando completadas. Desenvolvimento Incremental é desenvolver todo o sistema com uma única integração. Os projetos de softwares que definem requisitos iniciais bem definidos, por necessidade de fornecer um determinado conjunto funcional ao usuário, após esse fornecimento, passamos a melhorar e expandir suas funcionalidades em versões de software posteriores, assim utilizando o desenvolvimento de software incremental.

O modelo de processo incremental aplica sequencias lineares de forma escalonada, à medida que o tempo for avançando. Cada uma das sequências lineares gera um incremento de software. Tais incrementos são entregues ao cliente.

Tabela 1.2 – Vantagem e desvantagem de modelo incremental

MODELO INCREMENTAL

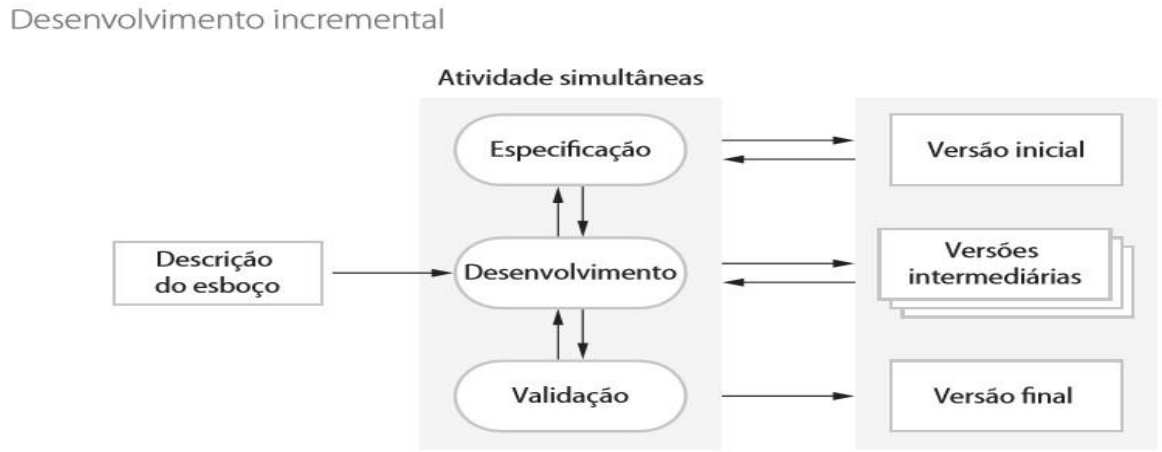
VANTAGENS	DESVANTAGNES
Entregas parciais facilita a identificação e correção de erros entre os componentes	Número de iterações não pode ser definida no início do processo.
Necessidades não especificadas no início pode ser desenvolvida no incremento	O fim do processo não pode ser previamente definido

Cada interação produz um conjunto de itens utilizáveis.	Gerenciamento e manutenção do sistema completo podem se tornar complexos.
Os feedbacks de interações anteriores podem ser usados nos próximos incrementos.	Gerenciamento do custo é mais complexo devido ao número de iterações.
Os incrementos podem ser desenvolvidos por menos profissionais.	
Entrega dos incrementos permite o cumprimento do prazo especificado.	
Facilita a manutenção dos módulos.	

Fonte: Fábio (2017)

> O desenvolvimento incremental é baseado na ideia de desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido (Figura 3). Atividades de especificação, desenvolvimento e validação são intercaladas, e não separadas, com rápido feedback entre todas as atividades (SOMMERVILLE, 2011, p.21).

Figura 3 – Desenvolvimento Incremental



Fonte: SOMMERVILLE (2011, p. 22)

É um clássico modelo de desenvolvimento de *software*, criado em respostas às fraquezas do modelo em cascata. Os padrões mais conhecidos de sistemas iterativos de desenvolvimento são o RUP (Processo Unificado da *Rational*) e o Desenvolvimento ágil de *software*.

Desenvolvimento Iterativo: é um planejamento de retrabalho em que o tempo de revisão e melhorias são pré-definidos. Desenvolvimento iterativo funciona muito bem quando for utilizado junto com o desenvolvimento incremental. Abordagem iterativa é desenvolver um sistema de software incremental, com a seguinte vantagem de aprendizagem mutuo do desenvolvedor e do usuário do sistema. O DI tem como fundamento, iniciar o desenvolvimento com um simples subconjunto de Requisitos de Software e alcançar a evoluções subsequentes das versões até o sistema estar todo implantado. A cada iteração, modificações de projeto são feitas e novas funcionalidades adicionadas.

- O projeto consiste da seguinte etapa de inicialização, iteração e lista de controle do projeto.

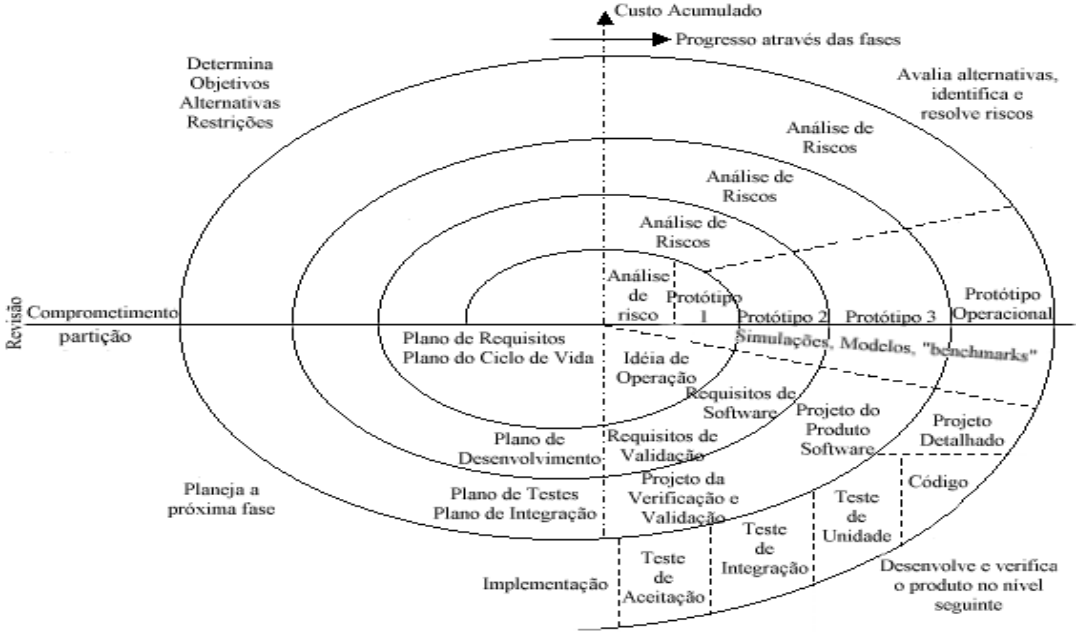
A etapa de inicialização cria uma versão base do sistema. O objetivo desta implementação inicial é criar um produto para que o usuário possa avaliar. Ele deve oferecer um exemplo dos aspectos chave do problema e prover uma solução que seja simples o bastante para que possa ser compreendida e implementada facilmente. Para guiar o processo iterativo, uma lista de controle de projeto é criada. Ela conterá um registro de todas as tarefas que necessitam ser realizadas. Isto inclui itens tais como novas características a serem implementadas e áreas para serem projeto na solução atual. A lista de controle deve ser continuamente revisada como um resultado da fase de análise. A etapa iterativa envolve o re-projeto e implementação das tarefas da lista de controle do projeto e a análise da versão corrente do sistema. O objetivo para o projeto de implementação de qualquer iteração é ser simples, direto e modular, preparado para suportar re-projeto neste estágio ou como uma tarefa a ser adicionada na lista de controle do projeto

1.7.4 MODELO ESPIRAL DE BOEHM

Modelo Espiral tem como objetivo, prover metamodelo que acomode diversos processos específicos, análise de riscos, planejamento e as principais características dos modelos anteriores, adaptando-os a necessidades específicas dos desenvolvedores ou as particularidades do *software* a ser desenvolvido. Também provê prototipação, desenvolvimento evolutivo, cíclico e as principais atividades do modelo em cascata.

Um framework de processo de software dirigido a riscos (o modelo em espiral) foi proposto por Boehm (1988). Isso está na Figura 4. Aqui, o processo de software é representado como uma espiral, e não como uma sequência de atividades com alguns retornos de uma para outra. Cada volta na espiral representa uma fase do processo de software. Dessa forma, a volta mais interna pode preocupar-se com a viabilidade do sistema; o ciclo seguinte, com definição de requisitos; o seguinte, com o projeto do sistema, e assim por diante. O modelo em espiral combina prevenção e tolerância a mudanças, assume que mudanças são um resultado de riscos de projeto e inclui atividades explícitas de gerenciamento de riscos para sua redução (SOMMERVILLE, 2011, p.32).

Figura 4 – Modelo espiral de processo de software de Boehm (@IEEE 1988)



Fonte: SOMMERVILLE (2011, p. 33)

Cada volta da espiral é dividida em quatro setores:

1. Definição de objetivos. Objetivos específicos para essa fase do projeto são definidos; restrições ao processo e ao produto são identificadas, e um plano de gerenciamento detalhado é elaborado; os riscos do projeto são identificados. Podem ser planejadas estratégias alternativas em função desses riscos.
2. Avaliação e redução de riscos. Para cada um dos riscos identificados do projeto, é feita uma análise detalhada. Medidas para redução do risco são tomadas. Por exemplo, se houver risco de os requisitos serem inadequados, um protótipo de sistema pode ser desenvolvido.
3. Desenvolvimento e validação. Após a avaliação dos riscos, é selecionado um modelo de desenvolvimento para o sistema. Por exemplo, a prototipação descartável pode ser a melhor abordagem de desenvolvimento de interface de usuário se os riscos forem dominantes. Se os riscos de segurança forem a principal consideração, o desenvolvimento baseado em transformações formais pode ser o processo mais adequado, e assim por diante. Se o principal risco identificado for a integração de subsistemas, o modelo em cascata pode ser a melhor opção.
4. Planejamento. O projeto é revisado, e uma decisão é tomada a respeito da continuidade do modelo com mais uma volta da espiral. Caso se decida pela

> continuidade, planos são elaborados para a próxima fase do projeto (SOMMERVILLE, 2011, p.33).

É um modelo de processo de *software* evolucionário e iterativo com a prototipação, com aspectos sistemáticos e controlados por modelo cascata. Modelagem em espiral fornece grande potência para desenvolver versões de *software* cada vez mais rápido e completa.

Análise de riscos e planejamento são realizados durante todo o processo de desenvolvimento e evolução do projeto

Modelo em Espiral é iniciado sempre pelo centro do espiral e prosseguimos no sentido horário. Na medida que cada evolução é realizada, os riscos são considerados. A primeira etapa deve se especificar o produto, depois desenvolvemos um protótipo e vamos evoluindo para versões cada vez mais sofisticadas do sistema. Cada passagem pela parte de planejamento, resulta em ajustes no planejamento do projeto. Custo e Cronograma deve ser ajustado sempre que houver feedback do cliente após uma entrega.

O modelo espiral é mais adequado para sistemas complexos e que exijam um alto nível de interações com os usuários a fim de possibilitar a abordagem de todos os problemas do sistema. É mais frequentemente utilizado em grandes projetos e internos.

Tabela 1.3 – Vantagem e desvantagem do modelo espiral

MODELO ESPIRAL

VANTAGENS	**DESVANTEGENS**
Pode ser adicionada novas funcionalidades em cada novo ciclo.	Destinado exclusivamente a desenvolvimento de software interno.
Não existe distinção entre desenvolvimento e pós-entrega.	A abordagem deste modelo exige grande experiência na avaliação dos riscos.

Maior controle sobre os riscos do projeto, tornando o processo de construção de um produto complexo mais seguro.	É difícil convencer grandes clientes de que a abordagem evolutiva é controlável.

Fonte: Ferreira (2017)

> "A principal diferença entre o modelo espiral e outros modelos de processo de software é seu reconhecimento explícito do risco" (SOMMERVILLE, 2011, p.33).

1.7.5 DESENVOLVIMENTO ÁGIL DE *SOFTWARE*

Metodologias ágeis, tornou se conhecida em 2001, quando especialistas em processos de desenvolvimento de software representando os métodos *Scrum* e Extreme *Programming* (XP), estabeleceram princípios e características comuns destes métodos. Assim foi criada o "Manifesto Ágil".

Extreme Programming (XP): é uma metodologia de *software* que auxilia na criação de *softwares* de qualidade em menor tempo e de forma econômica. Pequenos conjuntos de valores, princípios e prática faz com o que os objetivos sejam alcançados.

> Extreme Programming (XP) é talvez o mais conhecido e mais utilizado dos métodos ágeis. O nome foi cunhado por Beck (2000), pois a abordagem foi desenvolvida para impulsionar práticas reconhecidamente boas, como o desenvolvimento iterativo, a níveis 'extremos'. Por exemplo, em XP, várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes (SOMMERVILLE, 2011, pag. 44).

Valores do XP: são as pessoas. Porque? Pessoas se diferem umas das outras, com opiniões, conduta, conhecimentos e práticas desenvolvidas. Mas para o XP o importante é como esses indivíduos se comportaram perante um desenvolvimento em equipe. As pessoas devem pensar em que é importante para à equipe e deixar de lado certas particularidades, como modo de programar, documentar e etc.... deve ser alinhado a melhor forma para equipe e resultados satisfatórios. Para que o trabalho em equipe do desenvolvimento ágil XP funcione, é preciso seguir os seguintes valores para orientar no desenvolvimento;

1. Comunicação
2. Coragem
3. Feedback

4. Respeito
5. Simplicidade

- Princípios do XP: princípios existem para servir de ponte entre valores e prática, princípios servem como guias que se aplicam a um domínio específico. Valores, significa o propósito, ou seja, aquilo em que se acredita, razão pela qual agimos de certa maneira. Praticas, são nossas ações, o que efetivamente fazemos para desenvolver um projeto.

- Práticas do XP: práticas em XP, faz com que a equipe progrida em direção a eficácia do desenvolvimento e seus objetivos. Práticas devem ser usadas em conjunto, uma por vez garantido que os resultados do desenvolvimento e a progressão do projeto seja gradualmente e em harmonia, garantindo qualidade e proatividade no desenvolvimento até o estágio final.

> Há algum tempo já se reconhecia a necessidade de desenvolvimento rápido e de processos capazes de lidar com mudanças nos requisitos. Na década de 1980, a IBM introduziu o desenvolvimento incremental (MILLS et al., 1980). A introdução das linguagens de quarta geração, também em 1980, apoiou a ideia de desenvolvimento e entrega rápidos de software (MARTIN, 1981). No entanto, a ideia realmente decolou no final da década de 1990, com o desenvolvimento da noção de abordagens ágeis, como Metodologia de Desenvolvimento de Sistemas Dinâmicos (DSDM, do inglês *dynamics systems development method*) (STAPLETON, 1997), *Scrum* (SCHWABER e BEEDLE, 2001) e *Extreme Programming* (BECK, 1999; BECK, 2000) (SOMMERVILLE, 2011, p.39).

O desenvolvimento ágil é um conjunto de metodologia de desenvolvimento que adotam os princípios do manifesto ágil. Tais princípios são os seguintes:

- Indivíduos e interação entre eles, mais que processos e ferramentas
- Software em funcionamento, mais que documentação abrangente
- Colaboração com o cliente, mais que negociação de contratos
- Responder a mudanças, mais que seguir um plano

Os projetos que utilizam desenvolvimento ágil, adotam o modelo iterativo e em espiral. Todas as etapas descritas na modelo cascata são executadas diversas vezes ao longo do projeto, produzindo ciclos curtos que repetem ao longo do projeto, onde no final de cada ciclo, sempre se tem um *software* funcionando, testado e aprovado. As iterações crescem em números de funcionalidades a cada repetição de ciclo, onde que no último ciclo, todas as funcionalidades estarão implementadas, testadas e aprovadas.

> Nos dias de hoje, as empresas operam em um ambiente global, com mudanças rápidas. Assim, precisam responder a novas oportunidades e

> novos mercados, a mudanças nas condições econômicas e ao surgimento de produtos e serviços concorrentes. Softwares fazem parte de quase todas as operações de negócios, assim, novos softwares são desenvolvidos rapidamente para obterem proveito de novas oportunidades e responder às pressões competitivas. O desenvolvimento e entrega rápidos são, portanto, o requisito mais crítico para o desenvolvimento de sistemas de software. Na verdade, muitas empresas estão dispostas a trocar a qualidade e o compromisso com requisitos do software por uma implantação mais rápida do software de que necessitam (SOMMERVILLE, 2011, p.38).

Tabela 1.4 – Os princípios dos métodos ágeis

Princípios	**Descrição**
Envolvimento do cliente	Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Fonte: SOMMERVILLE (2011, p. 40)

O desenvolvimento de métodos ágeis tem sido melhor aproveitado e utilizado em desenvolvimento de sistemas pequenos e médios ou em sistemas personalizados dentro da organização.

> Na prática, os princípios básicos dos métodos ágeis são, por vezes, difíceis de se concretizar: 1. Embora a ideia de envolvimento do cliente no processo de desenvolvimento seja atraente, seu sucesso depende de um cliente disposto e capaz de passar o tempo com a equipe de desenvolvimento, e que possa representar todos os stakeholders do sistema. Frequentemente, os representantes dos clientes estão sujeitos a diversas pressões e não podem participar plenamente do desenvolvimento de software. 2. Membros individuais da equipe podem não ter personalidade adequada para o intenso envolvimento que é típico dos métodos ágeis e, portanto, não interagem bem com outros membros da equipe. 3. Priorizar as mudanças pode ser extremamente difícil, especialmente em sistemas nos quais existem muitos stakeholders. Normalmente, cada stakeholder dá prioridades diferentes para mudanças diferentes. 4. Manter a simplicidade exige um trabalho extra. Sob a pressão de cronogramas de entrega, os membros da equipe podem não ter tempo para fazer as simplificações desejáveis. 5. Muitas organizações,

> principalmente as grandes empresas, passaram anos mudando sua cultura para que os processos fossem definidos e seguidos. É difícil para eles mudar de um modelo de trabalho em que os processos são informais e definidos pelas equipes de desenvolvimento (SOMMERVILLE, 2011, p.41).

1.7.6 GERENCIAMENTO ÁGIL DE PROJETOS

Gerenciamento de projetos dirigida a planos é a abordagem padrão para o desenvolvimento e utilização do método ágil.

> “A principal responsabilidade dos gerentes de projeto de software é gerenciar o projeto para que o software seja entregue no prazo e dentro do orçamento previsto. Eles supervisionam o trabalho dos engenheiros de software e acompanham quão bem o desenvolvimento de software está progredindo” (SOMMERVILLE, 2011, pag. 49).

Um gerente de projeto de *software* tem como tarefa mais importante, o planejamento. O gerente precisa dividir o trabalho e cada parte deve ser designada a um membro da equipe; deve prever os problemas que surgirem e preparar soluções para eles. Quando se cria um plano de projeto no início o mesmo é utilizado para fazer a comunicação entre os clientes e avaliar o progresso do cliente.

O planejamento do projeto ocorre em três estágios:

1. Proposta, quando se propõem o contrato de desenvolvimento de *software*, decisão dos recursos e o preço a ser cotado para o cliente.
2. Planejamento, definição de quem irá trabalhar no projeto e como será dividido as tarefas de cada participante e como alocar os recursos na empresa.
3. Modificação do planejamento; como o decorrer do projeto, se adquire experiência e conhecimento sobre o funcionamento do *software* em construção, podendo alocar colaboradores de uma tarefa para outra de acordo com o grau de conhecimento e fazer estimativas mais precisas quanto de duração de cada ciclo e alterações alteração de requisitos que faz com que o cronograma mude.

> Existem três parâmetros principais que você deve usar ao calcular os custos de um projeto de desenvolvimento de software:
>
> - Custos de esforço (os custos de pagamentos de gerentes e engenheiros de software);
> - Custos de hardware e software, incluindo manutenção;

> - Os custos de viagens e treinamentos (SOMMERVILLE, 2011, pag. 432).

Após a concretização do contrato, deve se saber mais sobre os requisitos e criação de um plano de projeto a ser usado como suporte na tomada de decisão, como formação de equipe e elaboração do orçamento, podendo assim alocar recursos para o projeto e contratação de funcionários se necessário.

O Plano de projeto sofre evolução constante enquanto o processo de desenvolvimento e executado, o planejamento também deve garantir que o plano continue sendo um documento útil para a equipe compreender o objetivo a ser alcançado e data de entrega. Por isso o cronograma, riscos e estimativas de custos devem ser sempre atualizados conforme o desenvolvimento do *software*.

> Se um método ágil é usado, existe ainda a necessidade de um plano de iniciação de projeto, independentemente da abordagem escolhida; a empresa ainda precisa planejar como os recursos para o projeto serão alocados. No entanto, esse não é um plano detalhado e deve incluir apenas informações limitadas sobre a divisão do trabalho e o cronograma de projeto. Durante o desenvolvimento, um plano de projeto e as estimativas de esforço informais são delineados para cada versão do software, com toda a equipe envolvida no processo de planejamento (SOMMERVILLE, 2011, pag. 433).

O desenvolvimento ágil precisa ser gerenciado para se fazer um melhor aproveitamento dos recursos e do tempo disponíveis para a equipe, com isso a abordagem deve ser diferente, com adaptações ao método incremental e os pontos fortes dos métodos ágeis.

O método ágil geral é a abordagem *Scrum*, que tem o foco no desenvolvimento interativo.

O *Scrum* é composto por três fases:

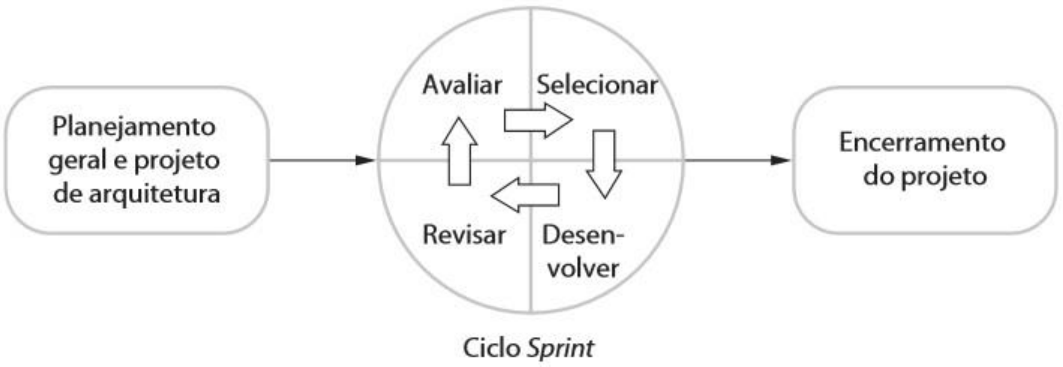
1. Planejamento geral, estabelecendo objetivos e arquitetura do projeto.
2. *Sprint*, são vários ciclos de *Sprint* onde cada ciclo desenvolve um incremento do sistema.
3. Encerramento do projeto, fornecendo documentação completa, como manual de usuário e quadro de ajuda do sistema e avaliação das lições aprendidas com o projeto.

> A característica inovadora do Scrum é sua fase central, chamada ciclos de Sprint. Um Sprint do Scrum é uma unidade de planejamento na qual o trabalho a ser feito é avaliado, os recursos para o desenvolvimento são selecionados e o software é implementado. No fim de um Sprint, a

funcionalidade completa é entregue aos stakeholders. As principais características desse processo são:

1. Sprint são de comprimento fixo, normalmente duas a quatro semanas. Eles correspondem ao desenvolvimento de um release do sistema em XP.
2. O ponto de partida para o planejamento é o backlog do produto, que é a lista do trabalho a ser feito no projeto. Durante a fase de avaliação do Sprint, este é revisto, e as prioridades e os riscos são identificados. O cliente está intimamente envolvido nesse processo e, no início de cada Sprint, pode introduzir novos requisitos ou tarefas.
3. A fase de seleção envolve todos da equipe do projeto que trabalham com o cliente para selecionar os recursos e a funcionalidade a ser desenvolvida durante o Sprint.
4. Uma vez que todos estejam de acordo, a equipe se organiza para desenvolver o software. Reuniões diárias rápidas, envolvendo todos os membros da equipe, são realizadas para analisar os progressos e, se necessário, repriorizar o trabalho. Nessa etapa, a equipe está isolada do cliente e da organização, com todas as comunicações canalizadas por meio do chamado 'Scrum Master'. O papel do Scrum Master é proteger a equipe de desenvolvimento de distrações externas. A maneira como o trabalho é desenvolvido depende do problema e da equipe. Diferentemente do XP, a abordagem Scrum não faz sugestões específicas sobre como escrever os requisitos ou sobre o desenvolvimento test-first etc. No entanto, essas práticas de XP podem ser usadas se a equipe achar que são adequadas.
5. No fim do Sprint, o trabalho é revisto e apresentado aos stakeholders. O próximo ciclo Sprint começa em seguida (SOMMERVILLE, 2011, pag. 50).

Figura 1.5 – O processo *Scrum*



Fonte: SOMMERVILLE (2011, p. 50)

2 MODELAGEM E REQUISITOS DE SOFTWARE

A modelagem de software junto com os requisitos é uma das partes mais importante no projeto de desenvolvimento de software. É nessa hora que identificamos o problema e planejamos a ação à ser tomada para solucionar o problema.

> As linguagens gráficas de modelagem existem há muito tempo na indústria do software. O propulsor fundamental por trás de todas elas é que as linguagens de programação não estão em um nível de abstração suficientemente alto para facilitar as discussões sobre projeto (FOWLER, 2005, p.25).

> As pessoas que criam software praticam a arte ou ofício ou disciplina em que consiste a engenharia de software. Mas em que consiste a “prática” de engenharia de software? De forma genérica, prática é um conjunto de conceitos, princípios, métodos e ferramentas aos quais um engenheiro de software recorre diariamente. A prática permite aos coordenadores (gerentes) administrar os projetos e aos engenheiros de software criar programas computacionais. A prática preenche um modelo de processo de software com recursos de como fazer para ter o trabalho realizado em termos de gerenciamento e de tecnologia. Transforma uma abordagem desfocada e confusa em algo mais organizado, mais efetivo e mais propenso a obter sucesso (PRESSMAN, 2011, p.109).

2.1 PRINCÍPIOS FUNDAMENTAIS

A engenharia de *software* e guiada por um conjunto de princípios fundamentais que auxiliam na aplicação de um projeto de *software*, os princípios fundamentais estabelecem uma infraestrutura filosófica que guia a equipe de *software*, estabelecendo artefatos e regras que servem de guias, enquanto se analisa um problema ou uma solução.

> “Segundo Jon Van de Snepscheut, “Teoricamente, não há nenhuma diferença entre a teoria e a prática. Porém, na prática, ela existe” (PRESSMAN, 2011, p.109).

Os princípios fundamentais que abrange o processo e a prática da engenharia de *software* são:

1. Fornecer valor aos usuários finais,
2. Simplificar,
3. Manter a visão do produto
4. Reconhecer que outros usuários consomem (e deve entender) o que se produz,
5. Manter abertura para o futuro,
6. Planejar antecipadamente para o reuso

7. Raciocinar

> “Embora tais princípios gerais sejam importantes, são caracterizados em um nível tão elevado de abstração que, às vezes, torna-se difícil a tradução para a prática diária da engenharia de software” (PRESSMAN, 2011, p.110).

2.2 PRINCÍPIOS QUE ORIENTAM O PROCESSO

A seguir será abordado um conjunto de princípios fundamentais que pode ser aplicado em todos os processos de *software* e metodologias.

> “(Aviso) Todo projeto e toda equipe são únicos. Isso significa que se deve adaptar o processo para que melhor se ajuste a suas necessidades” (PRESSMAN, 2011, p.110).

1. Seja ágil: manter abordagem técnica tão simples quanto possível, manter os produtos tão concisos quanto possível e tomar decisões localmente sempre que possível. Os princípios básicos do desenvolvimento ágil devem comandar sua abordagem.
2. Concentração na qualidade em todas as etapas: a condição final para toda atividade, ação e tarefa do processo deve focar na qualidade do produto produzido.
3. Estar pronto para adaptações: quando necessário, a abordagem deve ser adaptada as restrições impostas pelo problema, pessoas e o próprio projeto.
4. Montar uma equipe: pessoas são o fator mais importante, por isso a equipe deve ser capaz de se auto organizar e tenha confiança e respeito mútuos.
5. Estabelecer mecanismo para comunicação e coordenação: são itens de gerenciamento e devem ser direcionados.
6. Gerenciar mudanças: estabelecer a maneira como as mudanças serão solicitadas, avaliadas, aprovadas e implementadas.
7. Avaliação de riscos: é fundamental estabelecer planos de contingência em caso de erro quando se desenvolve um *software*.
8. Gerar artefatos que forneçam valores para outros: os artefatos devem proporcionar valor para outro processo, atividades, ações ou tarefas; não devem conter ambiguidades ou omissões, para que possa ser utilizado por outra pessoa.

> “Segundo, General H. Norman Schwarzkof; “A verdade da questão é que sempre sabemos a coisa certa a ser feita. A parte difícil é fazê-la” (PRESSMAN, 2011).

2.3 PRINCÍPIOS QUE ORIENTAM A PRÁTICA

A prática da engenharia de *software* tem como objetivo principal; entregar dentro do prazo, com qualidade, características e funcionalidade do *software*, satisfaçam as necessidades de todos os envolvidos.

> Tais princípios são importantes, independentemente do método de análise ou projeto aplicado, das técnicas de desenvolvimento (por exemplo, linguagem de programação, ferramentas de automação) escolhidas, ou da abordagem de verificação e validação, ferramentas de automação) escolhidas, ou abordagem de verificação e validação utilizada (PRESSMAN, 2011, p.111).

1. Dividir e conquistar: separação por interesses, terá a resolução de um problema mais facilmente, cada interesse fornece uma funcionalidade diferente.
2. Compreender o uso da abstração: na engenharia de *software* usa se abstrações de diferentes níveis, cada um incorporando um significado que deva ser comunicado.

> Joel Spolsky [Spo02] sugere que “todas as abstrações não triviais são, até certo ponto, frágeis”. O objetivo de uma abstração é eliminar a necessidade de comunicar detalhes. Algumas vezes, efeitos problemáticos se precipitam em virtude da “aresta” dos detalhes. Sem a compreensão dos detalhes, a causa de um problema não poderá ser facilmente diagnosticada. (PRESSMAN, 2011, p.111).

3. Esforçar por consistência: é sugerido que um contexto familiar facilita o uso do *software*
4. Focar na transferência de informação: a transferência é tratada a partir do banco de dados para um usuário final, de um *software* judiciário para a aplicação web, do usuário final para a interface gráfica e etc...
5. Construir *software* que apresente modularidade efetiva: a modularidade deve ser efetiva a cada módulo, focalizando aspectos exclusivos e restritos do *software*, deve ser preciso na apresentação do conteúdo e coeso em suas funções e interconectados de maneira simples.
6. Padronização: Brad Appleton [App00] propõe que:

> O objetivo da padronização é criar uma fonte literária para ajudar os desenvolvedores de software a solucionar problemas recorrentes, encontrados ao longo de todo o desenvolvimento de software. Os padrões ajudam a criar uma linguagem compartilhada para transmitir conteúdo e experiência acerca dos problemas e de suas soluções. Codificar formalmente tais soluções e suas relações permite que se armazene com sucesso a base do conhecimento, a qual define nossa compreensão sobre boas arquiteturas,

> correspondendo às necessidades de seus usuários (PRESSMAN, 2011, p.112).

7. Quando possível, representar o problema e sua solução sob uma série de perspectivas diferentes: em analisar problemas e soluções deve se ter diferentes perspectivas dos erros e omissões reveladas.

8. Lembrar que outra pessoa fará a manutenção do *software*: a manutenção pode ser aplicada com facilidade se houver uma prática de engenharia de *software* consistente.

Os princípios acima estabelecem uma base para todos os métodos de engenharia de *software*.

2.4 PRINCÍPIOS DA COMUNICAÇÃO

A comunicação entre as partes interessadas é muito importante para o desenvolvimento do projeto, é a comunicação constante que ajuda o desenvolvedor a criar a solução correta para o cliente e deixar todos felizes.

> Antes que os requisitos dos clientes sejam analisados, modelados ou especificado, eles devem ser coletados através da atividade de comunicação. Um cliente apresenta um problema que pode ser amenizado por uma solução baseada em computador. Responde-se ao pedido de ajuda e inicia-se a comunicação. Entretanto, o percurso da comunicação ao entendimento é, frequentemente, acidentado.
> A comunicação efetiva (entre parceiros técnicos com o cliente, com outros parceiros interessados e com gerenciadores de projetos) constitui uma das atividades mais desafiadoras. Nesse contexto, discutem se princípios de comunicação aplicados na comunicação com o cliente. Entretanto, muitos se aplicam a todas as formas de comunicação. (PRESSMAN, 2011, p.112).

2.5 PRINCÍPIOS DE PLANEJAMENTO

1. Compreender o escopo do projeto: indicação de destino

2. Envolver os interessados na atividade de planejamento: definir prioridades e restrições de projeto, negociando frequentemente cronogramas de entrega e outras questões referente ao projeto.

3. Reconhecer que o planejamento é interativo: alterações podem ocorrer e por isso o plano deverá ser ajustado.

4. Fazer estimativas com base no que conhece: oferece indicações de esforços, custo e prazo para a realização, com compreensão do trabalho a ser realizado.

5. Considerar os riscos ao definir o plano: ao detectar riscos de alto impacto e probabilidade, o plano de contingência será necessário.

6. Seja realista: sempre ocorre problemas de comunicação, omissão e ambiguidades. Porque todos cometemos erros e deve ser levado em consideração.

7. Ajustar particularidades ao definir o plano: são níveis de detalhamentos existente no plano de projeto.

8. Definir como se pretende garantir a qualidade: a qualidade deve ser determinada no plano de projeto junto com a equipe.

9. Descrever como pretende adequar as alterações: as alterações deveram ser identificadas ao longo do trabalho.

10. Verificar o plano frequentemente e fazer ajustes necessários: é preciso checar diariamente o progresso do projeto com intuído de prevenir atrasos na entrega.

> “General Dwight D. Eisenhower diz que: “Na preparação para a batalha os planos não eram uteis, porem o planejamento é indispensável” (PRESSMAN, 2011, p.114).

2.6 PRINCÍPIOS DE MODELAGEM

São vários os princípios de modelagem, e entre eles estão os:

- O objetivo principal da equipe de software é construir software, e não criar modelos.
- Criar somente os modelos necessários,
- Criar modelos mais simples possível,
- Criar modelos que facilitem alterações,
- Estabelecer proposito claro para cada modelo,
- Adaptar o modelo ao sistema disponível,
- Criar modelos úteis e não perfeitos,
- Deve transmitir o conteúdo claro e com sucesso,
- Se ocorrer controvérsias entre os modelos, há motivos para preocupar,

2.7 PRINCIPIOS DE MODELAGEM DE REQUISITOS

Os princípios abaixo devem ser utilizados pelos que utilizam o modelo de processos ágeis.

- As informações devem ser representadas e compreendidas corretamente,
- As funções do software definidas,
- O comportamento do software deve ser representado,
- Os modelos descrevendo funções e comportamentos devem ser divididas por camadas para revelarem seus detalhes,
- A análise se inicia nas informações essenciais e depois os detalhes,

> Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado engenharia de requisitos (RE, do inglês requirements engineering). (SOMMERVILLE (2011, p.57).

> No trabalho de engenharia de software, podem ser criadas duas classes de modelos: modelos de requisitos e modelos de projetos. Os modelos de requisitos (também denominados modelos de análise) representam os requisitos dos clientes, descrevendo o software em três domínios diferentes: o domínio da informação, o domínio funcional e o domínio comportamental. Os modelos de projeto representam características do software que ajudam os desenvolvedores a construí-lo efetivamente: a arquitetura, a interface para o usuário e os detalhes quanto a componentes (PRESSMAN, 2011,p.116).

2.8 PRINCÍPIOS DE MODELAGEM DE PROJETOS

Existem vários métodos para se identificar os vários elementos de um projeto, entre eles estão:

- Roteiro do projeto na modelagem de requisitos,
- Consideração da arquitetura do sistema a ser construído,
- Importância do projeto de dados como do projeto das funções e processamento,
- Cuidado na projeção das interfaces internas e externas,
- A interface de usuário final deve facilitar o uso,
- Em nível de componentes deve ser funcionalmente independente,

- Os componentes devem se relacionar livremente entre componentes e ambiente externo,
- Os modelos devem ser de fácil compreensão,
- Importante que o desenvolvimento seja interativo e que em cada interação deve se obter um maior grau de simplicidade.

> Se uma empresa pretende fechar um contrato para um projeto de desenvolvimento de software de grande porte, deve definir as necessidades de forma abstrata o suficiente para que a solução para essas necessidades não seja predefinida. Os requisitos precisam ser escritos de modo que vários contratantes possam concorrer pelo contrato e oferecer diferentes maneiras de atender às necessidades da organização do cliente. Uma vez que o contrato tenha sido adjudicado, o contratante deve escrever para o cliente uma definição mais detalhada do sistema, para que este entenda e possa validar o que o software fará. Ambos os documentos podem ser chamados documentos de requisitos para o sistema (SOMMERVILLE, 2011, p.57).

> "Ponto chave, os modelos de requisitos, representam os requisitos dos clientes. Os modelos de projetos oferecem uma especificação concreta para a construção do software" (PRESSMAN, 2011, p.116).

2.9 ENGENHARIA DE REQUISITOS

Requisitos é a descrição de que um *software* deve fazer, os serviços que vai oferecer e restrições de funcionamento. Requisito reflete a necessidade do cliente em sistema e sua finalidade. Engenharia de requisitos, é o processo de descobrir, analisar e documentar o que deve ser feito.

Geralmente os requisitos, são apenas uma declaração abstrata em alto nível de uma restrição ou serviço que o *software* deve oferecer, por outro lado, pode ser uma definição detalhada e formal do que o sistema deverá fazer.

Os requisitos devem ser organizados de acordo com a necessidade e distinto uns dos outros, como requisitos de usuários, requisitos dos sistemas, requisitos funcionais e não funcionais.

Os requisitos devem ser escritos em diferentes níveis de detalhamento para que qualquer pessoa possa compreender e usá-los de maneira adequada.

> Entender os requisitos de um problema está entre as tarefas mais difíceis enfrentadas por um engenheiro de software. Ao pensar nisso pela primeira vez, a engenharia de requisitos não parece assim tão difícil. Afinal de contas, o cliente não sabe o que é necessário? Os usuários finais não deveriam ter um bom entendimento das características e funções que trarão benefícios? Surpreendentemente, em muitos casos a resposta a essas perguntas é "não". E mesmo se os clientes e usuários finais fossem explícitos quando às suas necessidades, essas mudariam ao longo do projeto (PRESSMAN, 2011, p.116).

2.1.1 ANÁLISE DE REQUISITOS DE SOFTWARE

É à parte que lida com a investigação, definição e escopo de novo *software* ou alteração de um existente.

Analisar requisitos é importante no desenvolvimento de *software* para identificar as necessidades do cliente e onde o analista de negócios trabalha junto com o desenvolvedor. Após a definição dos requisitos, o projetista estará pronto para desenvolver a solução.

A análise de requisitos deve ser a primeira parte do desenvolvimento e deve ser revista sempre que uma parte do projeto for concluída, é a fase que se identifica a maioria dos erros e onde é dialogado com cliente e testado se é aquilo que o cliente deseja. Na análise de requisitos entrevistas constantes são realizadas, melhorias e concertos ao mesmo tempo.

Análise de requisitos é importante no projeto por ser responsável por coletar dados, necessários e importante para solucionar o problema do cliente e permitir que o mesmo alcance seus objetivos com o *software* desenvolvido.

Os requisitos são adquiridos através de entrevistas realizadas constantemente, após cada etapa ou fase realizada, os requisitos são adquiridos e divididos por partes ou tipos.

Os requisitos também devem ser separados por níveis de descrição entre os mesmos, usando o termo ‘requisitos de usuário’, expressando os requisitos abstratos de alto nível, e ‘requisitos de sistema’, expressando a descrição detalhada do que o sistema deva fazer.

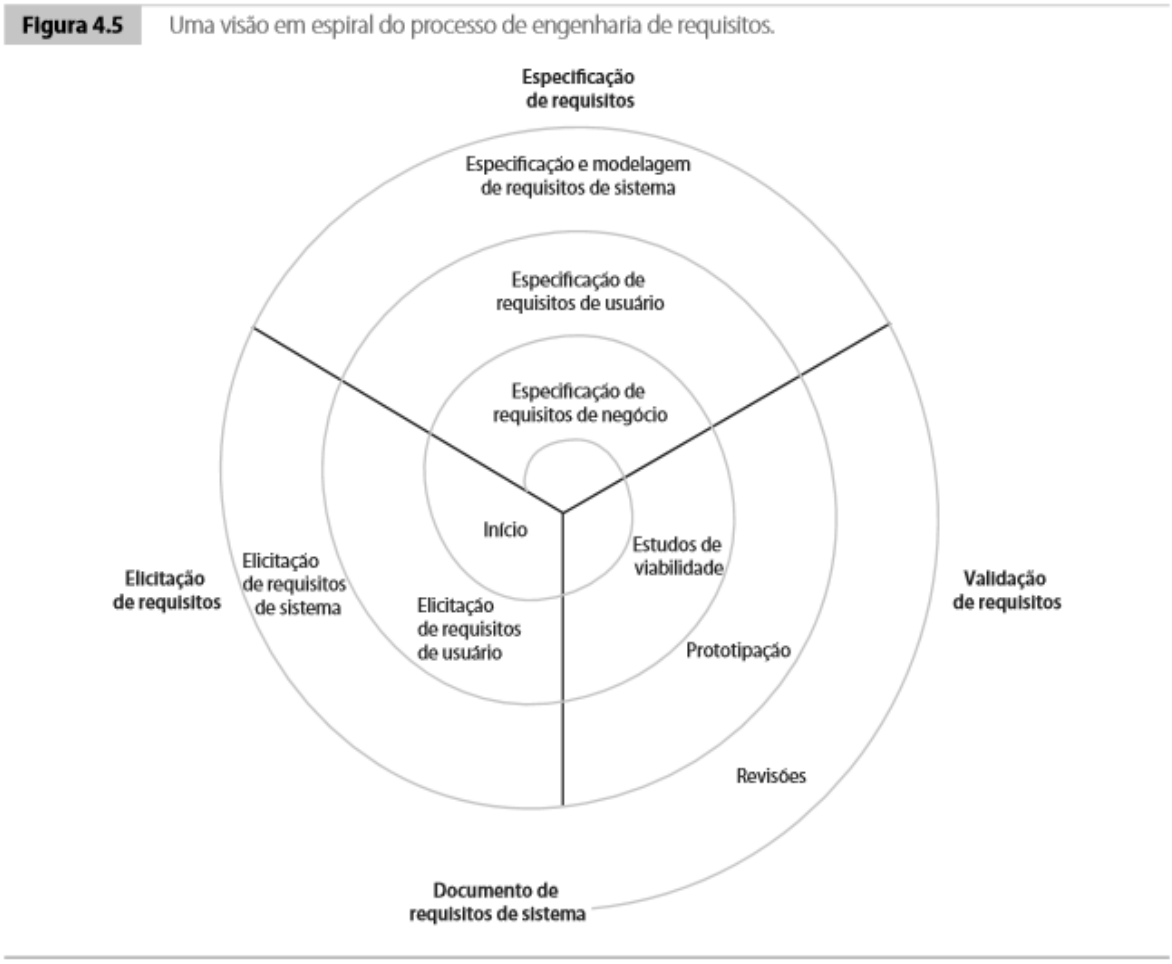
Após estudo de viabilidade, o próximo passo de engenharia de requisitos é a e licitação e análise de requisitos. Nessa parte o trabalho dos engenheiros de software é realizado junto com o cliente e usuário final do sistema para adquirir informações referentes aos serviços que o sistema deva oferecer, desempenho do sistema, domínio da aplicação e restrições de hardware e etc...

> Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços oferecem e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado engenharia de requisitos (RE, do inglês requirements engineering), (SOMMERVILLE, 2011, p. 57).

> A engenharia de requisitos fornece o mecanismo apropriado para entender aquilo que o cliente deseja, analisando as necessidades, avaliando a

viabilidade, negociando uma solução razoável, especificando a solução sem ambiguidades, validando a especificação e gerenciando as necessidades à medida que são transformadas em um sistema operacional [Tha97]. Ela abrange sete tarefas distintas: concepção, levantamento, elaboração, negociação, especificação, validação e gestão. É importante notar que algumas delas correm em paralelo e todas são adaptadas às necessidades do projeto (PRESSMAN, 2011, p.129).

Figura 5 – Engenharia de requisitos em espiral



Fonte: Ian Sommerville (2011, p. 58)

2.1.2 REQUISITOS DE USUÁRIO E DE SISTEMA

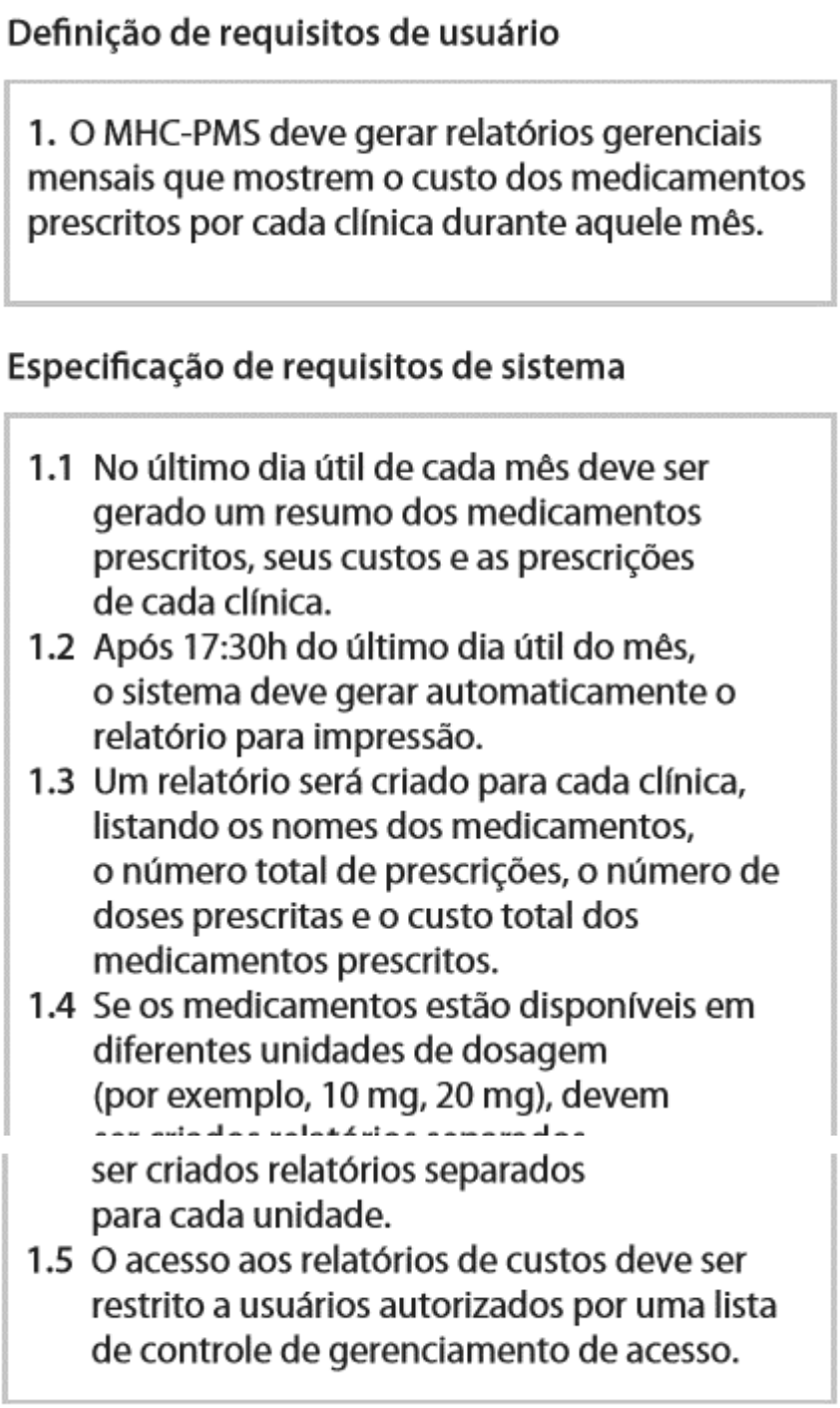
- Requisitos de usuários: são declarações, em língua natural com ‘diagramas’, de quais serviços o software deverá fornecer aos usuários e restrições de operação.

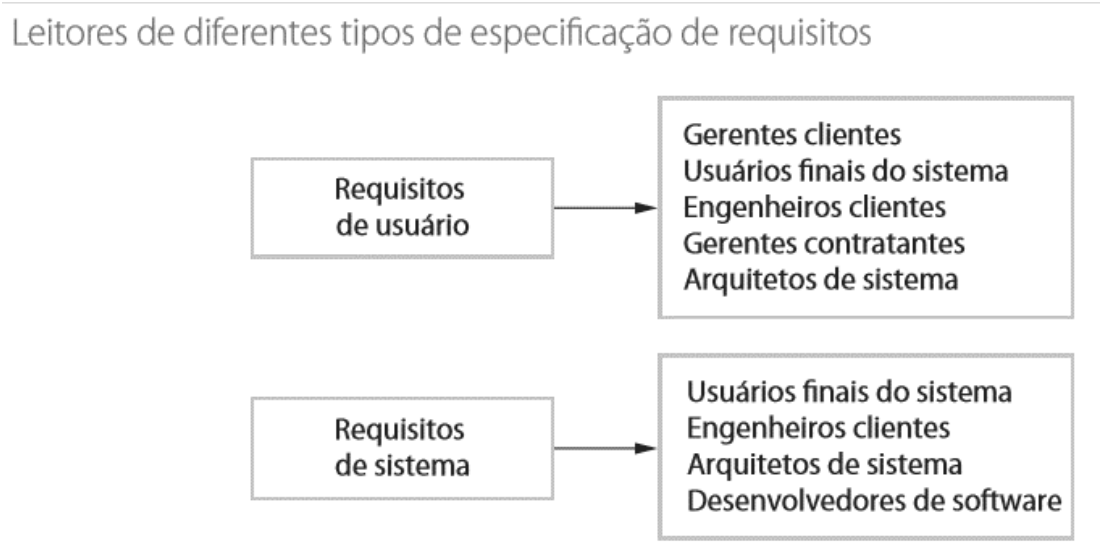
- Requisitos de sistema: são descrições detalhadas das funções, serviços e restrições operacionais do sistema.

É de fundamental importância que os requisitos sejam escritos em diferentes níveis de detalhamento para que as informações referentes ao sistema, possa ser usada diversas maneiras por diferentes leitores.

> Em um ambiente ideal, os interessados e os engenheiros de software trabalham juntos na mesma equipe. Em tais casos, a engenharia de requisitos é apenas uma questão de conduzir conversações proveitosas com colegas que sejam membros bem conhecidos da equipe. Porém, a realidade muitas vezes é muito diferente (PRESSMAN, 2011, p.131).

Figura 6 - Requisitos de usuários e sistema





Fonte: Sommerville (2011, p. 58)

2.1.3 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

- Requisitos funcionais: estabelece como o sistema deve se comportar, o que deve fazer, são as funcionalidades e serviços do software, deve ser descrito detalhadamente e utilizando o caso de uso da UML, fluxograma, facilitando o entendimento das funções.

- Requisitos não funcionais: estabelece como o software agirá de acordo com as restrições de serviços ou funções oferecidas, requisitos não funcionais aplica se ao sistema como um todo, deve ser descrita detalhadamente usando um fluxograma e um caso de uso para facilitar o entendimento.

Requisitos não funcionais surgem por meio das necessidades dos usuários, as necessidades de interoperabilidade com outros softwares ou hardware, restrições orçamentarias, politicas organizacionais, legislações de privacidade e regulamento de segurança.

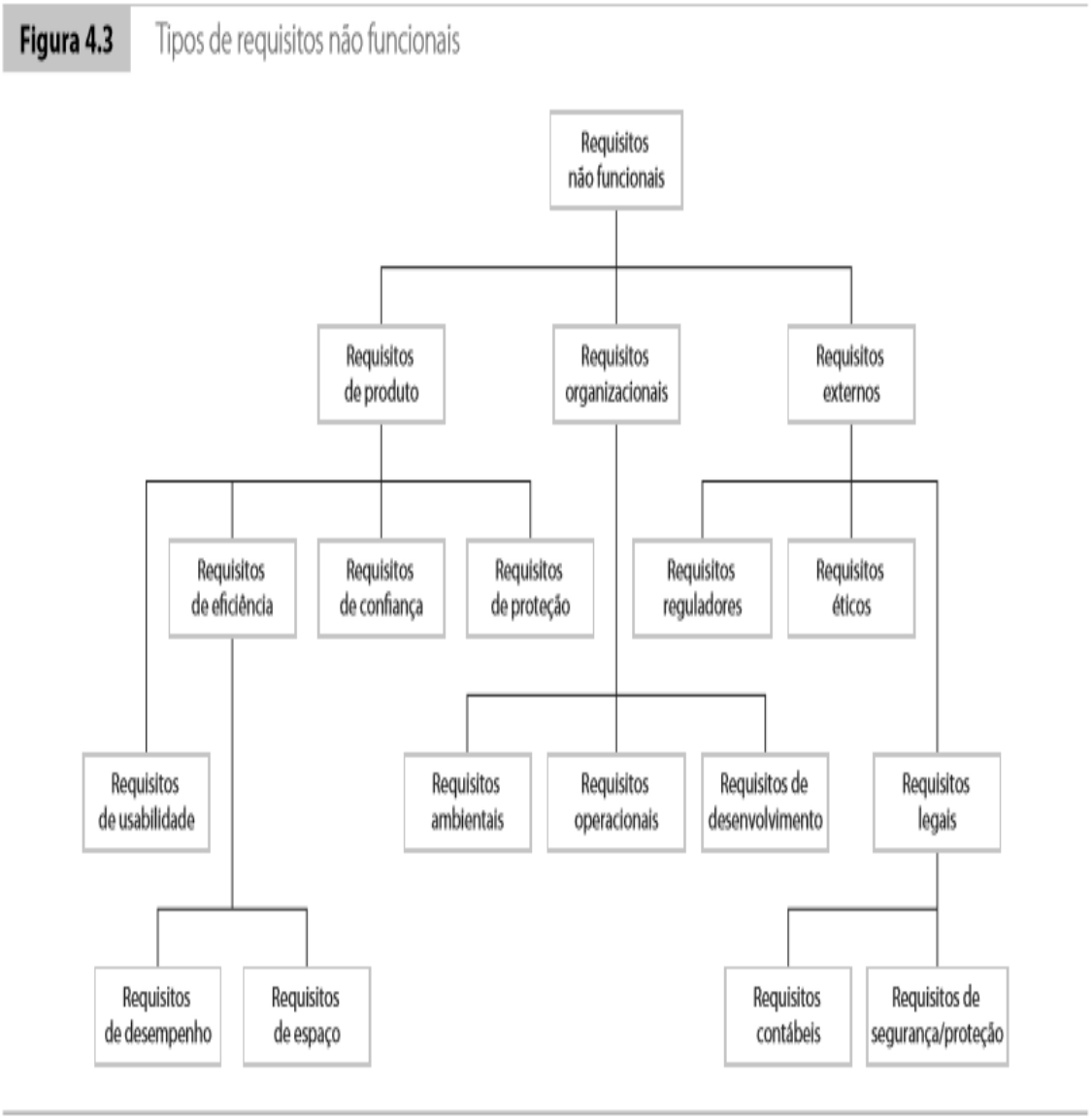
- Requisitos de projetos: tratas do negócio e entrega do produto.

- Requisitos do produto: são as técnicas de segurança e desempenho quanto à rapidez de resposta do sistema e quanto de memória será requerida, especificação, restrição e comportamento do software.

-Requisitos organizacionais: são os requisitos gerais derivados das políticas e procedimentos organizacionais da organização e do desenvolvedor, incluindo os requisitos referente em como o sistema será usado e processos operacionais, linguagem de programação, ambiente de desenvolvimento e etc...

Os requisitos não são independentes, quando se analisa mais detalhadamente o mesmo cria outros requisitos ou é dependente de um outro requisito.

Figura 7 - Requisitos não funcionais



Fonte: Sommerville (2011, p. 61)

Figura 8 - especificações de requisitos não funcionais

Tabela 4.1 Métricas para especificar requisitos não funcionais.

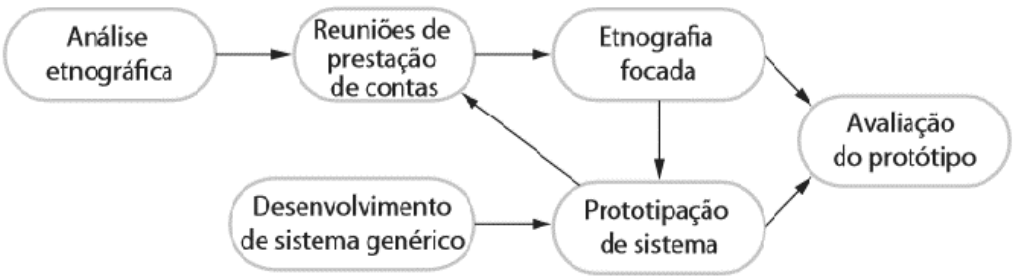
Propriedade	Medida
Velocidade	Transações processadas/segundo Tempo de resposta de usuário/evento Tempo de atualização de tela
Tamanho	Megabytes Número de chips de memória ROM
Facilidade de uso	Tempo de treinamento Número de *frames* de ajuda
Confiabilidade	Tempo médio para falha Probabilidade de indisponibilidade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

Fonte: Sommerville (2011, p. 63)

2.1.4 VALIDAÇÃO DE REQUISITOS

Figura 9 - Prototipação em análise de requisitos

Etnografia e prototipação para análise de requisitos.



Fonte: Sommerville (2011, p. 76)

> A validação de requisitos é o processo pelo qual se verifica se os requisitos definem o sistema que o cliente realmente quer. Ela se sobrepõe à análise, uma vez que está preocupada em encontrar problemas com os requisitos. A validação de requisitos é importante porque erros em um documento de requisitos podem gerar altos custos de retrabalho quando descobertos durante o desenvolvimento ou após o sistema já estar em serviço.
> O custo para consertar um problema de requisitos por meio de uma mudança no sistema é geralmente muito maior do que o custo de consertar erros de projeto ou de codificação. A razão para isso é que a ocorrência de mudança dos requisitos normalmente significa que o projeto e a implementação do sistema também devem ser alterados. Além disso, o sistema deve, posteriormente, ser reatestado (PRESSMAN, 2011- pag.76)

2.1.5 MODELAGEM DE CASO DE USO

Se usa modelagem de caso de uso em projetos de *software* para auxiliar a e licitação de requisitos. Caso de uso descreve o que o usuário espera do sistema.

Casos de uso é a descrição de interação externa com o sistema.

Caso de uso fornece uma visão simples das interações do sistema, por isso deve conter uma descrição textual descrevendo as interações e fluxos dos dados.

É uma técnica de descoberta de requisitos utilizando se de UML, o caso de uso identifica atores de interação e o respectivo nome de cada interação, contendo informações adicionais descrevendo a interação do usuário com o sistema. Informações adicionais podem ser descritas textualmente ou em modelos gráficos, como diagrama de sequência ou de estado da UML.

Casos de uso são representações em diagramas de alto nível, contendo as possíveis interações que serão descritas nos requisitos do sistema.

Casos de uso identifica as interações entre sistema e seus usuários ou entre sistemas. Todo caso de uso deve ser documentado com uma descrição textual e podendo ser ligada a outros modelos.

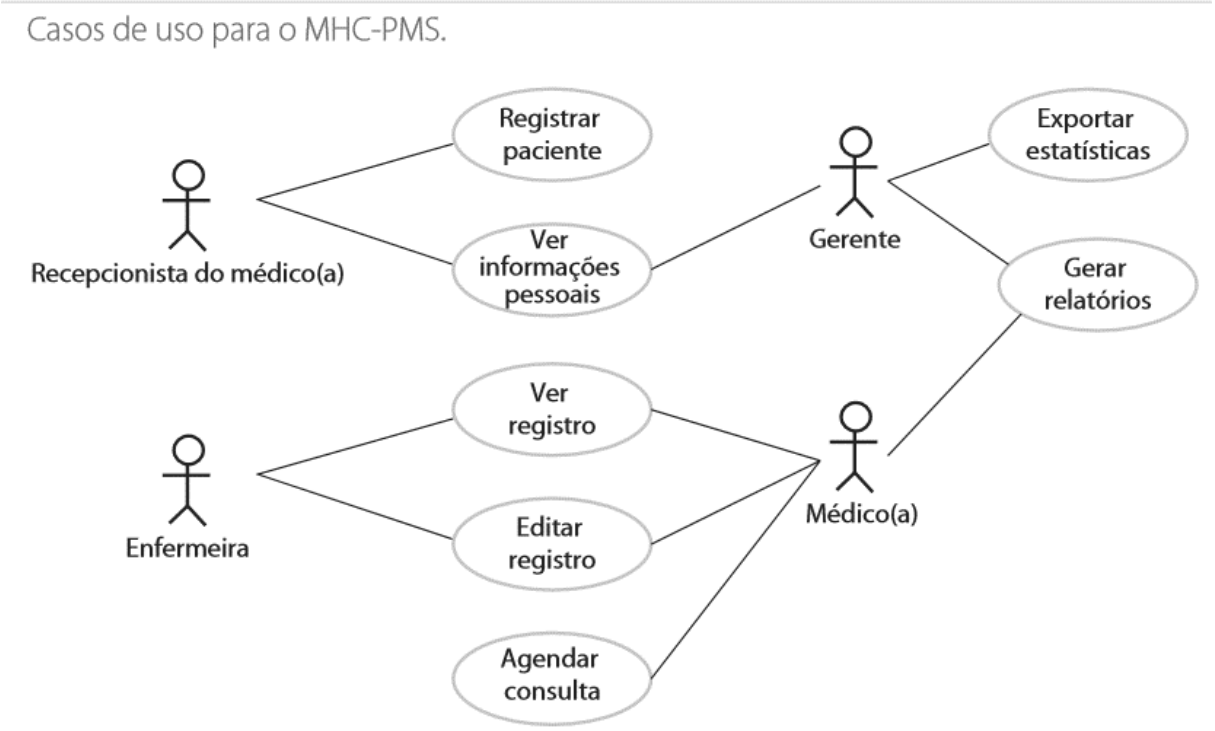
> O primeiro passo ao escrever um caso de uso é definir o conjunto de “atores” envolvidos na história. Atores são as diferentes pessoas (ou dispositivos) que usam o sistema ou produto no contexto da função e comportamento a ser descritos. Os atores representam os papéis que pessoas (ou dispositivos) desempenham enquanto o sistema opera. Definido de maneira um pouco mais formal, ator é qualquer coisa que se comunica com o sistema ou produto e que é externa ao sistema em si. Todo ator possui uma ou mais metas ao usar o sistema (PRESSMAN, 2011- pag.76).

> Os casos de uso são uma técnica de descoberta de requisitos introduzida inicialmente no método Objectory (JACOBSON et al., 1993). Eles já se tornaram uma característica fundamental da linguagem de modelagem unificada (UML — do inglês unified modeling language). Em sua forma mais simples, um caso de uso identifica os atores envolvidos em uma interação e dá nome ao tipo de interação. Essa é, então, suplementada por informações

> adicionais que descrevem a interação com o sistema. A informação adicional pode ser uma descrição textual ou um ou mais modelos gráficos, como diagrama de sequência ou de estados da UML (SOMMERVILLE, 2011, p. 74).

Exemplo: caso de uso de agendamento de consulta

Figura 10 - Caso de uso



Fonte: Sommerville (2011, p. 75)

Caso de uso é uma técnica eficaz se tratando do requisito dos '*stakeholders'*, ou seja, as pessoas que vão interagir diretamente com o sistema. Cada interação pode ser representada por um caso de uso.

A UML é um padrão de modelagem orientada a objetos, caso pode ser usado junto com outros modelos de diagrama para documentar o projeto de desenvolvimento de *software*.

2.1.6 DIAGRAMA DE SEQUÊNCIA

Diagramas de sequência em UML é utilizado para modelar as interações entre os atores e os objetos, e de objeto para objeto em um sistema computacional.

É possível modelar várias interações utilizando se da sintaxe UML.

Diagrama de sequência mostra a sequência em que as interações entre usuário cliente e usuário administrador e sistema ocorre e os passos a serem seguidos para acessar uma certa informação em particular.

> Os diagramas de interação descrevem como grupos de objetos colaboram em algum comportamento. A UML define várias formas de diagrama de interação, das quais a mais comum é o diagrama de sequência.
> Normalmente, um diagrama de sequência captura o comportamento de um único cenário. O diagrama mostra vários exemplos de objetos e mensagens que são passadas entre esses objetos dentro de um caso de uso (FOWLER, 2005, p.67).

> Os diagramas de sequência em UML são usados, principalmente, para modelar as interações entre os atores e os objetos em um sistema e as interações entre os próprios objetos. A UML tem uma sintaxe rica para diagramas de sequência, que permite a modelagem de vários tipos de interação. Como não tenho espaço para cobrir todas as possibilidades aqui, concentro-me nos fundamentos desse tipo de diagrama (SOMMERVILLE, 2011, p. 87).

Exemplo:

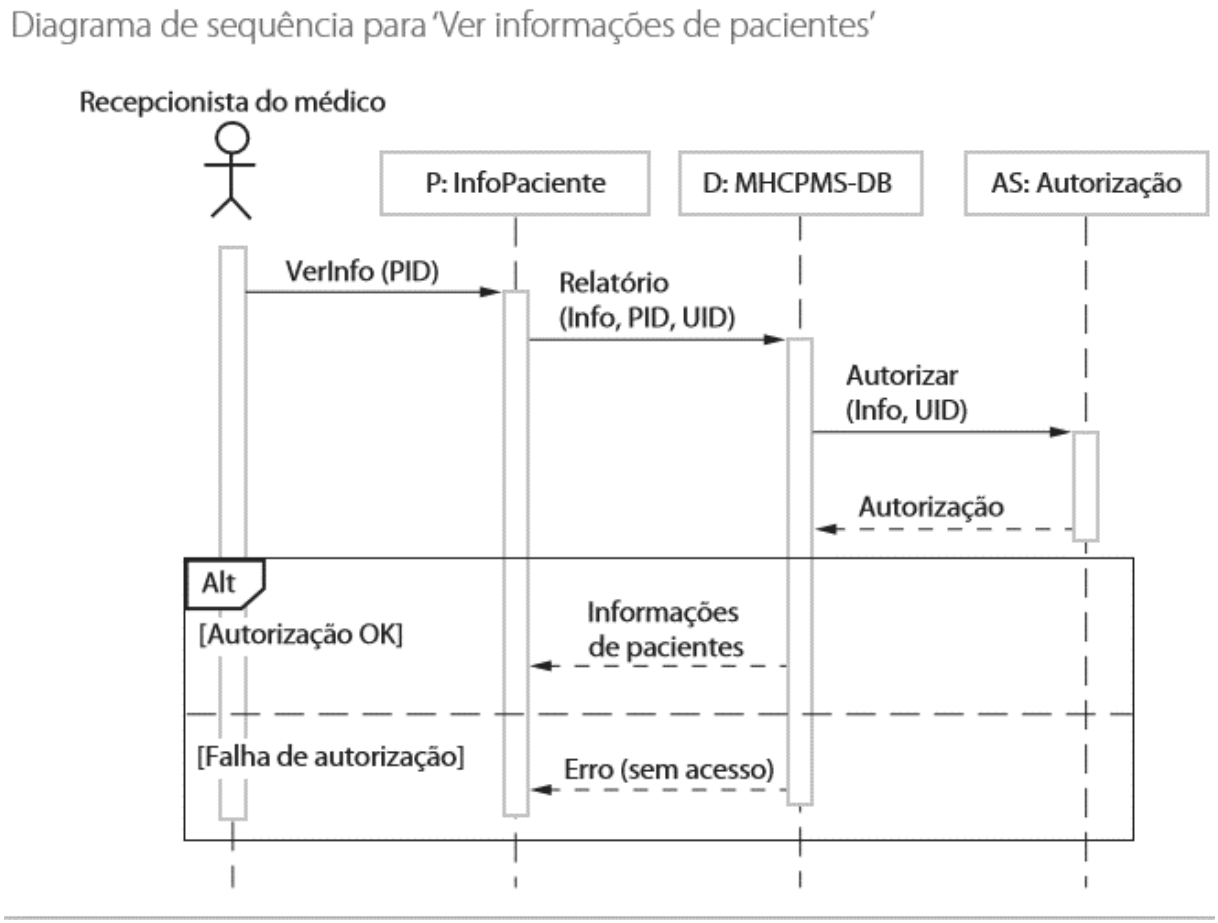
Lista dos atores e objetos envolvidos estão listados abaixo.

Tabela 2.1 Descrição tabular do caso de uso ‘Transferir dados’

Atores	Recepcionista do médico, sistema de registros de pacientes (PRS, do inglês *patient records* system)
Descrição	Uma recepcionista pode transferir dados do MHC-PMS para um banco de dados geral de registros de pacientes mantido por uma autoridade de saúde. As informações transferidas podem ser atualizadas com as informações pessoais (endereço, telefone etc.) ou com um resumo do diagnóstico e tratamento do paciente.
Dados	Informações pessoais do paciente, resumo do tratamento.
Estímulos	Comando de usuário emitido pela recepcionista do médico.
Resposta	Confirmação de que o PRS foi atualizado.
Comentários	A recepcionista deve ter permissões de proteção adequadas para acessar as informações do paciente e o PRS.

Fonte: SOMMERVILLE (2011, p.87)

Figura 11 - Diagrama de sequência



Fonte: Sommerville (2011, p. 88)

2.1.7 DIAGRAMA DE CLASSE

Se usa diagrama de classe em desenvolvimento orientado a objetos mostrando as classes de um *software* e suas associações. Os objetos são representações de algo no mundo real, como o paciente, receita e médico. Os diagramas de classes expressão diferentes níveis de detalhamento.

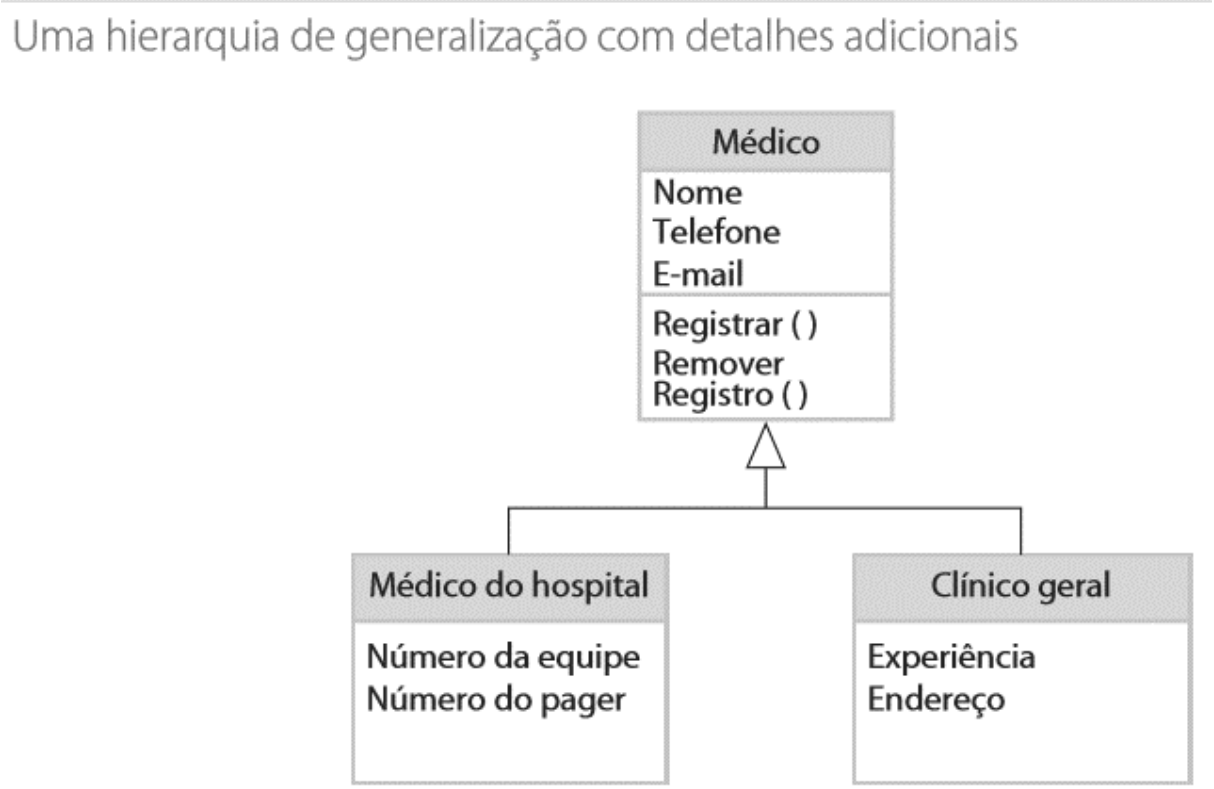
> Um diagrama de classes descreve os tipos de objetos presentes no sistema e os vários tipos de relacionamentos estáticos existentes entre eles. Os diagramas de classes também mostram as propriedades e as operações de uma classe e as restrições que se aplicam à maneira como os objetos estão conectados. A UML utiliza a palavra característica como um termo geral que cobre as propriedades e operações de uma classe (FOWLER, 2005, p.52).

> Os diagramas de classe em UML podem ser expressos em diferentes níveis de detalhamento. Quando você está desenvolvendo um modelo, o primeiro estágio geralmente é o de olhar para o mundo, identificar os objetos

> essenciais e representá-los como classes. A maneira mais simples de fazer isso é escrever o nome da classe em uma caixa. Você também pode simplesmente observar a existência de uma associação, traçando uma linha entre as classes (SOMMERVILLE, 2011, p. 87).

Exemplo:

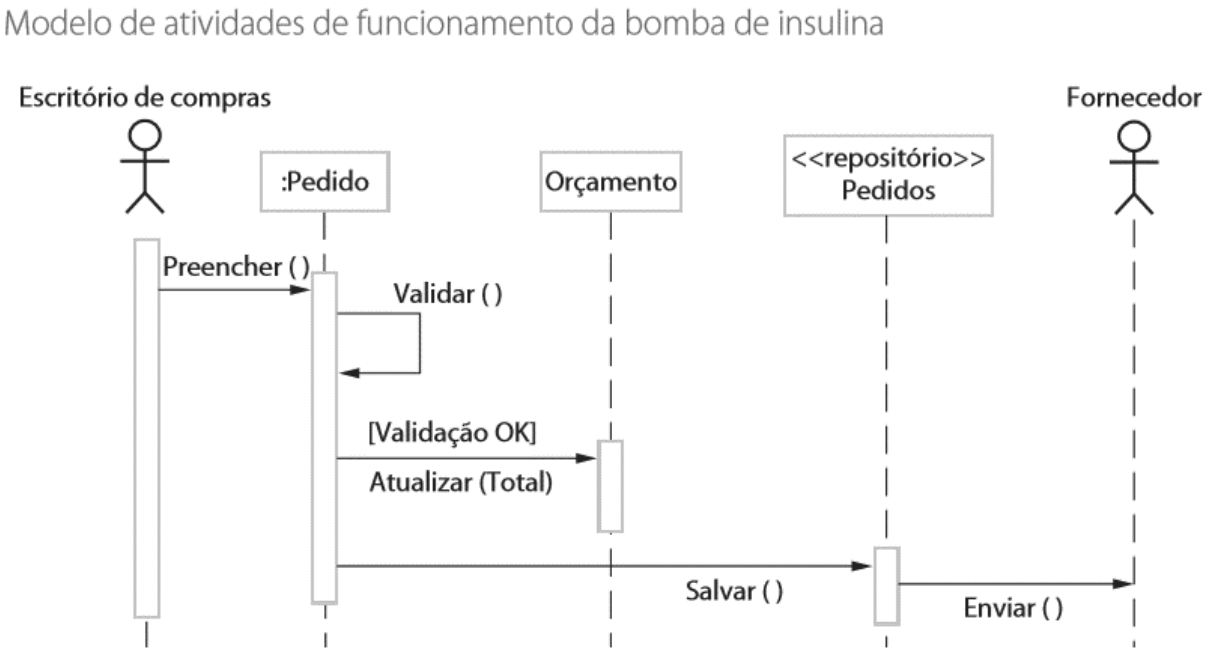
Figura 12 – Diagrama de classe



Fonte: Sommerville (2011, p. 93)

2.1.8 DIAGRAMA DE ATIVIDADE OU MODELAGEM ORIENTADA A DADOS

Figura 13 – Modelo de atividades



Fonte: Ian Sommerville (2011, p. 94)

Modelos dirigidos a dados mostram a sequência de ações envolvidas no processamento de dados de entrada e a geração de uma saída associada. Eles são particularmente úteis durante a análise de requisitos, pois podem ser usados para mostrar, do início ao fim, o processamento de um sistema. Ou seja, eles mostram toda a sequência de ações, desde uma entrada sendo processada até a saída correspondente, que é a resposta do sistema.

A UML não oferece apoio a diagramas de fluxo de dados, pois estes foram inicialmente propostos e usados para modelagem de processamento de dados. A razão para isso é que os DFDs se centram sobre as funções do sistema e não reconhecem os objetos do sistema. No entanto, devido aos sistemas dirigidos a dados serem tão comuns no mundo dos negócios, a UML 2.0 introduziu diagramas de atividades, semelhantes a diagramas de fluxo de dados.

Exemplo; a Figura abaixo mostra a cadeia de processamento envolvida no software da bomba de insulina. Nesse diagrama, você pode ver as etapas de processamento (representadas como atividades) e os dados fluindo entre essas etapas (representadas como objetos) (PRESSMAN, 2011- pag.94).

2.1.9 ESPECIFICAÇÃO DE ARQUITETURA DE *SOFTWARE*

O conceito de Arquitetura de *software* surgiu nos anos 60 com “*Dijkstra*”, e se tornou popular nos anos 90.

Arquitetura é = Elementos, Organização e decisão.

Consiste na definição dos componentes do *software*, suas propriedades externas e seus relacionamentos com outros *softwares*. Refere se também a documentação da arquitetura do *software*, relacionando a documentação entre os *stakeholders*, registrando as decisões e permitindo o reuso dos componentes do projeto. Também se refere à documentação da arquitetura de *software*. Documentação é o facilitador de comunicação entre os *stakeholders*, registrando as decisões e permitindo reuso dos componentes e padrões entre projetos.

A Arquitetura de *Software* orienta na escolha dos algoritmos e estruturas de dados, envolvendo; decisões, protocolos de comunicação, sincronização, acesso aos dados, atribuição de funcionalidades do *software*, distribuição física dos elementos, desempenho e escalabilidade.

> A arquitetura não é o software operacional, mas sim, uma representação que nos permite (1) analisar a efetividade do projeto no atendimento dos requisitos declarados, (2) considerar alternativas de arquitetura em um estágio quando realizar mudanças de projeto ainda é relativamente fácil e (3) reduzir os riscos associados à construção do software (PRESSMAM, 2011, p. 230).

> Arquitetura dirigida a modelos (KLEPPE et al., 2003; MELLOR et al., 2004; STAHL e VOELTER, 2006) é uma abordagem para projeto e implementação de software centrada em modelos, que usa um subconjunto de modelos da UML para descrever um sistema. Nesta abordagem, são criados modelos em diferentes níveis de abstração. De um modelo independente de plataforma em nível alto é possível, em princípio, gerar um programa de trabalho sem intervenção manual (SOMMERVILLE, 2011, p.98).

As informações representadas pela arquitetura de *software*, é filtrada e organizada, para facilitar o entendimento por parte dos interessados e auxiliar na tomada de decisões complexas e manutenção do *software*.

- Processos de Arquitetura de *Software*:

- Entendimento dos requisitos: levantamento dos requisitos e modelagem do domínio.

- Elaboração do modelo de negócio: analise de custo, tempo de desenvolvimento, restrições de mercado, interfaces com outros *softwares*.

- Criação ou seleção de uma arquitetura: identificação dos componentes e suas interações, dependências de construção e tecnologias que apoiam a implementação.

- Estilo Arquitetural, serve de apoio para os sistemas de grande porte facilitando à compreensão do projeto e comunicação entre os envolvidos no projeto. Estilo arquitetural oferece: suporte a atributos de qualidade ou requisitos não funcionais, diferenciação entre arquiteturas, menos esforço para entender o projeto e reuso de arquitetura e conhecimento em novos projetos. A característica da arquitetura de *software*: é possibilitar à identificação dos;

- Componentes; identifica principais elementos funcionais. Exemplo: autenticação do usuário em uma aplicação web.

- Mecanismo de interação; os componentes interagem entre si por meio de troca de mensagens, comunicação entre objeto.

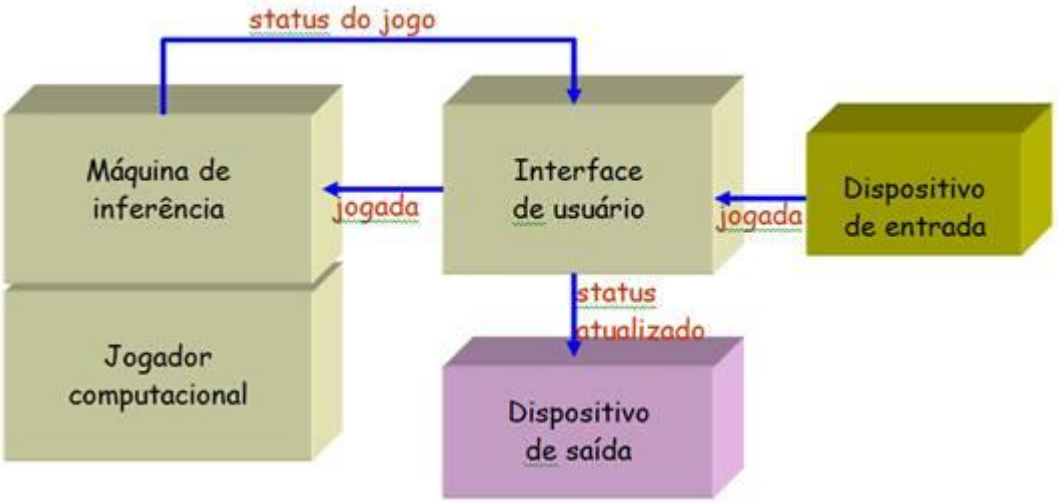
- Propriedades, analise e organização dos componentes e mecanismos de interação adquirido por cada estilo da organização. O estilo arquitetural considera o *software* por completo, permitindo que o Arquiteto ou Engenheiro de *software* determine como o *software* deve está organizado, caracterizando seus componentes e interações, também compreende o vocabulário de componentes, conectores e topologia empregada.

- Estilo Combinado de Objetos e Eventos

São mais utilizados em jogos online, onde é possível o jogador jogar com à máquina, computador ou melhor dizendo “Componente Jogador Computacional”, *(*que dispõe de recursos de inteligência artificial para simular um jogador humano)*.*

O componente (máquina) de inferência contém uma classe para tratar as jogadas feitas pelos jogadores bem como determinar a melhor jogada para o jogador computacional.

Figura 14 – Componentes de arquitetura



Fonte: Ferreira (2016, p. 18)

A combinação estilo arquitetural orientado para eventos e objetos permite a decomposição de um sistema em termos de objetos (componentes de inferência, componente de interface gráfica e componente jogador computacional) que são mais independentes além de possibilitar que atividades de computação e coordenação (de eventos) sejam realizadas separadamente. Há ainda a facilidade de reuso e manutenção, já que novos objetos podem ser facilmente adicionados. Essa característica, e interfaces bem definidas, facilitam ainda a integração.

O mecanismo de invocação é não determinístico (isto é, ocorre de forma aleatória) uma vez que considera a recepção de eventos. Adicionalmente, os componentes têm seus dados preservados de qualquer modificação acidental já que essas informações são encapsuladas em objetos, facilitando também a integração.

- Reuso de Arquitetura de *Software*

Arquiteturas que fornecem à organização e o modelo de coordenação, vai permitir seu reuso em diversos *softwares*, podendo aproveitar os componentes já testados em outros sistemas. Obs.: (o reuso de componentes e artefatos só são possíveis quando a arquitetura orienta o processo de desenvolvimento de *software*).

O quadro abaixo apresenta as principais características da Arquitetura de *Software*.

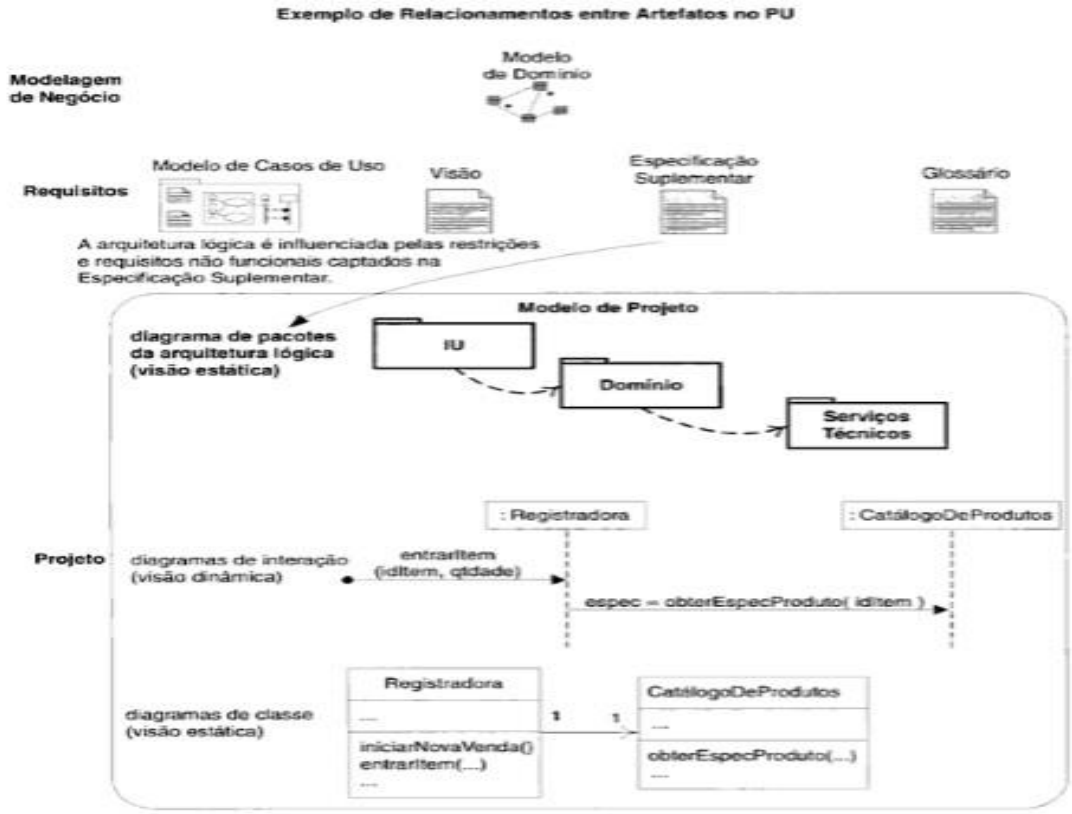
Tabela 3 – Arquitetura de *software*

Características de arquitetura de software	***Uso prático da arquitetura de software***
Constitui um artefato reutilizável	Como um arquiteto de softwares pode organizar o projeto e código de um sistema
Dispõe de mecanismos de interconexão	Como um arquiteto avalia e implanta arquiteturas de *software* em sistemas
Oferece um vocabulário de projeto e separa funcionalidades	Como um arquiteto de softwares atua no processo de desenvolvimento de *software*
Vincula o projeto a atributos de qualidade	Como um arquiteto avalia a qualidade do código baseada em métricas de produto
Suporta o desenvolvimento baseado em componentes e linha de produto (quando os requisitos são considerados para uma família de sistemas)	Como usar arquitetura como parâmetro para reduzir custos de manutenção e amortizar custos de desenvolvimento

Fonte: Ferreira (201, p. 45)

3 CONCEITO DE PROJETOS E QUALIDADE DE SOFTWARE

Figura 3.1: Ilustração da AL como parte do modelo de projeto



Fonte: Craig larman (2005, p.220)

A atividade de projeto de software engloba o conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto com alta qualidade. Os princípios de projeto estabelecem uma filosofia que prevalece sobre as atitudes e ações do desenvolvimento, orientando as atividades para realizar o projeto. Os conceitos de projeto devem ser estabelecidos e entendidos antes de aplicar a prática de projeto, que deve levar à criação de várias representações do software que servem como guia para a atividade de construção que se segue (PRESSMAM, 2011, p. 206).

O projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado e com a estrutura geral desse sistema. No modelo do processo de desenvolvimento de software, o projeto de arquitetura é o primeiro estágio no processo de projeto de software, como mostra o Capítulo 2. É o elo crítico entre o projeto e a engenharia de requisitos, pois identifica os principais componentes estruturais de um sistema e os relacionamentos entre eles. O resultado do processo de projeto de arquitetura é um modelo de arquitetura que descreve como o sistema está organizado em um conjunto de componentes de comunicação (SOMMERVILLE, 2011, p.103).

A influência de artefatos de PU, enfatizando a arquitetura lógica (AL), é mostrada na figura abaixo Diagrama de pacotes UML podem ilustrar a AL como parte do modelo de projeto – e também ser resumidos como uma visão no Documento de Arquitetura do Software. A principal entrada são as forças arquiteturais captadas na Especificação Suplementar. A AL define os pacotes dentro dos quais as classes do software são definidas (KRUCHTEN, 2005, P.220).

3.1 CONCEITO DE PROJETO

É um conjunto de princípios e práticas que levam ao desenvolvimento de *software* ou um produto de qualidade.

O que é projeto?

Projeto é a reunião das ideias referente a construção de algo novo ou melhoramento de algo já existente, essa reunião de ideias se baseia em unir o que o cliente deseja, como deve ser feito e como vai ficar após finalizar o projeto. O projeto cria representação, modelo de *software* a ser desenvolvido, o que deve ser feito com os dados, funções e comportamento do sistema. São os engenheiros de *software* que conduzem cada uma das etapas do projeto. O projeto permite modelar o *software* ou produto a ser construído, dessa forma se permite avaliações em termos de qualidade e aperfeiçoamento antes da implementação do código.

> Alguém resolve seus problemas. Neste capítulo, você saberá por que (e como) pode explorar o conhecimento e as lições aprendidas por outros desenvolvedores que tiveram o mesmo problema de design e sobreviveram. Antes de terminarmos, vamos analisar o uso e os benefícios dos padrões de projetos, ver alguns princípios importantes de design OO e dar um exemplo de como um padrão funciona. A melhor maneira de usar os padrões é carregar seu cérebro com eles e depois reconhecer locais em seus designs e aplicativos existentes onde eles possam ser aplicados. Em vez da reutilização de código, com os padrões você obtêm a reutilização de experiência (PATTERNS, 2ª ed, p. 25).

> O que é? Projeto é o que quase todo engenheiro quer fazer. É o lugar onde a criatividade impera-onde os requisitos dos interessados, as necessidades da aplicação e considerações técnicas se juntam na formulação de um produto ou sistema. O projeto cria uma representação ou modelo do software, mas diferentemente do modelo de requisitos (que se concentra na descrição do “O que” é para ser feito: dos dados, função e comportamento necessários), o modelo de projeto indica “O Como” fazer, fornecendo detalhes sobre a arquitetura de software, estruturas de dados, interfaces e componentes fundamentais para implementar o sistema (PRESSMAM, 2011, p.206).

O projeto representa o *software* de diferentes formas, tais formas estão a; arquitetura do *software* ou produto, modelagem das interfaces e os componentes de *software* utilizado para a construção do *software*. Cada uma dessas representações representa diferentes etapas e atividades do projeto. O modelo de projeto é avaliado pela equipe do projeto, com o intuído de investigar inconsistência, omissões, restrições de orçamento e prazo de entrega do produto estão de acordo. O projeto de sistema muda constantemente à medida que os métodos vão surgindo, análise e entendimento dos requisitos e métricas imposta pelos usuários vão se aperfeiçoando.

Projeto de *software* é a última ação da engenharia de *software* das etapas de modelagem à construção ‘geração de código e teste’.

Projeto de *software* é um processo interativo através do qual os requisitos são traduzidos em uma planta para construir o sistema. No início a planta representa uma visão holística do *software*. Em um alto nível de abstração o projeto pode ser associado diretamente ao objeto específico do sistema e aos requisitos mais detalhados de dados, funcionalidade e comportamento.

São quatro os modelos de projeto necessário para uma especificação completa de *software* ou produto a ser criado, os modelos são:

1. Projeto de dados, objetos / classes;
2. Projeto de arquitetura;
3. Projeto de Interface;
4. Projeto de componentes.

3.2 PROJETO DE DADOS / CLASSES

Sistemas orientado a objetos são mais fáceis de se realizar ações como: de manutenção e implementação de novos módulos, objetos e serviços adicionais sem que afete outras funcionalidades do sistema.

Ao desenvolver um projeto de *software* desde o conceito aos detalhes orientado a objetos, deve se seguir as seguintes atividades:

1. Compreender e definir claramente o contexto e as interações externas com o *software*.
2. Projetar a arquitetura do sistema.
3. Identificação dos principais objetos do *software*.
4. Desenvolver modelos de projetos.
5. Especificação das interfaces.

> Um sistema orientado a objetos é composto de objetos interativos que mantêm seu próprio estado local e oferecem operações nesse estado. A representação do estado é privada e não pode ser acessada diretamente, de fora do objeto. Processos de projeto orientado a objetos envolvem projetar as classes de objetos e os relacionamentos entre essas classes. Essas classes definem os objetos no sistema e suas interações. Quando o projeto é concebido como um programa em execução, os objetos são criados dinamicamente a partir dessas definições de classe. (SPRESSMAN, 2011- pag.125).

Observação:(projeto não é um processo sequencial, é um mundo ideias, propostas de soluções de acordo com a disponibilidade das informações).

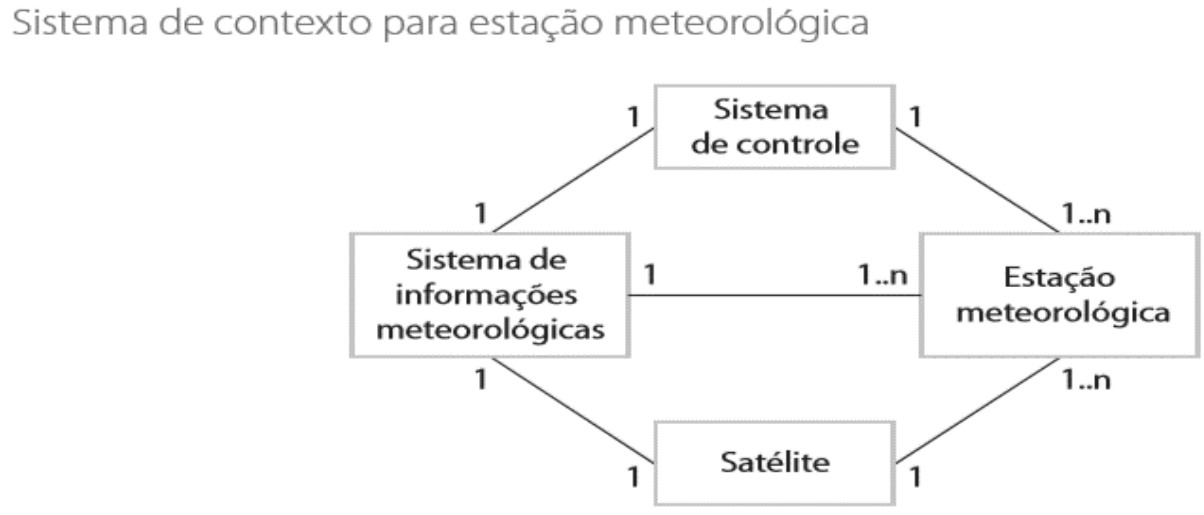
Na primeira etapa de projeto de *software* é desenvolver uma compreensão dos relacionamentos entre sistema e o ambiente externo, etapa fundamental para se ter ideia das funcionalidades que o sistema vai requerer, suas estruturas e interfaces externas. Definir limitações do sistema também ajuda na identificação de quais recursos serão implementados e quais serão associados a outro *software*.

Para se ter uma visão dos relacionamentos entre *software* e seu ambiente, é necessário fazer uso dos modelos seguintes modelos:

- Modelo de contexto estrutural, demonstrando outros *softwares* no ambiente do *software* a ser desenvolvido e pode ser representado por associações, com isso se especifica a natureza dos relacionamentos criando a documentação em forma de diagrama de blocos simples, mostrando as entidades e suas associação.

> O projeto de dados / classes transforma os modelos de classes em realizações de classes de projeto e nas estruturas de dados dos requisitos necessários para implantar o software. Os objetos e os relacionamentos definidos nos cartões CRC e no conteúdo detalhado dos dados representados por atributos de classes e outra notação fornecem a base para a realização do projeto de dados. Parte do projeto de classes pode ocorrer com o projeto da arquitetura de software (SPRESSMAN, 2011, p.207).

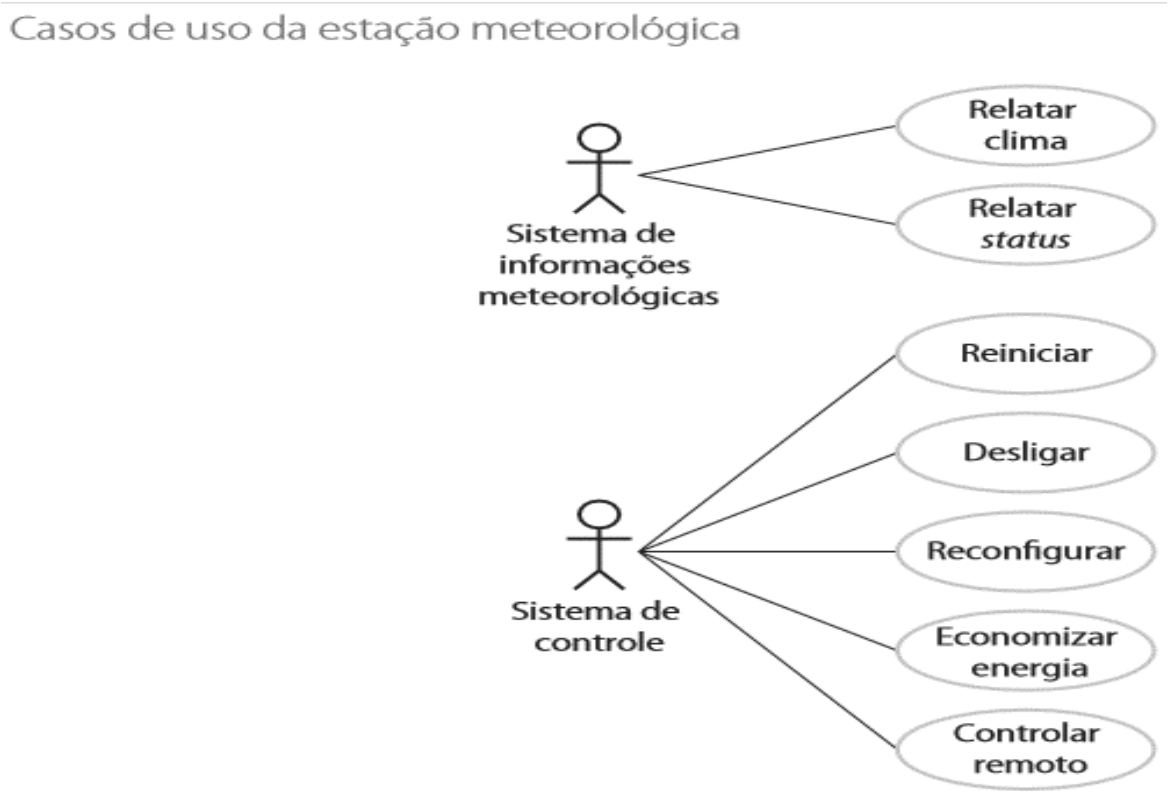
Figura 3.2.1 – Diagrama de contexto



Fonte: Ian Sommerville (2011, p. 126)

- Modelo de interação, dinâmico e mostra a interação do sistema com seu ambiente. Modelos de interações de sistema são construídos de forma abstrata sem detalhes, para se representar é necessário um modelo de caso de uso. Exemplo:

Figura 3.2.2 – Diagrama de caso de uso



Fonte: Ian Sommerville (2011, p. 127)

> Na prática, as visões conceituais são, quase sempre, desenvolvidas durante o processo de projeto e são usadas para apoiar a tomada de decisões de arquitetura. Elas são uma maneira de comunicar a essência de um sistema para os diferentes *stakeholders*. Durante o processo de projeto, quando diferentes aspectos do sistema são discutidos, outras visões também podem ser desenvolvidas, mas não há necessidade de uma descrição completa de todas as perspectivas. Também pode ser possível associar os padrões de arquitetura, discutidos na próxima seção, com as diferentes visões de um sistema (SOMMERVILLE, 2011, p. 107).

3.3 PROJETOS DE ARQUITETURA

Shaw e Garlan [Sha95a] “descrevem um conjunto de propriedades que devem ser especificadas como parte de um projeto de arquitetura” (SPRESSMAN, 2011, p. 213).

Propriedades estruturais: representação que define os componentes de software, como são empacotados e a interação entre eles.

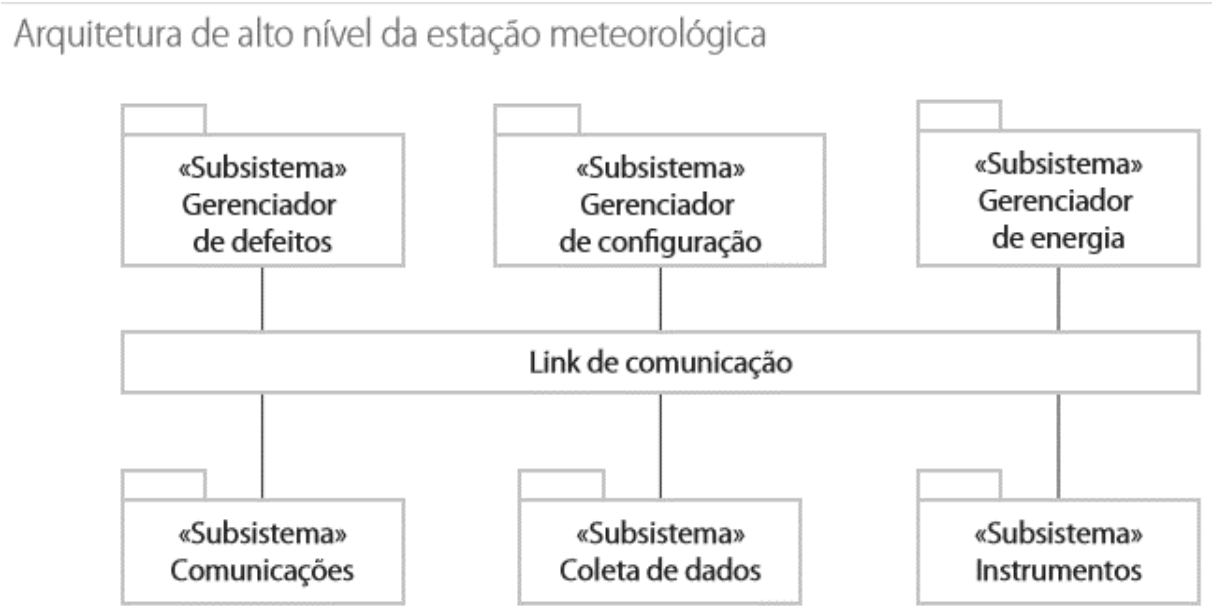
Propriedades não funcionais: são as tratativas dos requisitos de desempenho, confiabilidade, adaptabilidade, capacidade, segurança e outras características que não são acessadas pelo usuário.

Família de sistemas: é quando o projeto de arquitetura aproveita de padrões de projetos de sistemas similares, utilizando os componentes que fazem parte da arquitetura.

Após identificar os principais componentes e suas interações, é hora de organizar os componentes usando padrões de arquitetura, de modelo em camadas ou cliente-servidor. Exemplo:

> O projeto de arquitetura é um processo criativo no qual você projeta uma organização de sistema para satisfazer aos requisitos funcionais e não funcionais de um sistema. Por ser um processo criativo, as atividades no âmbito do processo dependem do tipo de sistema a ser desenvolvido, a formação e experiência do arquiteto de sistema e os requisitos específicos para o sistema. Por isso, é útil pensar em projeto de arquitetura como uma série de decisões, em vez de uma sequência de atividades (SOMMERVILLE, 2011, p. 105).
>
> A arquitetura de software refere-se à "organização geral do software e aos modos pelos quais disponibiliza integridade conceitual para um sistema" [Sha95a]. Em sua forma mais simples, arquitetura é a estrutura ou a organização de componentes de programa (modulado), a maneira através da qual esses componentes interagem e a estrutura de dados são usadas pelos componentes. Em um sentido mais amplo, entretanto, os componentes podem ser generalizados para representar os principais elementos de um sistema e suas interações (SPRESSMAN, 2011, p. 213).

Figura 3.3.1 - Arquitetura



Fonte: Ian Sommerville (2011, p. 128)

3.4 IDENTIFICAÇÃO DOS OBJETOS DE CLASSE

A descrição do caso de uso identifica objetos e operações no sistema. Há necessidade que haja objetos que representem as entradas de dados e o objeto que descrevendo o resumo dos dados e objeto do sistema de alto nível sintetizando as interações do sistema definidos no caso de uso.

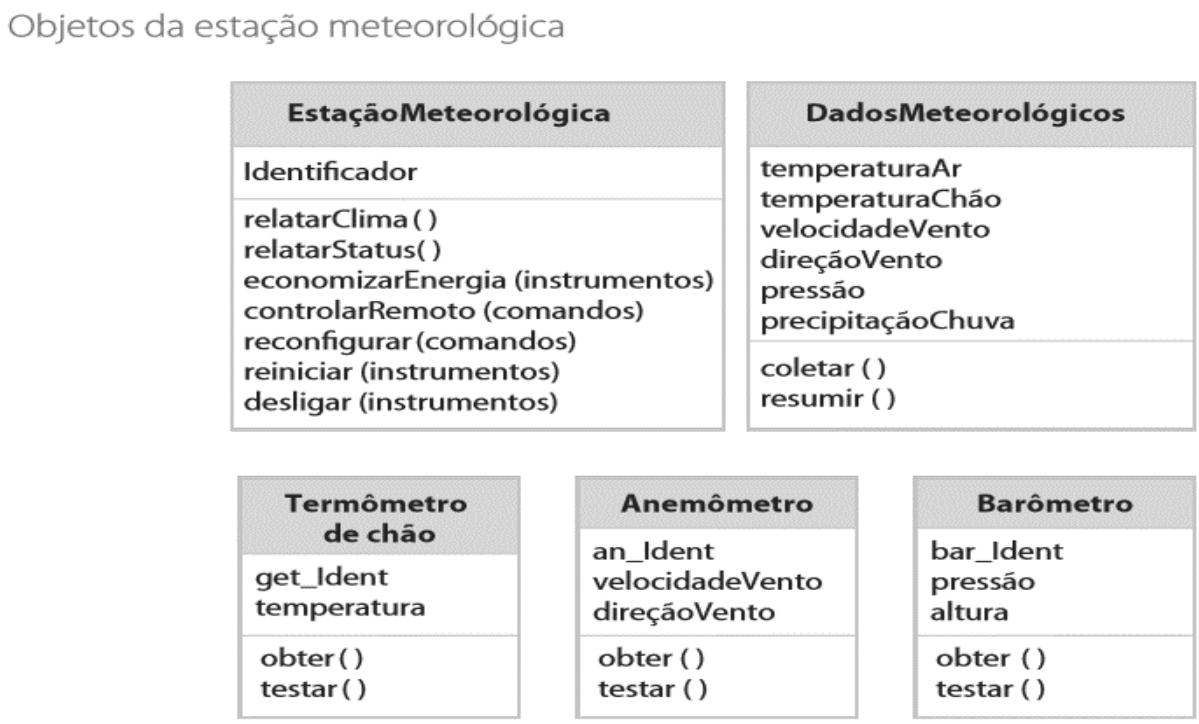
Identificando as classes de objetos de sistema:

1. Usar análise gramatical da descrição do sistema.
2. Usar entidades tangíveis no domínio de aplicação.
3. Usar uma análise baseada em cenários.

O ponto de partida de um projeto está na descrição informal do sistema e conhecimento de diversas fontes, para conseguir identificar, as classes de objetos, atributos e operações do sistema. Após conhecer o domínio de aplicação e de análise de cenário, mais formações serão usadas na refinação dos objetos. Informações essas que devem ser adquiridas nos documentos de requisitos, entrevista com o usuário, ou em análise em sistemas existentes.

Exemplo de objetos de classes:

Figura 3.4.1 – Identificação de objetos



Fonte: Ian Sommerville (2011, p. 129)

3.5 DESENVOLVER MODELOS DE PROJETOS

Com o uso da UML para desenvolver um projeto, são utilizados dois tipos de modelos de projeto.

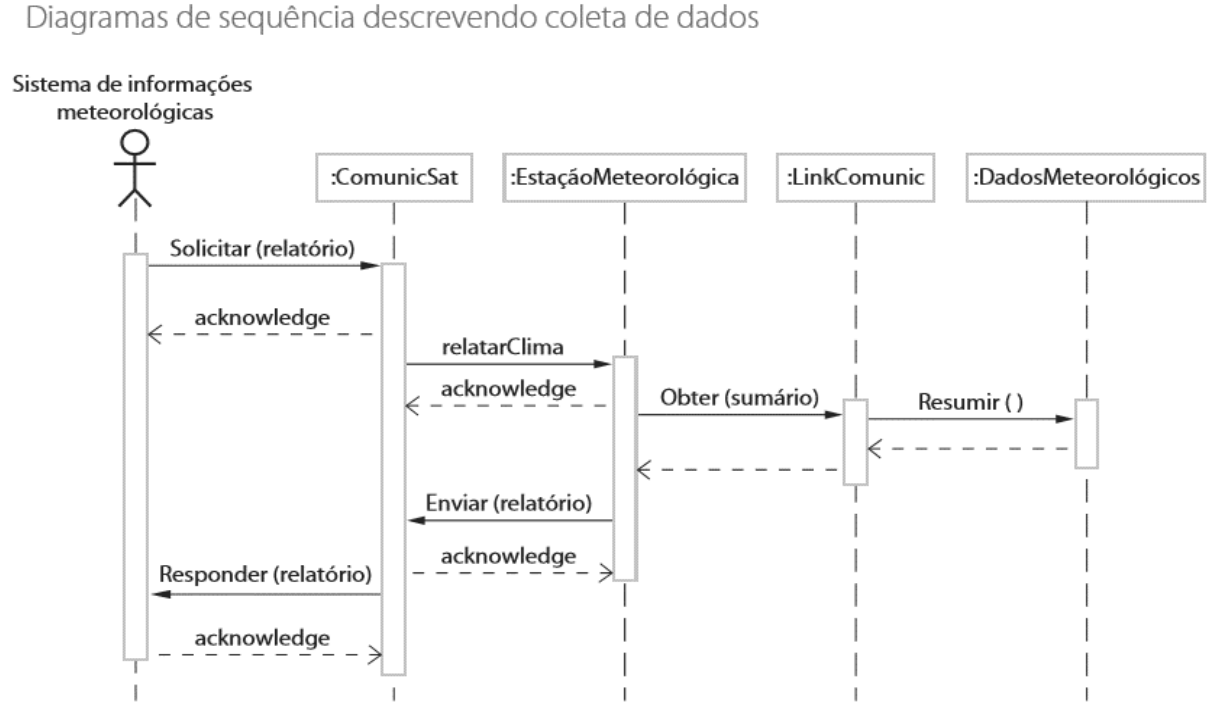
1. Modelo estrutural: que descreve estrutura estática do *software*, usando as classes de objetos e seus relacionamentos.

2. Modelo dinâmico: descrição dinâmica do sistema, mostrando as interações entre objetos do *software*, podendo ser representada por um diagrama de sequência.

> Modelos de projeto ou de sistema, mostram os objetos ou classes de objetos em um sistema. Eles também mostram as associações e relacionamentos entre essas entidades. Esses modelos são a ponte entre os requisitos do sistema e a implementação de um sistema. Eles precisam ser abstratos, para que os detalhes desnecessários não escondam os relacionamentos entre eles e os requisitos do sistema. No entanto, também devem incluir detalhes suficientes para que os programadores possam tomar decisões de implementação (PRESSMAN, 2011- pag.129).

Exemplo: diagrama de sequência, mostra sequência de interação entre o sistema externo e o interno.

Figura 19 – Diagrama de sequência



Fonte: Ian Sommerville (2011, p. 131)

3.6 GERENCIAMENTOS DE PROJETO

Gerenciar um projeto de desenvolvimento de software tem que estar focado em pessoas, produto, processo e projeto, nunca se esquecendo de que o trabalho do engenheiro de software consiste em esforço humano para se obter sucesso.

Pessoas: o People-CMM define as seguintes práticas-chave para o pessoal de software: "formação de equipe, comunicação, ambiente de trabalho, gerenciamento do desempenho, treinamento, compensação, análise de competência e de desenvolvimento, desenvolvimento de carreira, do grupo de trabalho, da cultura de equipe e outros mais" (SPRESSMAN, 2011, p.567).

Produto: é importante estabelecer os objetivos do produto, escopo, soluções alternativas e técnicas de gerenciamento antes de traçar o plano de projeto.

Processo: artefatos de software, ponto de controle e garantia de qualidade possibilitam que as metodologias sejam adaptadas as características do software e requisitos da equipe, garantindo qualidade de software, gerenciamento de configuração e medições.

Projeto: "Empregam-se projetos com planejamento e com controle por uma única e principal razão: é a única maneira de administrar a complexidade. E, mesmo assim, as equipes de software têm de se esforçar" (SPRESSMAN, 2011, p.568).

Para a maioria dos projetos existem critérios e metas, que entre elas existem quatro mais importantes.

1. Fornecer o software ao cliente no prazo estabelecido.
2. Manter os custos gerais dentro do orçamento.
3. Entregar software que atenda às expectativas do cliente.
4. Manter uma equipe de desenvolvimento trabalhando feliz.

> O que é? Embora muitos de nós (em momentos mais críticos) adotemos a visão de Dilbert, ainda resta uma atividade bastante útil quando sistemas e projetos computacionais são desenvolvidos. Gerenciamento de projeto envolve planejamento, monitoração e controle de pessoas, processos e eventos que ocorrem à medida que o software evolui desde os conceitos preliminares até sua disponibilização, operacional e completa (SPRESSMAN, 2011, p. 566).
>
> Algumas vezes, foi sugerido que a principal diferença entre a engenharia de software e outros tipos de programação é que a engenharia de software é um processo gerenciado. Dessa forma, o desenvolvimento de software ocorre dentro de uma organização, a qual está sujeita a uma série de restrições organizacionais, de orçamento e de cronograma. Portanto, o gerenciamento é muito importante para a engenharia de software. Nesta parte do livro, eu apresento uma variedade de tópicos de gerenciamento centrados em questões de gerenciamento técnico, e não em questões mais 'suaves' de

> gerenciamento, como gerenciamento de pessoas, ou o gerenciamento mais estratégico de sistemas corporativos (SOMMERVILLE, 2011, p.413).

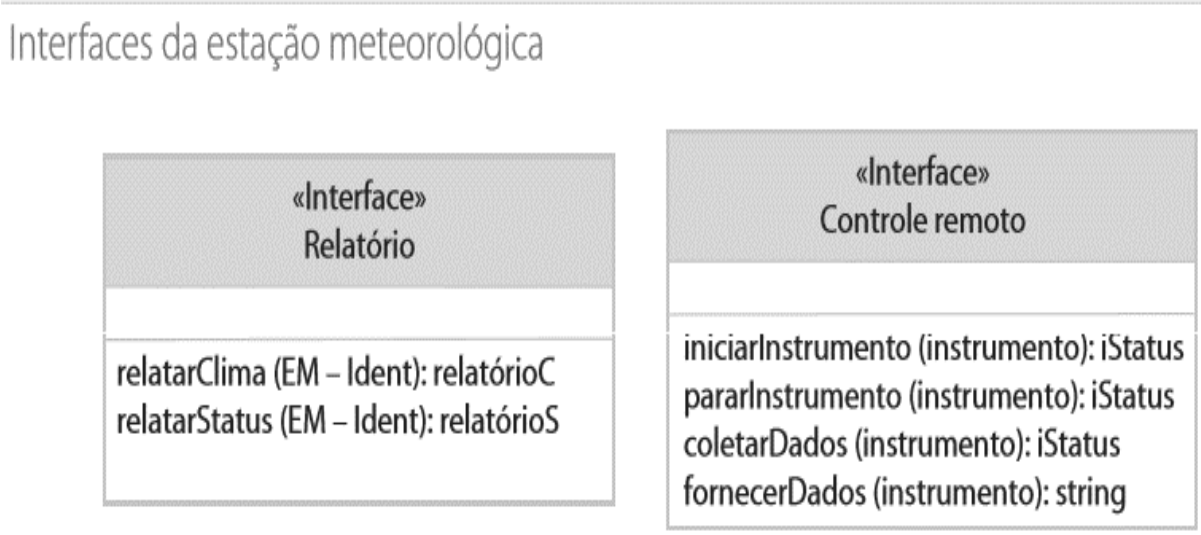
3.7 ESPECIFICAÇÕES DE INTERFACE

Em um projeto, é muito importante a especificação das interfaces e os componentes do projeto. Pois a especificação das interfaces deve permitir que os objetos e os subsistemas possam ser projetados em paralelo.

Projeto de interface tem como principal objetivo em tratar dos detalhes da interface para o objeto ou para um grupo de objetos. Deve ser incluído no projeto de interface as operações de acesso e atualização de dados, lembrando que cada objeto pode ter várias interfaces, com ponto de vista diferente aos métodos que oferecem.

Exemplo: duas interfaces que podem ser definidas para estação meteorológica.

Figura 20 – Especificação de interface



Fonte: Sommerville (2011, p. 133)

3.8 PADRÕES DE PROJETO

Na engenharia de *software*, padrão de projeto é uma solução para um problema que venha a ocorrer em um determinado contexto de projeto de sistema.

Padrão de projeto é uma descrição ou modelo de como solucionar problemas que possa vir a ocorrer em senários diferentes. Padrões são melhores práticas que o desenvolvedor possa utilizar para solucionar um problema comum que venha ocorrer em um projeto de *software*. Padrões de projetos orientados a objetos, mostram os

relacionamentos e interações entre as classes, objetos, sem especificar as classes ou objetos da aplicação final e seus envolvimentos.

Padrão é uma descrição do problema e da solução, forma que a solução possa ser reutilizada em diferentes contextos. Padrão de projeto não são descrições detalhadas, mas sim uma descrição de conhecimento e experiência, é uma solução testadas e aprovada para um problema comum.

> Os padrões são uma ótima ideia, mas, para usá-los efetivamente, você precisa de experiência de desenvolvimento de software. Você precisa reconhecer as situações em que um padrão pode ser aplicado. Programadores inexperientes, mesmo que tenham lido o livro de padrões, sempre acharão difícil decidir se podem reusar um padrão ou se é necessário desenvolver uma solução especial (SPRESSMAN, 2011- pag.133).

3.9 QUALIDADE DE *SOFTWARE*

O assunto, qualidade de *software* surgiu em 1990, quando várias empresas de grande poder aquisitivo, perceberam que estavam desperdiçando milhões em dólares com *software* que não fazia o prometido, isso deixou as empresas preocupadas com erros graves que vinham ocorrendo na indústria de *software*.

É difícil definir qualidade pelo seguinte motivo; o que é qualidade para mim, pode não ser para quem está lendo este trabalho agora, ou qualquer outra pessoa. Qualidade vareia de acordo com classificação de cada um, porem em ‘qualidade de *software’*, é algo que deve ser definido e apreenderemos agora!

Qualidade de software é um objetivo importante para todos os engenheiros e desenvolvedores de software.

> “[Roger spressman], define qualidade de software como: uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam” (SPRESSMAN, 2011- pag.360).

Definir qualidade de *software* é uma missão árdua e que deve ser sempre discutida pelos envolvidos em: projeto de software. Definição de qualidade enfatiza três pontos importantes.

1. Uma gestão de qualidade estabelece a infraestrutura que dá suporte na construção de um produto de *software* de alta qualidade.
2. Produto útil, que satisfaça as exigências definidas pelos interessados.

3. Agrega valor para o fabricante e para o usuário de um *software*, *software* de alta qualidade beneficia tanto o usuário final como o fabricante. O *software* de alta qualidade exige menos suporte ao cliente e manutenção.

> Segundo DARVID GARVIN [Gar87] sugere que a qualidade deve ser considerada adotando-se um ponto de vista multidimensional que começa com uma avaliação da conformidade e termina com uma visão transcendental (estética). Embora as oito dimensões de qualidade de Garvin não tenham sido desenvolvidas especificamente para software, elas podem ser aplicadas quando se considera qualidade de software (SPRESSMAN, 2011- pag.361).

Qualidade de desempenho, o *software* gera valor ao usuário, fornecendo as funções e recursos especificados nos requisitos.

Qualidade de recursos, quem utiliza o *software* pela primeira fez, se impressiona pelos recursos oferecidos.

Confiabilidade, o *software* está sempre disponível, oferecendo todos os recursos sem falhas.

Conformidade, o *software* interage perfeitamente com aplicações externas e internas, colocando em práticas regras abordadas no projeto.

Durabilidade, o sistema pode ser mantido ou atualizado sem gerar indisponibilidade ou erros em algum de seus módulos.

Facilidade de manutenção, o suporte técnico tem acesso a todas as informações do sistema para fazer modificações ou reparos.

Estética, depende da classificação de cada um.

Percepção, vai depende do histórico do fabricante e da mente aberta de quem está adquirindo o *software*.

3.9.1 FATORES DE QUALIDADE ISO 9126

Norma ISSO para qualidade de *software*, define um conjunto de parâmetros com o objetivo de padronização e avaliação da qualidade do produto de *software*.

A norma brasileira correspondente é a NBR 13596 (substituída pela NBR ISSO/IEC 9126-1). Norma desenvolvida com o intuito de identificar os atributos fundamentais para sistemas de computadores. Estabelecendo os seguintes componentes, como modelo de qualidade:

- Processo de Desenvolvimento: a qualidade do produto é influenciada pela natureza do produto.

- Produto: compreende os atributos de qualidade do sistema, que são; atributos internos e externos.
- Qualidade em uso: qualidade do usuário.

3.1.1 CONSIDERAÇÕES FINAIS

No primeiro capitulo desse TCC, foi abordado os assuntos ligado a engenharia de software e seus processos, onde foi feito uma revisão bibliográfica referente classificação de software, tipos de sistemas, processos de software modelos de processos; tais como cascata, prototipação, interativo, incremental, espiral e desenvolvimento ágil.

No segundo capítulo foi abordado arquitetura, modelagem e requisitos. Foi falado dos vários princípios como os princípios orientado a pratica, processos, planejamento, modelagem de projetos e requisitos. Requisitos funcionais e não funcionais, validação de requisitos e os diversos modelos de arquitetura como os de caso de uso, sequência, de classes e atividades.

O terceiro e ultimo capítulo foi falado de projetos de dados, arquitetura. Identificação de objetos e classes. Padrões de projeto, interfaces e a qualidade de software.

O trabalho de conclusão de curso, foi uma experiência fantástica e muito difícil de realizar.

Tive muitos contratempos com as formatações em normas ABNT em como fazer as citações corretamente, sem contar na exigência em citar quatro autores diferentes por capítulos e subcapitulo.

Também tive problemas com o tempo que é curto.

O objetivo desse TCC, foi alcançado pelo seguinte proposito:

Explicar e esclarecer o passo a passo para se realizar um projeto de software, utilizando a engenharia de software como um manual de boas práticas, mostrando diversos níveis de engenharia, desde a requisição de requisitos em entrevista à própria análise de requisitos.

A engenharia de software abordada nessa monografia deixou claro os vários processos e modelos utilizados em projetos de desenvolvimento de software.

Referências

Engenharia de Software / Ian Sommerville; tradução Ivan Bosnic e Kalinka G. De O. Gonçalves; revisão técnica Kechi Hirama. — 9ª. Ed. — São Paulo: Pearson Prentice Hall, 2011.

Engenharia de Software – **Uma Abordagem Profissional** – 7ª Edição / Roger S. Pressman, PH.D.

Bem Soher, https://guiadoestudante.abril.com.br/orientacao-profissional/qual-a-diferenca-entre-ciencia-da-computacao-e-engenharia-de-software/

Silva,https://guiadoestudante.abril.com.br/orientacao-profissional/faco-engenharia-da-computacao-ou-ciencia-da-computacao/

Livro**: Use a cabeça, Padrões de Projeto** 2ª Edição